

Data Structures and Algorithms

Lesson 1: **Data Structures and Algorithms using Java**

[Data Structures and Algorithms Overview](#)

[Algorithm Performance](#)

[Constant Performance](#)

[Logarithmic Performance](#)

[Linear Performance](#)

[Quadratic Performance](#)

[Comparing Classification Families](#)

[Lessons Learned](#)

[ArrayList Amortized Reallocation](#)

[Quiz 1 Project 1](#)

Lesson 2: **Data Structures and the Java Collections Framework**

[Introduction to Java Collections Framework](#)

[Set Interface](#)

[List Interface](#)

[Queue Interface](#)

[Map Interface](#)

[Summarizing the Implementations You Need To Know](#)

[Important Methods For Keys And Values](#)

[Lessons Learned](#)

[Quiz 1 Project 1](#)

Lesson 3: **Algorithms Using Java**

[Designing Algorithms](#)

[Skyline Problem](#)

[Lessons Learned](#)

[Quiz 1 Project 1](#)

Lesson 4: **Working With Big Data**

[Working with Big Data](#)

[Sorting Large Sets Using External Storage](#)

[Characterizing Storage Requirements for an Algorithm](#)

[MergeSort with \$O\(n\)\$ Storage Requirements](#)

[Working with Large Datasets](#)

[Never Be Satisfied](#)

[Lessons Learned](#)

[Quiz 1 Project 1](#)

Lesson 5: **Representing Graph Data Structures**

[Representing Graphs](#)

[Using Adjacency Matrix To Represent Graph](#)

[Searching a Graph](#)

[Practical Application](#)

[Lessons Learned](#)

[Quiz 1 Project 1](#)

Lesson 6: **Graph Adjacency List and Shortest Path Algorithms**

[Searching For Optimal Paths](#)

[Representing Graph By Adjacency List](#)

[Breadth-First Search](#)

[Lessons Learned](#)

[Quiz 1 Project 1](#)

Lesson 7: **Priority Queues**

[Priority Queue Data Structure](#)

[Minimum Spanning Tree](#)

[Heap Data Structure](#)

[Prim's Algorithm Implementation](#)

[Evaluating Minimum Spanning Tree Implementations](#)

[Lessons Learned](#)

[Quiz 1 Project 1](#)

Lesson 8: **Binary Tree Data Structure**

[Binary Tree Data Structure](#)

[Naive Binary Tree Implementation](#)

[Evaluating Binary Tree Implementation](#)

[Rebalancing Binary Trees](#)

[Using Collections TreeSet](#)

[Lessons Learned](#)

[Quiz 1 Project 1](#)

Lesson 9: **Multidimensional Algorithms**

[A Data Structure For Multidimensional Algorithms](#)

[Traversing a kd-tree](#)

[Using kd-trees to Search for Points](#)

[Lessons Learned](#)

[Project](#)

[Quiz 1 Project 1](#)

Lesson 10: **Mathematical Algorithms and Floating Point Computations**

[Mathematical Algorithms and Floating Point Computations](#)

[Gauss Jordan Elimination](#)

[Rounding Errors](#)

[Partial Input Data](#)

[Matrix Determinant](#)

[Lessons Learned](#)

[Quiz 1 Project 1](#)

Lesson 11: **Brute Force Algorithms**

[Using Brute Force To Solve Permutation Problems](#)

[Finding All Five-Letter words in PALINDROME](#)

[N Queens Problem](#)

[Lessons Learned](#)

[Quiz 1 Project 1](#)

Lesson 12: **Path Finding for Single-Player Games**

[Path Finding For Single-Player Games](#)

[Breadth-First Search](#)

[Evaluating Search Tree Algorithms](#)

[Lessons Learned](#)

[Quiz 1](#) [Project 1](#)

Lesson 13: **[Path Finding for Two-Player Games](#)**

[Path Finding For Two-Player Games](#)

[Minimax Implementation](#)

[Lessons Learned](#)

[Quiz 1](#) [Project 1](#)

Lesson 14: **[Algorithms On Sound Data](#)**

[Signal Processing Algorithms](#)

[Composed Wave Forms](#)

[Analyzing Composed Wave Forms](#)

[Using FFT on WAV file samples](#)

[Lessons Learned](#)

[Quiz 1](#) [Project 1](#)

Lesson 15: **[Conclusion](#)**

[Concluding Lesson For Algorithms](#)

[Removing Elements From a Sorted Array](#)

[Removing Elements From Binary Search Trees](#)

[Removing Elements From AVL Trees](#)

[Removing Elements From KD-trees](#)

[Lessons Learned](#)

[Quiz 1](#) [Project 1](#)

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Data Structures and Algorithms using Java

Welcome to the O'Reilly School of Technology course on Data Structures and Algorithms Using Java!

Course Objectives

When you complete this course, you will be able to:

- identify the core data structures provided by the JDK.
- identify appropriate data structures based on problems you are likely to face.
- explain the essential design techniques necessary for developing algorithms.
- develop algorithms that efficiently process data.
- characterize the performance of an algorithm in both space and time.

In this Java course, you'll learn how to write efficient Java code, which means learning about data structures and algorithms. Here you'll refine your Java skills to identify the appropriate data structures to use when solving real-world problems. These data structures are already provided for you in the Java Development Kit (JDK) release. You'll learn key algorithms that you'll use again and again so your code performs efficiently every time.

In each lab, you'll learn about data structures and algorithms within the context of a solution to a real-world problem. Once you understand the solution, you'll demonstrate mastery by extending the existing code in a project. Throughout this course you will write Java code from scratch while solving real problems. There will also be references to **Algorithms in a Nutshell**, the associated textbook for this course. The book comes with an online code base, the Algorithms Development Kit (ADK), that can be used as a reference in addition to the code described in these lessons.

Each quiz will validate that you learned the key information and the projects and will describe likely extensions to the data structures and algorithms.

As you progress through the course, you'll write professional test cases to verify the behavior of your data structures and algorithms.

Lesson Objectives

When you complete this lesson, you will be able to:

- explain the limitation of using arrays to store dynamic collections.
- characterize the input, processing and output steps for an algorithm.
- explain why using classes to model structured information is preferred to just using multiple arrays containing primitive types.
- characterize the run-time performance of an algorithm based on the size of a problem instance.

Welcome to the O'Reilly School of Technology's course on Data Structures and Algorithms. Although it's unlikely that this sixth course in the Java series is your first OST course, we'll describe how OST works, just in case. If you already have a solid understanding of our tools and methods, feel free to skip ahead to the [Data Structures and Algorithms Overview](#) section.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add *looks like this*.

If we want you to remove existing code, the code to remove ~~*will look like this*~~.

We may also include instructive comments that you don't need to type.

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

The plain black text that we present in these INTERACTIVE boxes is provided by the system (not for you to type). The commands we want you to type *look like this*.

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

Gray "Observe" boxes like this contain **information** (usually code specifics) for you to *observe*.

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

Data Structures and Algorithms Overview

In 1976, Niklaus Wirth, the inventor of the Pascal language and a pioneering figure in Computer Science, published a fundamental textbook on Software Engineering called *Algorithms + Data Structures = Programs*. In short, he proposed that developers must understand data structures and algorithms as a prerequisite to writing efficient programs.

In your earliest programs, you no doubt used variables to store information that you processed. For example, think of the first time you wrote a program that converted temperatures between Celsius and Fahrenheit (and you know that you did). To understand how to write this program, a developer must identify the appropriate *Algorithm* and *Data Structure* to use.

An *algorithm* is a step-by-step procedure for computation that **processes input data** to produce an **output result**.

We'll highlight **input data**, **processes**, and **output results** with these colors throughout this lesson to identify the different functional parts of the algorithm implementations.

The computation for temperature conversion is a straightforward calculation, so you only need to determine the format of input. In most situations, the input is most easily represented as text strings typed by the user; however, you could also retrieve input as a binary sequence of bits, perhaps from an embedded microcontroller.

A *problem instance* is a particular input data set to which a program is applied.

In practical terms, different programs will process their input using code that decodes or translates the input into the proper data structures that they need to function. Let's try an example.

Create a new project for this lesson named **DataStruct**, and assign the **Java6_Lessons** working set to it:

New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name: **DataStruct**

☒ Use default location

Location: V:\workspace\DataStruct [Browse...](#)

JRE

☐ Use an execution environment JRE: JavaSE-1.6

☐ Use a project specific JRE: jre7

☒ Use default JRE (currently 'jre7') [Configure JREs...](#)

Project layout

☐ Use project folder as root for sources and class files

☒ Create separate folders for sources and class files [Configure default...](#)


Working sets

☒ Add project to working sets

Working sets: Java6_Lessons [Select...](#)


[?](#) [< Back](#) [Next >](#) [Finish](#) [Cancel](#)

When prompted to open the perspective, check **Remember my answer** and click **No**.

 In the **DataStruct** project **/src** source folder, create a **TemperatureConversion** class:

New Java Class

Java Class

 The use of the default package is discouraged.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

CODE TO TYPE: TemperatureConversion class

```
import java.util.*;

public class TemperatureConversion {
    public static void main(String[] args) {
        System.out.println("Enter Celsius: ");
        Scanner sc = new Scanner (System.in);

        double c = Double.valueOf(sc.nextLine());
        double f = c*9.0/5 + 32;

        System.out.println("Fahrenheit is: " + f);
    }
}
```

Right-click the class and select **Run As | Java Application**. When prompted "Enter Celsius:", enter a number. The Fahrenheit equivalent prints:

Run TemperatureConversion

```
Enter Celsius:
100
Fahrenheit is: 212.0
```

Let's take a closer look at the program.

OBSERVE: TemperatureConversion class

```
import java.util.*;

public class TemperatureConversion {
    public static void main(String[] args) {
        System.out.println("Enter Celsius: ");
        Scanner sc = new Scanner (System.in);

        double c = Double.valueOf(sc.nextLine());
        double f = c*9.0/5 + 32;

        System.out.println("Fahrenheit is: " + f);
    }
}
```

The above program conforms to the generic **Input - Process - Output** structure of most programs. The **Celsius temperature is typed by the user** and you **convert it to Fahrenheit, and multiply by 9.0/5** to retain precision in your computation. For output, **print the computed Fahrenheit value to the console**.

The world gets more complicated when you need to process groups of data. Let's extend the above program to produce a table of Fahrenheit conversions given a number of Celsius values. This first attempt stores just the collection of Celsius values and computes the Fahrenheit conversion values as needed. We'll assume that the user enters the requested values correctly; adding error-handling logic would only complicate these small programs and obscure the points we're trying to make in this lesson.

 In the **DataStruct** project **/src** source folder, create a **TemperatureConversionTable** class, and modify it as shown:

CODE TO TYPE: TemperatureConversionTable class

```
import java.util.*;

public class TemperatureConversionTable {

    public static void main(String[] args) {
        System.out.println("Enter Celsius values separated by spaces then press Enter: ");
        Scanner sc = new Scanner (System.in);
        String values = sc.nextLine();

        StringTokenizer st = new StringTokenizer(values, " ");
        double cValues[] = new double[st.countTokens()];

        int index = 0;
        while (st.hasMoreTokens()) {
            cValues[index] = Double.valueOf(st.nextToken());
            index++;
        }

        System.out.println("Celsius\tFahrenheit");
        for (int i = 0; i < cValues.length; i++) {
            double f = cValues[i]*9.0/5 + 32;
            System.out.println(cValues[i] + "\t" + f);
        }
    }
}
```

Save and run **TemperatureConversionTable** to see how it executes on a sample problem instance:

Run TemperatureConversionTable

```
Enter Celsius values separated by spaces then press Enter:
3 2.7 1.3 9.9 123.4
Celsius Fahrenheit
3.0 37.4
2.7 36.86
1.3 34.34
9.9 49.82
123.4 254.12000000000003
```

Let's look at the different functional elements of this code.

TemperatureConversionTable broken down into its parts

```
import java.text.*;
import java.util.*;

public class TemperatureConversionTable {

    public static void main(String[] args) {
        System.out.println("Enter Celsius values separated by spaces then press Enter: ");
        Scanner sc = new Scanner (System.in);
        String values = sc.nextLine();

        StringTokenizer st = new StringTokenizer(values, " ");
        double cValues[] = new double[st.countTokens()];

        int index = 0;
        while (st.hasMoreTokens()) {
            cValues[index] = Double.valueOf(st.nextToken());
            index++;
        }

        System.out.println("Celsius\tFahrenheit");
        for (int i = 0; i < cValues.length; i++) {
            double f = cValues[i]*9.0/5 + 32;
            System.out.println(cValues[i] + "\t" + f);
        }
    }
}
```

The above code uses an array of **double cValues[]** to store the Celsius values entered by the user. With arrays you need to specify the size of the collection in advance. In many cases, you either know the proper size in advance or you can set the size to be some value large enough for any conceivable program execution. The **Input** consists of Celsius values separated by spaces. You can use **StringTokenizer** to extract each of these values as a String token, which is then converted into a double value using the **Double.valueOf()** method. As an added bonus, **StringTokenizer** can tell you the total number of tokens that will be extracted using the **countTokens()** method. The above code uses an obvious array structure to store a set of singularly typed values (doubles, in this case) and you easily traverse each element in the array using a **for** loop to iterate over every element.

The output of this program is still a bit rough. Let's make some enhancements:

1. Retrieve the Celsius values from the user one per line; after that, the user simply presses **Enter**.
2. Present the conversion table sorted in ascending order.
3. Round Temperature values to two digits of precision.
4. Don't display any duplicate values in the table.

The first enhancement changes the entire **Input** phase of the program. The program no longer knows in advance how many values are to be read; rather, it must read them one at a time until told to stop. Once the entire array of values has been created, you can satisfy the second enhancement by sorting the array. Finally, you need to do some extra

processing to ensure that there are no duplicates for the fourth enhancement. The upcoming code handles all of these enhancements while still using a simple array to store its values. The structure of the code has changed to allow you to conduct testing at the end of this lesson.

CODE TO TYPE: Modified TemperatureConversionTable class

```
import java.io.*;
import java.text.*;
import java.util.*;

public class TemperatureConversionTable {
    static double cValues[];
    static NumberFormat nf;

    public static void main(String[] args) {
        System.out.println("Enter Celsius values, one per line, then press Enter when done:");
Scanner sc = new Scanner(System.in);
String values = sc.nextLine();
StringTokenizer st = new StringTokenizer(values, " ");
double cValues[] = new double[st.countTokens()];
int index = 0;
while (st.hasMoreTokens()) {
    cValues[index] = Double.valueOf(st.nextToken());
    index++;
}
        process(System.in);
        output(System.out);
    }

    static void process(InputStream is) {
        Scanner sc = new Scanner(is);

        nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);

        cValues = new double[0];
        while (true) {
            String value = sc.nextLine();
            if (value.equals("")) { break; }
            double val = Double.valueOf(value);
            String formatVal = nf.format(val);

            boolean found = false;
            for (double d : cValues) {
                if (nf.format(d).equals(formatVal)) {
                    found = true;
                    break;
                }
            }

            if (found) {
                System.err.println(" ** omitting duplicate value:" + formatVal);
            } else {
                cValues = java.util.Arrays.copyOf(cValues, cValues.length+1);
                cValues[cValues.length-1] = val;
            }
        }
    }

    static void output(PrintStream out) {
        java.util.Arrays.sort(cValues);

System.out.println("Celsius\tFahrenheit");
        for (int i = 0; i < cValues.length; i++) {
            double f = cValues[i]*9.0/5 + 32;
System.out.println(nf.format(cValues[i]) + "\t" + nf.format(f));
        }
    }
}
```

Try running this revised **TemperatureConversionTable** on the following problem instance:

INTERACTIVE SESSION: Sample Run

Enter Celsius values, one per line, then press Enter when done:

22.4

13.7

18.003

31

18

** omitting duplicate value:18

Celsius Fahrenheit

13.7 56.66

18 64.41

22.4 72.32

31 87.8

Let's break this code down and try to deal with a number of separately identified code blocks:

OBSERVE: revised main and new process method

```
public static void main(String[] args) {
    System.out.println("Enter Celsius values, one per line, then press Enter when done: ");
};
process(System.in);
output(System.out);
}

static void process(InputStream is) {
    Scanner sc = new Scanner (is);

    nf = NumberFormat.getInstance();
    nf.setMaximumFractionDigits(2);

    cValues = new double[0];
    while (true) {
        String value = sc.nextLine();
        if (value.equals("")) { break; }
        double val = Double.valueOf(value);
        String formatVal = nf.format(val);

        boolean found = false;
        for (double d : cValues) {
            if (nf.format(d).equals(formatVal)) {
                found = true;
                break;
            }
        }

        if (found) {
            System.err.println(" ** omitting duplicate value:" + formatVal);
        } else {
            cValues = java.util.Arrays.copyOf(cValues, cValues.length+1);
            cValues[cValues.length-1] = val;
        }
    }
}
```

The above code will read strings from **System.in** which contain the Celsius values. All Celsius values will be stored in an array of **double** values; however, since the program cannot determine in advance the number of Celsius values entered, it starts—literally—with an empty array of **double**. A **NumberFormat** instance accurate to two digits of precision is created to be used both during **processing** and **output**.

The bulk of the work is handled by the code that reads one string line at a time to extract Celsius values to be added to the array of double values. The program must ensure that no duplicate value appears in the output table. Of course you could filter the output to avoid printing duplicate values, but it's better idea to just avoid storing duplicate values in the first place. Note that you must avoid having two values in the table which would otherwise "round" to the same two digits of precision. So, if the input contained both **15.234** and **15.2321**, only the first value should be entered into the array, because both values round to **15.23**.

As each String **value** is read from the input, the code checks for the empty string as a signal that the user is done; otherwise, it converts the string value into a **double** using **Double.valueOf()** and also constructs a **formatVal** String representing the two-digit rounded value that it would represent in the output table. The **for (double d : cValues)** loop checks to see whether any other Celsius value (once formatted) also equals **formatVal**. If it does, the program considers the new value to be a duplicate and alerts the user that the value will be omitted.

If the value is not eliminated, you must add it to the **cValues** array. Since the size of the array cannot be known in advance, this code uses the **java.util.Arrays.copyOf()** method to extend the array to be one greater in size. The method copies values from the old array into the new one because the new array is allocated as a new Java object. Note that the last value in the array will be the value typed in by the user, **val**.

OBSERVE: output method

```
static void output(PrintStream out) {
    java.util.Arrays.sort(cValues);

    out.println("Celsius\tFahrenheit");
    for (int i = 0; i < cValues.length; i++) {
        double f = cValues[i]*9.0/5 + 32;
        out.println(nf.format(cValues[i]) + "\t" + nf.format(f));
    }
}
```

The final logic in the code **sorts the Celsius values** in **cValues** using the **java.util.Arrays.sort()** method provided by Java. Once **cValues** is sorted, the Fahrenheit temperatures are converted as needed, and the table is output in ascending Celsius order, line by line.

You may be satisfied with this program as it is. It looks like it solves the problem. However, performance may be an issue; the run-time performance on this small data is fine, but it might not work as well on a much larger data set.

With algorithms, the key performance question to consider is what happens when the size of a random problem instance grows; more specifically, when the size doubles. To anticipate the performance of this code, you need to understand how practitioners evaluate the performance of algorithms.

Algorithm Performance

Choosing an algorithm depends on the problem being solved and the problems it will likely face. Algorithms are typically presented with three common cases in mind:

- **Worst case:** The class of problem instances for which an algorithm exhibits its worst runtime behavior. Instead of trying to identify the specific input, algorithm designers typically describe *properties* of the input that prevent an algorithm from running efficiently.
- **Average case:** The expected behavior when executing the algorithm on random problem instances. This measure describes the expectations an average user of the algorithm should have.
- **Best case:** The class of problem instances for which an algorithm exhibits its best runtime behavior. In reality, the best case rarely occurs.

We compare algorithms by evaluating their performance on problem instances of size n . The goal is to determine the number of steps or *operations* the algorithm needs to solve the problem. This is an abstract way of measuring the cost of an algorithm. Intuitively, an operation can be the assignment of a variable, comparing two numbers together, or performing a mathematical operation. This methodology is the standard means for comparing algorithms. By counting the number of operations, we can determine which algorithms scale to solve problems of nontrivial size by evaluating the running time needed by the algorithm in relation to the size of the provided input. This form of evaluation is consistent and does not depend on the programming language used or the specific processor on which the program is run.

When you determine the number of operations performed by an algorithm, you must represent the total count with regards to the original size, n , of the problem instance. For example, the following sample count counts the number of times an integer value appears in an arbitrary array of integer values:

OBSERVE: sample count method

```
static int count (int[] A, int val) {  
    int count = 0;  
    for (int i = 0; i < A.length; i++) {  
        if (A[i] == val) {  
            count++;  
        }  
    }  
    return count;  
}
```

There are six individual statements in the above code. For each statement, you can determine the **maximum** number of times it executes on a problem instance of size n :

Statement	Execution Count
1. int count = 0	executes once
2. int i = 0	executes once
3. if (i < A.length)	executes $n+1$ times
4. if (A[i] == val)	executes n times
5. count++	executes NO MORE THAN n times
6. i++	executes n times

In total, there will be no more than $4*n+3$ statements executed. If n is very large, the constant $+3$ becomes insignificant and you can just say that the number of operations will be four times the total number, n , of values. The phrase used in this course is that the number of operations for the above algorithm *is on the order of* n or $O(n)$. The statement "an order n algorithm"—written as $O(n)$ —means that the total number of operations is bounded by a constant (in this case it was 4) multiplied by n .

In some cases, the number of operations is constant and does not depend on the problem instance size. In these cases, you would represent the behavior as $O(1)$. For a more detailed discussion on the "big O" notation used here, review Chapter 2 in the **Algorithms In A Nutshell** book.

There are a number of classifications in this course. They are ordered here by decreasing efficiency:

- Constant $O(1)$
- Logarithmic $O(\log n)$
- Linear $O(n)$
- Loglinear $O(n \log n)$
- Quadratic $O(n^2)$
- Exponential $O(2^n)$

You will see examples of each of these classifications during this course. For now, let's focus on these four examples:

Constant Performance

Suppose you want to determine the first element of an unordered array of n elements. The effort you'd expend to accomplish this task would be the same even if you had $2*n$, or twice as many, elements. When an algorithm can solve a problem **in a fixed number of operations**, regardless of the size of the problem instance, the algorithm exhibits *Constant Performance*.

Logarithmic Performance

Consider looking for a given last name in the phone book with 1000 pages. You don't typically start on page 1; rather you start on page 500 and determine "which side" of the phonebook you need to search further. You repeat this process until you find the proper page. With each step of work, you reduce the size of the problem by half. When an algorithm can solve a problem in a number of steps relative to the logarithm (base 2) of the problem instance size, we say that the algorithm exhibits *Logarithmic Performance*.

Linear Performance

When the number of steps required by an algorithm to solve a problem grows at the same rate as the problem size grows, then the algorithm exhibits *Linear Performance*. If you're searching for a value in an unordered array of n elements, you would require twice as much work to search through an array with $2*n$ elements.

Quadratic Performance

For some algorithms, doubling the size of the problem instance makes the execution four times longer, resulting in *Quadratic Performance*. Consider the problem of determining whether there are two values in an unordered array of n elements that are the same. For each value in the array, you may have to compare it against each of the other values in order to find a match.

Whenever you identify a nested loop over all elements in a collection, you can be sure that the performance is at least *Quadratic*.

OBSERVE: Sample nested for loop exhibiting quadratic performance

```
for (int i = 0; i < values.length; i++) {  
    for (int j = 0; j < values.length; j++) {  
        Inner Code Block  
    }  
}
```

The *Inner Code Block* executes n^2 times, where n is the number of elements in the **values** array. Here's another common nesting pattern:

OBSERVE: Another sample nested for loop exhibiting quadratic performance

```
for (int i = 0; i < values.length-1; i++) {  
    for (int j = i+1; j < values.length; j++) {  
        Inner Code Block  
    }  
}
```

In that code, the *Inner Code Block* executes once for every *unique pair* of elements in **values**. The total number of times this executes is $n*(n-1)/2$ or $n^2/2 - n/2$. The performance of this nested for loop is still considered to be *Quadratic* with respect to the size of the problem instance, n , despite the subtraction of $n/2$ and the coefficient $1/2$. This is due to the dominance of n^2 in the equation. As n continues to increase, the growth in size of this equation will always be larger than a corresponding growth in a linear equation.

Comparing Classification Families

The run-time behaviors of algorithms can be compared by classification. That is, an $O(1)$ algorithm is considered to be more efficient than an $O(n)$ algorithm. When two algorithms exhibit the same performance classification—say, $O(n \log n)$ —one might still be more efficient than another "because of the constants." Recall how earlier we said that the constants become insignificant with increasing sizes of n ? Theoretically this is true, but one implementation of an algorithm may be more efficient than another even when they belong to the same classification. You may also find that an $O(n^2)$ algorithm is more efficient than a comparable $O(n \log n)$ algorithm **for small values of N** . The associated constants for the $O(n \log n)$ algorithm make the code run slower than it does with the $O(n^2)$ algorithm for small values of n . Once n increases, the $O(n \log n)$ algorithm will outperform any $O(n^2)$ algorithm regardless of constants.

Let's go back and evaluate the performance of **TemperatureConversionTable**.

🔗 In your **DataStruct** project, Create a **/performance** source folder to store all performance-related classes.

🌀 Create a **TimeTemperatureConversion** class in the default package of the **/performance** source folder.

CODE TO TYPE: TimeTemperatureConversion class

```
import java.util.*;
import java.io.*;


public class TimeTemperatureConversion {
    public static void main(String[] args) throws Exception {
        Scanner sc = new Scanner (System.in);
        System.out.println("Enter number of different values to add.");
        int numItems = Integer.valueOf(sc.nextLine());

        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < numItems; i++) {
            sb.append(i).append("\n");
        }
        sb.append("\n");    // empty string to terminate the input

        // execute with timing in place
        double total = 0;
        int numTrials = 10;
        for (int run = 0; run < numTrials; run++) {
            ByteArrayInputStream is = new ByteArrayInputStream(sb.toString().getBytes());
            System.gc();

            long now = System.currentTimeMillis();
            TemperatureConversionTable.process(is);
            long end = System.currentTimeMillis();
            total += (end - now);
        }


        System.out.println(numItems + ", " + (total/numTrials));
    }
}
```

 Run **TimeTemperatureConversion** now for 64, 128, 256, 512, 1024, 2048, and 4096 and note the results, then we'll make some changes and re-run the code using the same values.

The above code creates a string, **sb**, from which the input is to be read. The **process()** method of **TemperatureConversionTable** was designed to process its input from an **InputStream** object; in the actual code, input came from **System.in**, but in this performance code a **ByteArrayInputStream** object is created and used instead. This code design allows you to write automated test cases so you don't have to input your data manually like we're doing here. Ten trials are executed and the average over all executions is printed.

As the problem size doubles, the time required by **TemperatureConversionTable** to solve each problem increases (tripling or even quadrupling), suggesting that this implementation exhibits *Quadratic Performance*. But does that mean that there is no faster implementation?

To improve our results, we'll use data structures provided by the JDK. (In the next lab, we'll introduce a number of core data structures with behaviors that will be useful as you write more advanced algorithms.)

 Create a **CelsiusValue** class in the default package of the **DataStruct/src** source folder.

CODE TO TYPE: CelsiusValue class

```
import java.text.NumberFormat;

public class CelsiusValue implements Comparable<CelsiusValue> {
    public String formatted;
    public double value;
    public String fahrenheit;

    static NumberFormat nf = null;

    public CelsiusValue(double v) {
        value = v;

        if (nf == null) {
            nf = NumberFormat.getInstance();
            nf.setMaximumFractionDigits(2);
        }

        formatted = nf.format(v);
        fahrenheit = nf.format(9.0*v/5 + 32);
    }

    public boolean equals (Object o) {
        if (o == null) { return false; }
        if (o instanceof CelsiusValue) {
            CelsiusValue other = (CelsiusValue) o;
            return (formatted.equals (other.formatted));
        }

        return (false);
    }

    public int compareTo(CelsiusValue other) {
        return formatted.compareTo(other.formatted);
    }
}
```

This class represents an entry in the Celsius conversion table. The **CelsiusValue** constructor stores the **formatted** Celsius value (accurate to two digits) and its computed **fahrenheit** equivalent. To increase efficiency, there's a static **NumberFormat** field, **nf**, that is constructed the very first time a **CelsiusValue** object is constructed.

You need two other methods to arrive at this solution. You may be familiar with the Java standard **equals(Object)** method, which determines whether two objects are equal to each other. For this problem, two **CelsiusValue** objects are equal if they have the same **formatted** representation; this will prevent two duplicate entries from appearing in the table when their formatted values are the same. The **compareTo** method determines the ordering of two **CelsiusValue** objects, to sort entries in the table properly. The names of these methods should be familiar to Java programmers and in the next lesson we will further investigate the naming conventions and standard interfaces and classes provided by the JDK. As you develop classes to represent information in the problem domain, you will see that these classes are no longer exclusively **Input** or **Process** classes.

Now modify **TemperatureConversionTable** as shown:

CODE TO TYPE: Modified TemperatureConversionTable class

```
import java.io.*;
import java.text.*;
import java.util.*;

public class TemperatureConversionTable {
    static TreeSet<CelsiusValue> cValues;
static double cValues[];
static NumberFormat nf;

    public static void main(String[] args) {
        System.out.println("Enter Celsius values, one per line, then press Enter when done:");
        process(System.in);
        output(System.out);
    }

    static void process(InputStream is) {
        Scanner sc = new Scanner (is);

        nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);

        cValues = new TreeSet<CelsiusValue>();new double[0];
        while (true) {
            String value = sc.nextLine();
            if (value.equals ("")) { break; }
            double val = Double.valueOf(value);
            CelsiusValue cval = new CelsiusValue(val);
            String formatVal = nf.format(val);

            boolean found = false;
            for (double d : cValues) {
                if (nf.format(d).equals(formatVal)) {
                    found = true;
                    break;
                }
            }

            if (cValues.contains(cval)found) {
                System.err.println(" ** omitting duplicate value:" + valueformatVal);
            } else {
                cValues.add(cval);
                cValues = java.util.Arrays.copyOf(cValues, cValues.length+1);
                cValues[cValues.length-1] = val;
            }
        }
    }

    static void output(PrintStream out) {
        java.util.Arrays.sort(cValues);
        out.println("Celsius\tFahrenheit");
        for (CelsiusValue cv : cValues) {
            out.println(cv.formatted + "\t" + cv.fahrenheit);
        }
        for (int i = 0; i < cValues.length; i++) {
            double f = cValues[i]*9.0/5 + 32;
            out.println(nf.format(cValues[i]) + "\t" + nf.format(f));
        }
    }
}
```

Now rerun **TimeTemperatureConversion** for the same values we used in the earlier test (64, 128, 256, 512, 1024, 2048, and 4096).

The second column below shows how **TemperatureConversionTable** performed the first time; the third column shows approximate performance after the last changes (your results may differ somewhat):

Problem Size	TemperatureConversionTable Average Execution Time (milliseconds)	TemperatureConversionTable with ArrayList Average Execution Time (milliseconds)
64	3.0	2.6
128	8.0	3.0
256	22.8	6.0
512	79.7	7.0
1024	293.6	9.0
2048	949.8	13.7
4096	3814.8	24.2
8192	*	47.1
16384	*	97.4

You can see that, as the problem instance size increases, this revised implementation is ten times faster (size 512) and even 100 times faster (size 4096). Clearly the second implementation is much more efficient! In this case, it appears that the choice of data structure vastly improved the efficiency of the code, reaffirming the observation by Wirth that *Algorithms + Data Structures = Programs*.

Lessons Learned

Real-world problems are not always as clean and simple as those presented here. In particular, you must routinely maintain a highly dynamic collection of values. Sometimes you might want to add or remove a value to or from the collection. You might want to store the entire collection to persistent storage (such as a database or the file system) so you can retrieve it entirely at a later point. Since release 1.5 of the JDK, Java provides the *Collections Framework*, which is a sophisticated set of classes to represent and manipulate collections. You have likely come across these classes because of their versatility (classes such as **ArrayList** and **HashMap**, for example). As a programmer, you need to know about these classes and—more importantly—to know the specific circumstances under which to use each one. There is no need for you to reimplement these data structures on your own, but you do need to understand how to select the appropriate classes for your needs.

We started this course by completing a problem using the most primitive capabilities offered by the Java programming language. The **TimeTemperatureConversion** class works correctly, but it's not efficient enough when tackling larger problem instances. You can find ways to modify your programs to improve their efficiency, but in most cases it's easier to use the available data structures rather than implement your own versions.

At the end of each lab, we'll review key concepts regarding data structures and algorithms. Here is the first list of key concepts:

- **Use arrays in the way they were designed:** Use arrays when you have a fixed and bounded number of values and you need immediate access to any of these values using a position index.
- **Avoid searching through unordered arrays:** It's inefficient. If searching for an item is a key part of your algorithm, do not store your items in an unordered array.
- **Avoid dynamically resizing arrays to be just one size larger:** If you are frequently adding an item to a collection, review the Java Collections Framework (described in next lesson) to find a more suitable data structure for dynamic behavior.

ArrayList Amortized Reallocation

If you have access to the source code of the JDK, review the **ensureCapacity()** method of the **java.util.ArrayList** class. You can see that whenever a new element is added to an **ArrayList** object, **ensureCapacity()** first ensures that it has enough capacity for the new element. The size of the new array is roughly 1.5 times larger whenever it needs to expand the size of the underlying **elementData** array.

OBSERVE: ArrayList add() method uses Amortization through ensureCapacity()

```
public boolean add(E e) {
    ensureCapacity(size + 1); // Increments modCount!!
    elementData[size++] = e;
    return true;
}

public void ensureCapacity(int minCapacity) {
    modCount++;
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity) {
        Object oldData[] = elementData;
        int newCapacity = (oldCapacity * 3) / 2 + 1;
        if (newCapacity < minCapacity)
            newCapacity = minCapacity;

        // minCapacity is usually close to size, so this is a win:
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}
```

When you expand an array to accommodate more elements, make sure that you don't call the operation too frequently. Instead, increase the size of the array by multiplying it by some number rather than by adding room for a constant number of items.

The next lesson will introduce the Java Collections Framework, which we'll use extensively in this course.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Data Structures and the Java Collections Framework

Lesson Objectives

When you finish this lesson you will be able to:

- identify the core interfaces provided by the Java Collections Framework.
- explain the difference between a Set and a List.
- implement equals(Object) and hashCode() methods as required by Set and List implementations.

Introduction to Java Collections Framework

In this course, we use the commonly-accepted term "Java Development Kit (JDK)" to refer to the platform for developing Java applications. The JDK defines a range of Application Programming Interfaces (APIs) for general purpose functionality, including network programming packages ([java.net](#)) and graphical user interfaces ([java.awt](#), [javax.swing](#)). Altogether, the JDK release dated August 1, 2013 contains over 12,000 classes (18,000 classes if you include anonymous and inner classes). In this lesson, we are concerned with the **java.util** package, which contains the Collections Framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

Before describing the classes in the Collections Framework, we need to discuss the nature of an *interface* in Java. Each Java class defines public methods to be used by external classes. The **java.lang.String** class, for example, contains 82 public methods. Did you know that you can determine the last index position of a character within a String? You might if you had read the documentation and discovered that there is a **lastIndexOf(char ch)** method. This class also has a **compareTo(String s)** method that compares two strings in alphabetical order; it returns 0 if the two string objects are equal to each other and returns a negative or positive number to determine the alphabetic order of the two String objects. Because comparing two objects is a fundamental operation for so many classes, Java designers developed an *interface* that declares this behavior (here's part of the documentation in the interface):

OBSERVE: java.lang.Comparable Interface

```
package java.lang;

public interface Comparable<T> {
    /**
     * Compares this object with the specified object for order. Returns a
     * negative integer, zero, or a positive integer as this object is less
     * than, equal to, or greater than the specified object.
     */
    public int compareTo(T o);
}
```

An interface contains a set of methods that represents a behavior; in this case, the **Comparable** interface represents the ability to order two objects. This behavior is the same whether the object is a String, an Integer or a Double object. The interface specifies the behavior and a class provides the behavior by declaring that the class *implements the interface*. The String class, for example, declares this:

OBSERVE: String interfaces

```
public final class String implements java.io.Serializable, Comparable<String>, CharSequence { ... }
```

Once you know that the String class implements **Comparable**, you know that it must provide an implementation of its methods, and specifically that it will have the **compareTo** method defined by the interface.

To describe the Collections Framework, let's start with the **java.util.Collection<E>** interface, which is the fundamental interface in this package. A **Collection<E>** represents a group of elements of type **E** defined using the Java Generics concept. This general definition applies to a wide range of behaviors. This interface defines 15 methods, including these six fundamental methods:


- **int size()** returns size of the collection.
- **boolean isEmpty()** determines if the collection is empty.

- **contains (Object element)** determines whether it contains a given element.
- **Iterator<E> iterator()** enables retrieval of elements in the collection *in some order*.
- **boolean add(E element)** adds an element to the collection (optional method).
- **remove(Object element)** removes an element from the collection (optional method).

Because some collection objects are *immutable*—that is, their elements cannot be changed—the last two methods above are optional and a Collection object may or may not honor requests to add or remove an element. However, if an add method returns **true**, the Collection guarantees that the element was added to the set. If the add method returns **false**, the collection already contains the element and will not allow duplicates. If a Collection refuses to add a particular element for a reason other than it being a duplicate, it *must* throw an Exception.

The following code sample shows how to use all of these methods with an **ArrayList** object to determine the unique letters used in a given sentence.

Create a **Collections** project and add it to the **Java6_Lessons** working set.

 In the default package of the **Collections/src** source folder, create a **UniqueLetters** class as shown:

CODE TO TYPE: UniqueLetters class

```
import java.util.*;

public class UniqueLetters {
    static final String vowels = "aeiou";

    public static void main(String[] args) {
        System.out.println("Enter at least one character:");
        Scanner sc = new Scanner (System.in);
        String s = sc.nextLine();
        Collection<Character> unique = new ArrayList<Character>();

        for (int i = 0; i < s.length(); i++) {
            char c = Character.toLowerCase(s.charAt(i));
            if (Character.isWhitespace(c)) { continue; }

            if (!unique.contains(c)) {
                unique.add(c);
            }
        }

        if (unique.isEmpty()) {
            System.out.println("Please enter at least one character");
            System.exit(1);
        }

        System.out.println("There are " + unique.size() + " unique characters");
        System.out.print("  Vowels:");
        for (int i = 0; i < vowels.length(); i++) {
            char c = vowels.charAt(i);
            if (unique.contains(c)) {
                System.out.print(c);
                unique.remove(c);
            }
        }

        System.out.print("\n  Consonants:");
        for (Iterator<Character> it = unique.iterator(); it.hasNext(); ) {
            System.out.print(it.next());
        }
        System.out.println();
    }
}
```

Run this program on the sample input below:

INTERACTIVE SESSION: Sample execution of UniqueLetters

Enter at least one character:

Now is the time for all good men to come to the aid of their country

There are 18 unique characters

Vowels:aeiou

Consonants:nwsthmfrlgdcy

Let's take a closer look at this code.:

OBSERVE: Adding elements to a Collection

```
Collection<Character> unique = new ArrayList<Character>();

for (int i = 0; i < s.length(); i++) {
    char c = Character.toLowerCase(s.charAt(i));
    if (Character.isWhitespace(c)) { continue; }

    if (!unique.contains(c)) {
        unique.add(c);
    }
}
```

The **unique** object is constructed as an **ArrayList** of **Character** objects. For each **non-whitespace character** in the **input string (converted to lowercase)**, we check to make sure that the **character is not already a member of unique** before **adding the character to it**. Just keep in mind that once the **add** method completes, the element is guaranteed to be a member of the collection. The specific implementation of Collection—in this case **ArrayList**—is responsible for handling the **add** request; **ArrayList** appends the character to the end of the growing **list of Character objects**.

OBSERVE: Searching for and removing elements from a Collection

```
static final String vowels = "aeiou";
...
for (int i = 0; i < vowels.length(); i++) {
    char c = vowels.charAt(i);
    if (unique.contains(c)) {
        System.out.print(c);
        unique.remove(c);
    }
}
```

The code iterates over the **known vowels** and removes these characters from the collection **unique**. Finally, all Collection classes provide a consistent means to iterate over their elements, as shown in this code:

OBSERVE: Iterate over elements in a Collection using an Iterator

```
for (Iterator<Character> it = unique.iterator(); it.hasNext(); ) {
    System.out.print(it.next());
}
```

The Collection classes also implement the **Iterable** interface, which means you can use Java's *enhanced for loop* to iterate over the elements. The above iteration code could have been written more simply as:

OBSERVE: Iterate over elements in a Collection using enhanced for loop

```
for (Character c : unique) {
    System.out.print(c);
}
```

The Collection interface forms the root of the hierarchy for three important interfaces in the Collections Framework: List, **Set**, and Queue. This code example uses the **ArrayList** class, which is a concrete implementation of the List class. We'll discuss each interface in the upcoming sections of the lesson.

Each implementation of `Collection` guarantees certain performance of the methods defined earlier. In describing this performance, the JDK documentation uses the same terms used in the previous lesson to describe the worst-case run-time performance of algorithms. Here is the "specification sheet" for **ArrayList** for the six fundamental `Collection` methods:

- **O(1)**: `size`, `isEmpty`
- **Amortized Constant Time**: `add`
- **O(n)**: `remove`, `contains`, `iterate`

The **add** method is declared to have *amortized constant time* because **ArrayList** must reallocate more memory when it becomes full. Recall the discussion in the previous lesson about the dangers of resizing arrays? When **ArrayList** needs to resize its array of n elements, it makes sure to request $(3*n/2+1)$ elements to reduce the number of times that it has to reallocate memory. In the long run, adding n elements requires only a total of $O(n)$ time, which means each individual **add** operation is considered to be *amortized constant time*.

There are two reasons to choose a particular data structure: (1) because of the functional behavior that it provides; (2) because of the performance associated with that behavior. All operations for these classes will be described using the algorithm performance classification.

Set Interface

We start with the **Set** interface because it adds no methods to the `Collection` interface; it only constrains methods that add an element to a **Set**. Specifically, a **Set** is a `Collection` that contains no duplicate elements and never contains **null** as an element. The interface matches the mathematical concept of a set. This constraint changes the behavior of **add(E element)** to return **false** if the provided element already exists in the **Set**.

The `Collections` Framework is designed to conform to basic Java principles, so there are some subtle changes to both the standard **equals** and **hashCode** methods. The **equals** method is the standard means in Java to determine whether two objects are equal. When dealing with two `Collection` objects, you'll often want to determine if they represent the same collection of objects *regardless of the actual concrete class implementing this interface*. However, it would be cumbersome to require this capability of all `Collection` subclasses. Java's designers have solved that problem by requiring that **Set** implementations only support **equals** with other **Set** implementations. The main reason for this is that **Set** objects cannot be ordered, so the **equals** method for all implementations of **Set** must return **false** whenever the provided object is itself not a class that implements **Set**. In other words, **Set** objects can only be compared for equality against other **Set** objects.

Finally, when **set1.equals(set2)** is true, it must be the case that **set1.hashCode()** equals **set2.hashCode()**. This is the essential relationship between the **hashCode** and **equals** methods as required by the Java specification. However, classes that implement **Set** may choose how to implement these methods. To ensure the proper relationship between **equals** and **hashCode**, regardless of the specific implementation, any implementation of **Set** must make sure that its **hashCode** method returns the sum of the **hashCode** of all of its members. Because addition is a commuting operation (that is, $a+b$ equals $b+a$), this ensures that the respective **hashCode** values of two **Set** objects will always be the same if they represent the same elements. This design principle is incredibly insightful because it separates the contract of the interface from any of its potential implementations.

Let's reimplement the sample problem described earlier, which actually seems more easily implemented using sets. For this implementation, replace **ArrayList** with **HashSet**:

CODE TO TYPE: Modify UniqueLetters class

```
import java.util.*;

public class UniqueLetters {
    static final String vowels = "aeiou";

    public static void main(String[] args) {
        System.out.println("Enter at least one character:");
        Scanner sc = new Scanner (System.in);
        String s = sc.nextLine();
        Collection<Character> unique = new HashSetArrayList<Character>();

        for (int i = 0; i < s.length(); i++) {
            char c = Character.toLowerCase(s.charAt(i));
            if (Character.isWhitespace(c)) { continue; }

if (!unique.contains(c)) {
            unique.add(c);
}
        }

        if (unique.isEmpty()) {
            System.out.println("Please enter at least one character");
            System.exit(1);
        }

        System.out.println("There are " + unique.size() + " unique characters");
        System.out.print("  Vowels:");
        for (int i = 0; i < vowels.length(); i++) {
            char c = vowels.charAt(i);
            if (unique.contains(c)) {
                System.out.print(c);
                unique.remove(c);
            }
        }

        System.out.print("\n  Consonants:");
        for (Iterator<Character> it = unique.iterator(); it.hasNext(); ) {
            System.out.print(it.next());
        }
        System.out.println();
    }
}
```

Now execute the modified **UniqueLetters** class using the same input as before to generate the same output:

INTERACTIVE SESSIONS: Output of UniqueLetter remains the same

```
Enter at least one character:
Now is the time for all good men to come to the aid of their country
There are 18 unique characters
Vowels:aeiou
Consonants:fgdcnlmhwtsry
```

A **HashSet** offers improved performance for the core operations described earlier by using a scheme that subdivides a collection into *b* buckets. Elements are placed into a bucket based on the hashing function, **hashCode()**; these performance characteristics are valid if the hashing function disperses the elements properly among the buckets.

- **O(1)**: size, isEmpty, contains, add, remove
- **O(n)**: iterate over the set of elements

When you use a **HashSet**, you have the option of predeclaring an *initial Capacity* which specifies the number of buckets to use to store the collection. If you set this too high, the iteration over all elements in the set requires time proportional to $n+b$ (where b is the number of buckets in the **HashSet**); it's better to let **HashSet** manage its own structure.


List Interface

The List interface provides a sequence-oriented perspective on a collection. First, it is ordered, which provides the first distinction with regard to **Set**. Second, a List may contain duplicate elements or even null values. Third, this interface is much more powerful than the "list" concepts that most programmers intuitively have in mind—specifically, the List interface offers additional behaviors to Collection:

- **Index (positional) access.** You can retrieve, remove or replace any element by its position in the List. You can also insert elements into a List at a specific location, bumping all elements up by one spot from that point in the list.
- **Search.** You can identify the ordinal location ($0 \dots n-1$) in the list of a given element (from either the front or the end of the list).

As with **Set**, any class implementing List must properly implement the **equals()** method to return **false** whenever it is compared against a non-List object. Two List objects are considered to be equal if they are of the same size and they contain the same elements in the same order. However, the **hashCode** method must be defined to work with **equals**. For this reason, any class that implements List must be sure that its **hashCode** method follows the contract as defined by the List class.

You've already seen how useful **ArrayList** can be. Another useful List implementation is **LinkedList**, which implements a doubly-linked list of items. Why would you choose to use one class over the other? Suppose you wanted to implement a *double-ended queue* (or "dequeue" for short; pronounced "deek"). The following benchmark directly compares **ArrayList** to **LinkedList**. First, it creates a list by adding integer elements at the "end" of the list and then randomly removing either the first or the last element in the list. Once *max* iterations have completed, it drains the remaining elements of the list by repeatedly removing the first one.

 Create a **CompareDequeue** class in the default package of the **/src** source folder:

CODE TO TYPE: CompareDeque class


```
import java.util.*;

public class CompareDeque {
    static long[] performance(int max, List<Integer> list) {
        long nanoTime = System.nanoTime();
        while (max > 0) {
            list.add(max);
            list.add(max+1);
            if (Math.random() < 0.5) {
                list.remove(0);
            } else {
                list.remove(list.size()-1);
            }
            max--;
        }
        long inner = System.nanoTime();
        while (!list.isEmpty()) {
            list.remove(0);
        }
        long lastTime = System.nanoTime();
        return new long[]{ inner-nanoTime, lastTime-inner };
    }

    public static void main(String[] args) {
        int max = 65536;
        float m = 1000000;

        System.out.println("\t\tConstruct\t\tDrain\t\tTotal");
        System.gc();
        long[] arraylist = performance(max, new ArrayList<Integer>());
        System.out.println("ArrayList\t" + arraylist[0]/m + "\t" + arraylist[1]/m +
            "\t" + (arraylist[0] + arraylist[1])/m);

        System.gc();
        long[] linkedlist = performance(max, new LinkedList<Integer>());
        System.out.println("Linkedlist\t" + linkedlist[0]/m + "\t" + linkedlist[1]/m
            +
            "\t" + (linkedlist[0] + linkedlist[1])/m);
    }
}
```

 Save and run this program to produce the performance profile. The code measures the execution time of two phases of the program (growing phase and draining phase). The final number in the column is the total time in milliseconds:

INTERACTIVE SESSION: Output from CompareDeque (Time in milliseconds)

	Construct	Drain	Total
ArrayList	253.11002	469.56882	722.67883
LinkedList	17.202131	1.695052	18.897184

In this benchmark, **LinkedList** performs much faster than **ArrayList**. **ArrayList** suffers in comparison because it must constantly resize its internal array to meet the growing demand. In addition, when removing the first element from the **ArrayList**, all subsequent items in the List must be copied down.

So, what if you made a small change to the benchmark? Remove a random element instead of just removing an element from either the head or tail of the list. Modify the code as shown:

CODE TO TYPE: CompareDequeue class

```
import java.util.*;

public class CompareDequeue {
    static long[] performance(int max, List<Integer> list) {
        long nanoTime = System.nanoTime();
        while (max > 0) {
            list.add(max);
            list.add(max+1);
            list.remove(max % list.size());
            if (Math.random() < 0.5) {
                list.remove(0);
            } else {
                list.remove(list.size() - 1);
            }
            max--;
        }
        long inner = System.nanoTime();
        while (!list.isEmpty()) {
            list.remove(0);
        }
        long lastTime = System.nanoTime();
        return new long[]{ inner-nanoTime, lastTime-inner};
    }

    public static void main(String[] args) {
        int max = 65536;
        float m = 1000000;

        System.out.println("\t\tConstruct\tDrain\t\tTotal");
        System.gc();
        long[] arraylist = performance(max, new ArrayList<Integer>());
        System.out.println("ArrayList\t" + arraylist[0]/m + "\t" + arraylist[1]/m +
            "\t" + (arraylist[0] + arraylist[1])/m);

        System.gc();
        long[] linkedlist = performance(max, new LinkedList<Integer>());
        System.out.println("Linkedlist\t" + linkedlist[0]/m + "\t" + linkedlist[1]/m
+
            "\t" + (linkedlist[0] + linkedlist[1])/m);
    }
}
```

 Save and run it; the tables have turned!

INTERACTIVE SESSION: Output from CompareDequeue (Time in milliseconds)

	Construct	Drain	Total
ArrayList	320.66714	466.6363	787.3034
Linkedlist	1834.3232	1.668898	1835.9921

When choosing the appropriate data structure, you need to understand exactly how the data structure is to be used, especially if it is updated frequently, you need randomized access to any element, or the ratio of add/remove compared to the number of queries is used to determine whether an element exists in the collection.

Queue Interface

A *Queue* is a data structure that allows you to remove an element only at the head of an ordered sequence and insert elements only at the end of the sequence. The actual implementation determines whether the Queue is first-in, first-out (what would be expected) or a variation (such as a last-in, first-out). Further, one can only remove an element from the head of the queue as determined by the implementation. There is a Deque interface that extends Queue to offer double-ended queueing behavior.

In the Collections Framework, a Queue specifies a collection designed to hold elements prior to processing. The Queue interface extends Collection. This interface is designed to support collections that have a maximum size (such as bounded queues) in addition to more general queues with no restrictions. You can *offer* an element to the Queue, which simply returns **false** if the Queue denies this request (usually because it is a bounded queue). The remaining four methods (**remove**, **poll**, **element**, and **peek**) each return (without modifying the queue) or remove the element at the head of the queue.


It may seem odd that a Queue is not predefined to be a subinterface of List. However, it is critical when designing frameworks to separate structure from behavior. Using list-based semantics is not the only way to implement a queue (you could use circular buffers, for example, which is an efficient way to implement a bounded queue). The **LinkedList** class chooses to implement both List and **Deque** (and thus by extension, Queue). **ArrayList** does not implement Queue though, likely because its performance as a queue would be horrible.

Map Interface

The **Map** interface in the Collections Framework allows you to create collections that associate a value with a unique key. Given a Map, you can add (or replace) a key mapping with **put(key,value)**; to retrieve a value or to determine whether the value exists in the Map, use **get(key)**.

The designers of Map had to decide whether a Map was a Collection object; after all, it stores a collection of values. However, the fundamental operation on a Collection is the **add** method, and there is no easy way to apply this operation to a Map. Instead, Map objects offer two Collection views over its objects. Because the keys in a Map are unique, you can retrieve the **Set** of all keys in the Map, but their values might not be unique, so Map only allows you to retrieve the Collection of values in the map.

Map is designed to optimize the insertion and removal of (*key, value*) pairs. In doing so, any ordering properties among the keys are lost. For example, try to order the keys in a Map alphabetically.

 Create a **SortingMap** class in the default package of the **/src** source folder:

CODE TO TYPE: Sorting key values in map

```
import java.util.*;

public class SortingMap {
    public static void main(String[] args) {


        float m = 1000000;
        Map<String,Integer> map = new HashMap<String,Integer>();

        long start = System.nanoTime();
        byte[] word = new byte[3];
        for (byte c0 = 'A'; c0 <= 'Z'; c0++) {
            word[0] = c0;
            for (byte c1 = 'A'; c1 <= 'Z'; c1++) {
                word[1] = c1;
                for (byte c2 = 'A'; c2 <= 'Z'; c2++) {
                    word[2] = c2;
                    String s = new String(word);
                    map.put(s, c0*c1*c2);
                }
            }
        }

        long created = System.nanoTime();
        ArrayList<String> keys = new ArrayList<String>(map.keySet());
        Collections.sort(keys);
        long sorted = System.nanoTime();

        long total = 0;
        for (String k : keys) {
            total += map.get(k);
        }
        long done = System.nanoTime();
        System.out.println("Total\t" + total);
        System.out.println("Created\t" + (created-start)/m +
            "\nSorted\t" + (sorted-created)/m +
            "\nDone\t" + (done - sorted)/m);

        int print = 10;
        for (String s : map.keySet()) {
            if (--print < 0) { break; }
            System.out.println(s);
        }
    }
}
```

 Save and run it.

INTERACTIVE SESSION: Output of SortedMap

```
Total    8181353375
Created  99.77721
Sorted   68.29778
Done     13.256433
GDC
GDD
GDA
GDB
GDK
GDL
GDI
GDJ
GDG
GDH
```

The code inserts pairs (s, n) for *max* entries and then outputs the keys in sorted value. To do that, it must retrieve the **keySet()** from the Map, but the set must be converted into a List so it can be sorted, so an **ArrayList** keys is constructed from the set of keys. Finally, it sorts the keys using the **Collections.sort** method. The execution shows the performance, as well as the first ten keys in the Map; note that these keys aren't sorted because the Map iteration does not maintain ordering.

Summarizing the Implementations You Need To Know

We refer to a number of common implementations provided for you in the JDK. These are the "go to" classes you'll use again and again to solve your programming issues. You must never re-implement these data structures on your own; these implementations have already been fine-tuned by experts.

Interface	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked List Implementations
Set	HashSet		TreeSet	
List		ArrayList		LinkedList
Map	HashMap		TreeMap	

Each of these default implementations provide distinct performance profiles for the Collection methods. In addition, both List and Map add methods to their interface definitions. It's important to differentiate between the behaviors of the classes that implement these interfaces as well.

Important Methods For Keys And Values

All of the classes in the Collection Framework support Java Generics, so you don't just refer to an **ArrayList**, but an **ArrayList<String>**. Doing so makes your code more robust because it enables the compiler to detect many class cast exceptions. The **contains** method—common to many Collections classes—must be implemented properly, otherwise the Collection class won't work. Consider this example:

 Create a class **Tuple** in the default package of the **/src** source folder:

CODE TO TYPE: Tuple class

```
public class Tuple {
    String value;
    int    x;
    int    y;

    public Tuple (String v, int x, int y) {
        this.value = v;
        this.x = x;
        this.y = y;
    }

    public String toString () {
        return "(" + value + "," + x + "," + y + ")";
    }
}
```

 Now, create the following driver class named **TupleDriver** in the default package of the **/src** source folder:


CODE TO TYPE: TupleDriver class

```
import java.util.*;

public class TupleDriver {
    public static void main(String[] args) {
        ArrayList<Tuple> al = new ArrayList<Tuple>();

        Tuple t1 = new Tuple("Sample", 10, 20);
        al.add(t1);
        System.out.println("ArrayList Contains tuple:" + al.contains(t1));

        Tuple t2 = new Tuple("Sample", 10, 20);
        System.out.println("ArrayList Contains tuple:" + al.contains(t2));
    }
}
```

 Save and run the above code to see some surprising output:

INTERACTIVE SESSION: Output of TupleDriver

```
ArrayList Contains tuple:true
ArrayList Contains tuple:false
```

Although **Tuple** *t1* and *t2* are exactly the same, only one of them is found in the **ArrayList**. You see this behavior because you haven't implemented the requisite **equals(Object o)** method required by the Collections Framework. The **contains(o)** method will return **true** if there is an object in the Collection for which **equals(o)** is **true**. Go ahead and add the **equals** method now:

CODE TO TYPE: Modified Tuple class

```
public class Tuple {
    String value;
    int    x;
    int    y;

    public Tuple (String v, int x, int y) {
        this.value = v;
        this.x = x;
        this.y = y;
    }

    public boolean equals (Object o) {
        if (o == null) { return false; }
        if (!(o instanceof Tuple)) { return false; }
        Tuple other = (Tuple) o;

        if (value == null) {
            if (other.value != null) {
                return false;
            }
        } else if (!value.equals(other.value)) {
            return false;
        }

        return x == other.x && y == other.y;
    }

    public String toString () {
        return "(" + value + "," + x + "," + y + ")";
    }
}
```

Review the **equals** method. It should not throw any Exception, but rather handle all cases (such as **null** object references). This code is able to work with **Tuple** objects that have a *value* String attribute of **null**. Now go back and rerun the **TupleDriver**:

INTERACTIVE SESSION: Output of TupleDriver

```
ArrayList Contains tuple:true
ArrayList Contains tuple:true
```

Now let's try to use this **Tuple** as a key in **HashSet**. Modify **TupleDriver** as shown:

CODE TO TYPE: Modified TupleDriver class

```
import java.util.*;

public class TupleDriver {
    public static void main(String[] args) {
        ArrayList<Tuple> al = new ArrayList<Tuple>();

        Tuple t1 = new Tuple("Sample", 10, 20);
        al.add(t1);
        System.out.println("ArrayList Contains tuple:" + al.contains(t1));

        Tuple t2 = new Tuple("Sample", 10, 20);
        System.out.println("ArrayList Contains tuple:" + al.contains(t2));

        HashSet<Tuple> values = new HashSet<Tuple>();
        values.add(t1);
        System.out.println("HashSet Contains tuple:" + values.contains(t1));
        System.out.println("HashSet Contains tuple:" + values.contains(t2));
    }
}
```



save and run it again:

INTERACTIVE SESSION: Output of TupleDriver

```
ArrayList Contains tuple:true
ArrayList Contains tuple:true
HashSet Contains tuple:true
HashSet Contains tuple:false
```

When using a class as a key value in any of the "Hash" collection classes (that is, **HashSet**, **HashMap**, **LinkedHashSet**, or **LinkedHashMap**) you need to implement the **hashCode()** method. Specifically, if two objects are equal, their **hashCode()** values must be identical. If you don't provide your own **hashCode** method, then you will inherit the default from **java.lang.Object** which means **hashCode()** values must be identical. Modify **Tuple** to add a reasonable implementation of **hashCode**.

CODE TO TYPE: Modified Tuple class

```
public class Tuple {
    String value;
    int    x;
    int    y;

    public Tuple (String v, int x, int y) {
        this.value = v;
        this.x = x;
        this.y = y;
    }

    public boolean equals (Object o) {
        if (o == null) { return false; }
        if (!(o instanceof Tuple)) { return false; }
        Tuple other = (Tuple) o;

        if (value == null) {
            if (other.value != null) {
                return false;
            }
        } else if (!value.equals(other.value)) {
            return false;
        }

        return x == other.x && y == other.y;
    }

    public int hashCode() {
        int hash = 0;
        if (value != null) { hash += value.hashCode(); }
        return hash + x + y;
    }

    public String toString () {
        return "(" + value + "," + x + "," + y + ")";
    }
}
```

The **hashCode** method must always return the same value upon each execution. Because all of the fundamental classes in the JDK have suitable **hashCode** methods, you may want to compose your own methods using their values in numerical computations. Rerun **TupleDriver**:

INTERACTIVE SESSION: Output of TupleDriver

```
ArrayList Contains tuple:true
ArrayList Contains tuple:true
HashSet Contains tuple:true
HashSet Contains tuple:true
```

But wait—we're not done. You have to be especially careful with classes that have objects that will be key values. Specifically, you must make these classes immutable, otherwise strange things can happen. For example, modify **TupleDriver** as shown:

CODE TO TYPE: Modified TupleDriver class

```
import java.util.*;

public class TupleDriver {
    public static void main(String[] args) {
        ArrayList<Tuple> al = new ArrayList<Tuple>();

        Tuple t1 = new Tuple("Sample", 10, 20);
        al.add(t1);
        System.out.println("ArrayList Contains tuple:" + al.contains(t1));

        Tuple t2 = new Tuple("Sample", 10, 20);
        System.out.println("ArrayList Contains tuple:" + al.contains(t2));

        HashSet<Tuple> values = new HashSet<Tuple>();
        values.add(t1);
        System.out.println("HashSet Contains tuple:" + values.contains(t1));
        t1.value = "Other";
        System.out.println("HashSet Contains tuple:" + values.contains(t1));
        System.out.println("HashSet Contains tuple:" + values.contains(t2));
    }
}
```



save and run it again:

INTERACTIVE SESSIONS: Output of TupleDriver

```
ArrayList Contains tuple:true
ArrayList Contains tuple:true
HashSet Contains tuple:true
HashSet Contains tuple:false
HashSet Contains tuple:false
```

Once you change the value of an object that has been used as a key value within **HashSet**, you may not be able to find it again. Use the **final** modifier for the attributes of your key classes to avoid these situations:

CODE TO TYPE: Modified Tuple class

```
public class Tuple {
    final String value;
    final int    x;
    final int    y;

    public Tuple (String v, int x, int y) {
        this.value = v;
        this.x = x;
        this.y = y;
    }

    public boolean equals (Object o) {
        if (o == null) { return false; }
        if (!(o instanceof Tuple)) { return false; }
        Tuple other = (Tuple) o;

        if (value == null) {
            if (other.value != null) {
                return false;
            }
        } else if (!value.equals(other.value)) {
            return false;
        }

        return x == other.x && y == other.y;
    }

    public int hashCode() {
        int hash = 0;
        if (value != null) { hash += value.hashCode(); }
        return hash + x + y;
    }

    public String toString () {
        return "(" + value + "," + x + "," + y + ")";
    }
}
```

With this change, the **TupleDriver** class no longer compiles because it's trying to modify the (now immutable) attribute. Delete the offending line in **TupleDriver**. Now you have a working implementation of a class suitable for use as a key value in any of the "Hash" Collections classes.

Lessons Learned

The Collections Framework contains many implementations of the fundamental data structures that you'll use in your algorithms. When you Understand these structures it will help you write efficient code:

- **Rather than reimplement your own, use default implementations of Set, List, and Queue.** The default implementations are made to work efficiently under the most common usages.
- **Seek data structures that lead to $O(n \log n)$ behavior.** Many problems have "naive" solutions that result in $O(n^2)$ run-time performance. In most cases you'll be able to increase performance to $O(n \log n)$ by applying the appropriate data structure. You'll see this happen often in future lessons.

The Collections Framework contains a **Collections** class that has a number of static methods useful for your algorithms. Again, these methods are optimized for use by the various classes in the Collections Framework. Use these methods rather than reimplementing them. Each of these methods has a published performance contract to which it adheres, which makes it possible to use them and be assured of reasonably good performance. Review the class in the API documentation when you get a chance. Here are four that you'll use often:

- `sort(List)`: efficiently sort the List in place in $O(n \log n)$ time.
- `binarySearch(List)`: assuming that List is ordered, locate the index of the given key in $O(\log n)$ time.

- `reverse(List, key)`: locate the key in the ordered List object in $O(\log n)$ time.
- `shuffle(List)`: permutes the List in place in $O(n)$ time.

This lesson covered just the highlights of the Collections Framework. The **java.util** package contains 283 classes and 19 interfaces. To delve deeper into Collections, follow the standard [Tutorial](#) on the Collections framework after you complete this course.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Algorithms Using Java

Lesson Objectives

After completing this lesson, you will be able to:

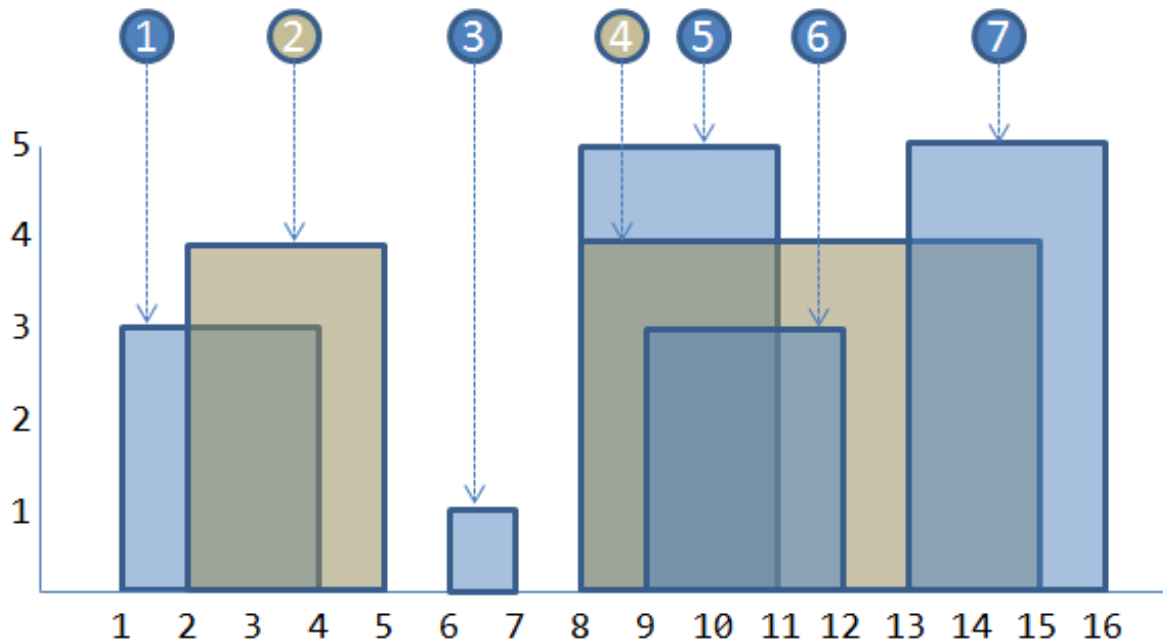
- maintain the ordering property of a List when inserting values.
- use a TreeSet to iterate over the elements of a set in their natural ordering.

Designing Algorithms

There are well-known algorithms you can use to sort an array of strings, such as QuickSort, but the real value of an algorithm is that it is a concise explanation of an efficient way to solve a specific problem. The algorithms you will encounter are as varied as the problems they solve. Let's start by trying to solve a given problem. As you work toward a solution, you'll solve numerous sub-tasks and make important decisions and develop are essential in the process. Let's get started.

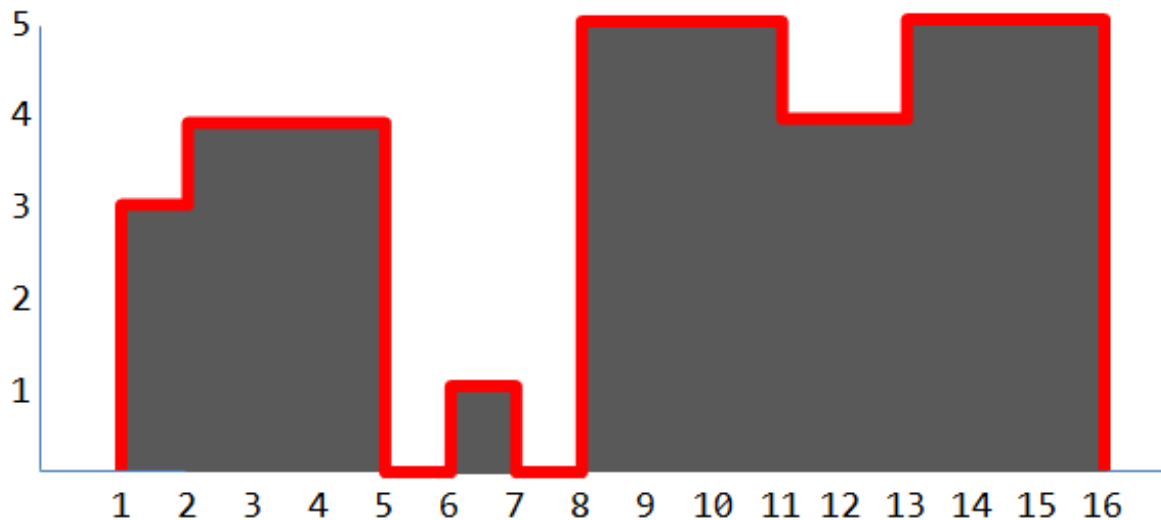
Skyline Problem

Let's say you have a set of rectangular building profiles in two dimensions, compute the *Skyline view* based on the partially overlapping and fully obstructed building profiles at your disposal. Assume there are seven buildings as shown. Note that two buildings (4 and 5) share the same left coordinate, although building 5 is taller:



The Skyline for the solution is shown in **red** with black shadow highlights:

Solution: (1,0) (1,3) (2,3) (2,4) (5,4) (5,0)
(6,0) (6,1) (7,1) (7,0)
(8,0) (8,5) (11,5) (11,4) (13,4)
(13,5) (16,5) (16,0)



If you solve this problem by hand, you'd probably have no trouble tracing the appropriate edges, but you might find it hard to explain the exact sequence of steps you took. No doubt, you would be able to determine the skyline for any conceivable set of rectangular buildings. So, how can you write a program to do the same thing? Let's start by identifying the **Input - Process - Output** phases for this problem.

Each skyline problem instance is defined by a set of buildings. A building can be represented any number of ways; here we'll represent a building with three integers: *left*, *right*, and *height*. Building 1 above, for example, can be defined as (*left*=1, *right*=4, *height*=3). The Input to our final program will be a sequence of text lines, each of which contains three integers separated by spaces. When considering the input, ask yourself these questions: Will the sequence of buildings in the Input already be sorted somehow from left to right? Is it possible for two buildings in the input to share the same *left* coordinate? Can two buildings have the exact same values for all three coordinates? As you consider these questions, make the fewest possible assumptions. This will help make your code more robust and able to handle diverse situations.

For this problem, assume that each building coordinate is represented by an integer value greater than 0. This prevents bizarre situations (a building with zero or negative height). Also, assume *left* < *right* for each building. This ensures you will know that the smaller coordinate is truly "left" of a building's right coordinate. There may be duplicate coordinates (even buildings) in the input. It would be more difficult to try to find and remove duplicate buildings from the input set than to write your program to work even when the input contains duplicate buildings.

To record the input, you will need data structures to store the information about each building. First you need to define a way to store information about a building.

Create a project named **Algorithms** and assign it to your **Java6_Lessons** working set.

📁 In the **Algorithms** project **/src** source folder, create a package named **skyline** to contain all the code you need to solve this problem.

🌱 Create a **Building** class in the **/src** source folder **skyline** package to represent a building:

CODE TO TYPE: Building class

```
package skyline;


public class Building {
    final public int left;
    final public int right;
    final public int height;

    public Building(int left, int right, int height) {
        this.left = left;
        this.right = right;
        this.height = height;

        if (left <= 0 || right <= 0 || height <= 0 || left >= right) {
            throw new IllegalArgumentException ("Invalid building parameters: " +
                left + "," + right + "," + height);
        }
    }

    public String toString () {
        return "[" + left + "," + right + "]" @ " + height;
    }
}
```

This class represents a valid building; any attempt to construct an invalid Building object will throw an Exception. The input for a Skyline problem instance consists of a set of Building objects which you can store as an **ArrayList<Building>** object.

 Create a **Skyline** class in the **skyline** package of the **/src** source folder and enter the code as shown (the code processes a set of text lines representing buildings and the outputs the buildings it found):

CODE TO TYPE: Skyline class

```
package skyline;

import java.io.*;
import java.util.*;

public class Skyline {

    public static Collection<Building> retrieveInput(InputStream is) {
        ArrayList<Building> buildings = new ArrayList<Building>();
        Scanner sc = new Scanner (is);
        while (sc.hasNextLine()) {
            String s = sc.nextLine();
            if (s.equals("")) { break; }

            try {
                StringTokenizer st = new StringTokenizer(s);
                int left = Integer.valueOf(st.nextToken());
                int right = Integer.valueOf(st.nextToken());
                int height = Integer.valueOf(st.nextToken());

                Building b = new Building (left, right, height);
                buildings.add(b);
            } catch (NumberFormatException nfe) {
                System.err.println(" ** Ignoring " + s + ": all values must be integers.
");
            } catch (Exception e) {
                System.err.println(" ** Ignoring " + s + ": " + e.getMessage());
            }
        }

        return (buildings);
    }

    public static void main(String[] args) {
        Collection<Building> buildings = retrieveInput(System.in);

        for (Building b : buildings) {
            System.out.println "[" + b.left + ", " + b.right + "] @ " + b.height);
        }
    }
}
```

This "scaffolding" code contains a **retrieveInput** method that reads a series of lines that represent the buildings in the Skyline problem instance. **retrieveInput** returns an **ArrayList** of **Building** objects that it parsed. When in doubt as to which class to use when representing a list, start with **ArrayList**. The **main** method outputs the building information for all buildings retrieved from the input.

Run **Skyline** with the input set below (press **Enter** twice when you finish); the output shows that all buildings were properly processed by this scaffolding class:

INTERACTIVE SESSION: Demonstrate Skyline processes sample input

```

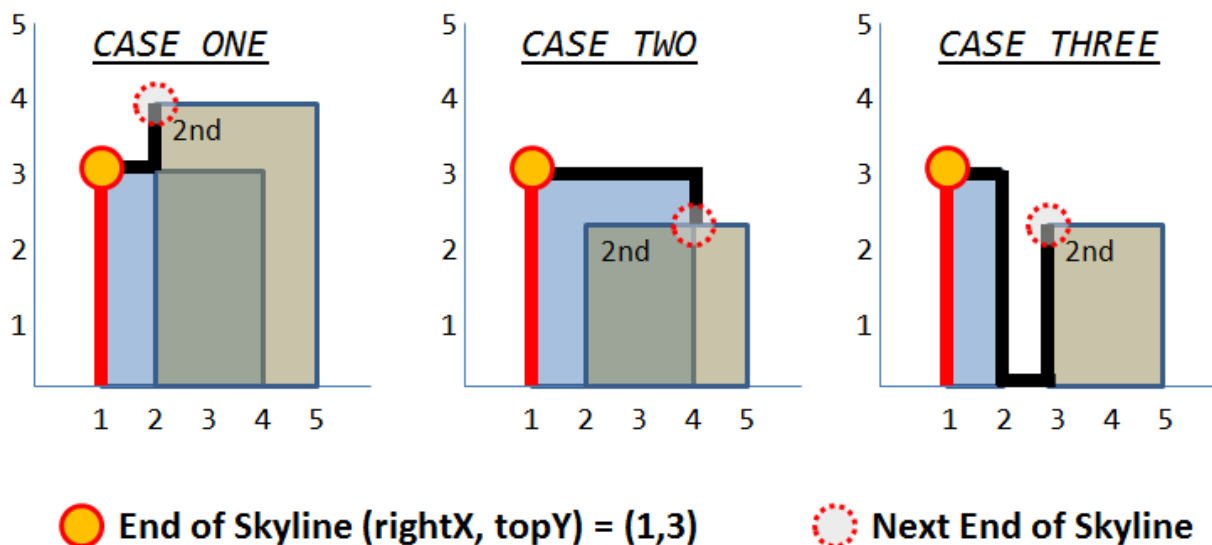
1 4 3
6 7 1
8 15 4
8 11 5
9 12 3
2 5 4
13 16 5

[1,4] @ 3
[6,7] @ 1
[8,15] @ 4
[8,11] @ 5
[9,12] @ 3
[2,5] @ 4
[13,16] @ 5

```

Now you are ready to consider how to represent a valid solution to the Skyline problem—how the Output should be represented. In the above graphic, the solution is represented as a sequence of "(x,y)" points that determines the Skyline from left to right. The bottom of each building is $y=0$ and the first point in the solution is $(L, 0)$ where L is the *left* coordinate of the leftmost building. Finally, the last point in the solution is $(R, 0)$ where R is the *right* coordinate of the rightmost building. Because the buildings are rectangular, you know that the Skyline is composed of a sequence of alternating vertical and horizontal edges. With this information in hand, you can see that your program should compute the set of edges from the building information in the Input. From the sequence of edges, you can easily compute the sequence of points in the Skyline.

When tackling a complex problem, it really helps to spend the time (like you did) to understand the input and output requirements. Now you're ready to address the Process phase of this algorithm. When trying to determine how to solve the Skyline problem, start with what you know. You know that the leftmost vertical (**red**) edge of the leftmost building will form the start of the Skyline; the first point is **(1,0)** and the "end" of the Skyline is **(1,3)**. Let's use two variables to compute the Skyline; **rightX=1** and **topY=3** are the x- and y-coordinates of the rightmost point in the Skyline. Starting with a first (**blue**) building, consider three different possibilities when processing the "next" second building to determine how to extend the Skyline. The second building is the building in the input set with the left coordinate that is closest to the left coordinate of the first building.



In **Case One**, the second building is taller than the first building, so the Skyline rises (as shown by thick black lines) adding the point **(2,3)** and ending at the "next end" of **(2,4)**. However, in **Case Two**, the second building is smaller than the first building, so the Skyline will come back down (again, shown in thick black lines) adding the point **(4,3)** and ending at the "next end" of **(4,2)**. In addition to maintaining the "end point" of the Skyline (**rightX, topY**) you also need to know **buildingRight**, or the right coordinate of the current building being processed (in **Case Two** you need this value so you know where to "turn").

Now, neither of these cases handles the situation when the second building doesn't actually overlap the first

building (as shown by **Case Three**). We can handle this case though because the Skyline comes back down to "ground zero" and then continues back up the left edge of the second building. Here three points are added to the Skyline—**(2,3)**, **(2,0)**, **(3,0)** and you end up with a "next end" of **(3,2)**.

These cases lay the foundation for an algorithm. Now you need to consider the initialization phase of the algorithm (where do you start?) and the termination phase (how do you end?). To start this algorithm, you need to find the leftmost building with tallest height (just in case two or more buildings start with the same smallest coordinate) and start the Skyline with its vertical edge. To terminate this algorithm, there will be no second building (since they will have all been processed), so you can "close" the algorithm by extending the Skyline to **buildingRight** and then back down to the "ground zero".

Before implementing the algorithm, you should describe its logic using *pseudocode*; this allows you to see the structure without the complicated syntax of a regular programming language. When sketching an algorithm using pseudocode, you can define helper functions as needed. Define a **nextTallest(x)** function that returns the building whose *left* coordinate is closest to the right of x. In the event of a "tie," this function must return the tallest of all such buildings starting at that coordinate. Review the following pseudocode description and note how **Case Three** is handled first, since that detects when the next building doesn't intersect the "current building." With each pass through the loop, **current** and **next** are updated accordingly to represent the index of the current building being processed and the next building to process.

OBSERVE: Pseudocode description of algorithm

```
compute ()
  current = nextTallest(0)
  rightX = current.left
  topY = current.height
  skyline starts with (rightX, 0) and (rightX, topY)

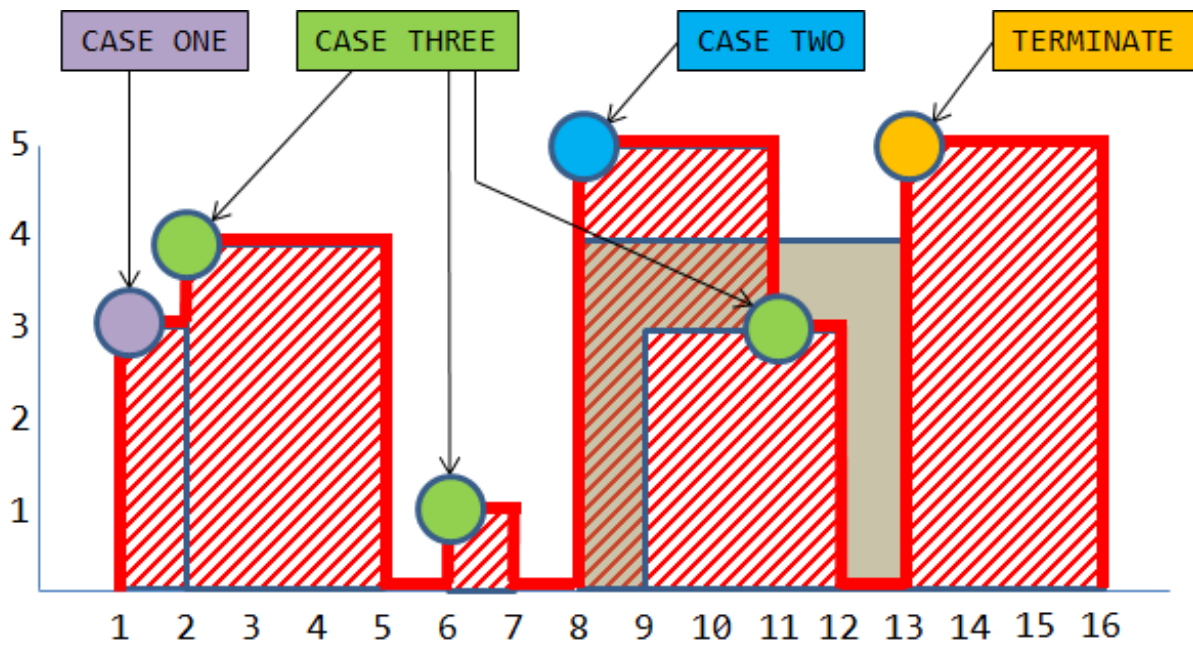
  buildingRight = current.right

  while exists next = nextTallest(rightX) do
    if next.left > buildingRight then
      add edges to skyline according to Case Three
      rightX = next.left
    else if next.height > topY then
      add edges to skyline according to Case One
      rightX = next.left
    else if next.height < topY then
      add edges to skyline according to Case Two
      rightX = current.right

    topY = next.height
    buildingRight = next.right
    current = next

  close skyline by adding (buildingRight, topY) and (buildingRight, 0)
```

Before jumping into implementation, review how this algorithm would work on the sample problem you saw earlier. If you follow the above pseudocode on the sample data, you will see that it **incorrectly computes** this Skyline:



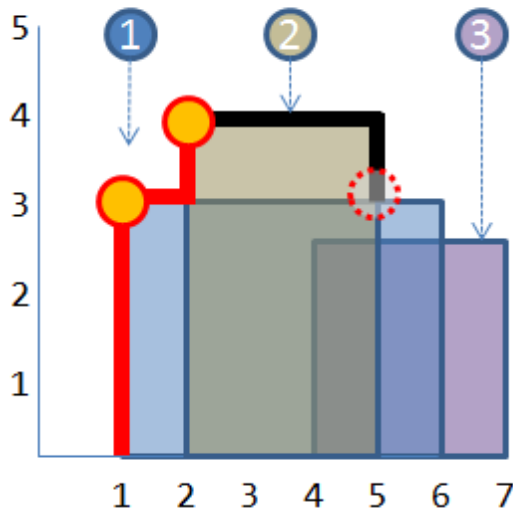
The actual points in the Skyline are:

OBSERVE: Skyline processes sample input	
[(1,0) - (1,3)]	
[(1,3) - (2,3)]	
[(2,3) - (2,4)]	
[(2,4) - (5,4)]	
[(5,4) - (5,0)]	
[(5,0) - (6,0)]	
[(6,0) - (6,1)]	
[(6,1) - (7,1)]	
[(7,1) - (7,0)]	
[(7,0) - (8,0)]	
[(8,0) - (8,5)]	
[(8,5) - (11,5)]	
[(11,5) - (11,3)]	** Here is where the Skyline is incorrect **
[(11,3) - (12,3)]	
[(12,3) - (12,0)]	
[(12,0) - (13,0)]	
[(13,0) - (13,5)]	
[(13,5) - (16,5)]	
[(16,5) - (16,0)]	

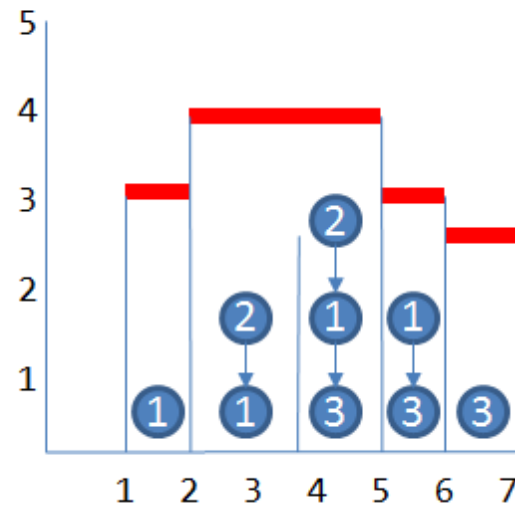
Note In my first attempt to solve this problem, I coded the solution *before* I considered all possible cases, then realized the implementation was incorrect. You can avoid wasting time on faulty implementations by checking your pseudocode against a real example.

As you uncover the missing case that you hadn't considered before, it looks like the whole approach will have to change.

CASE FOUR



Revised Approach

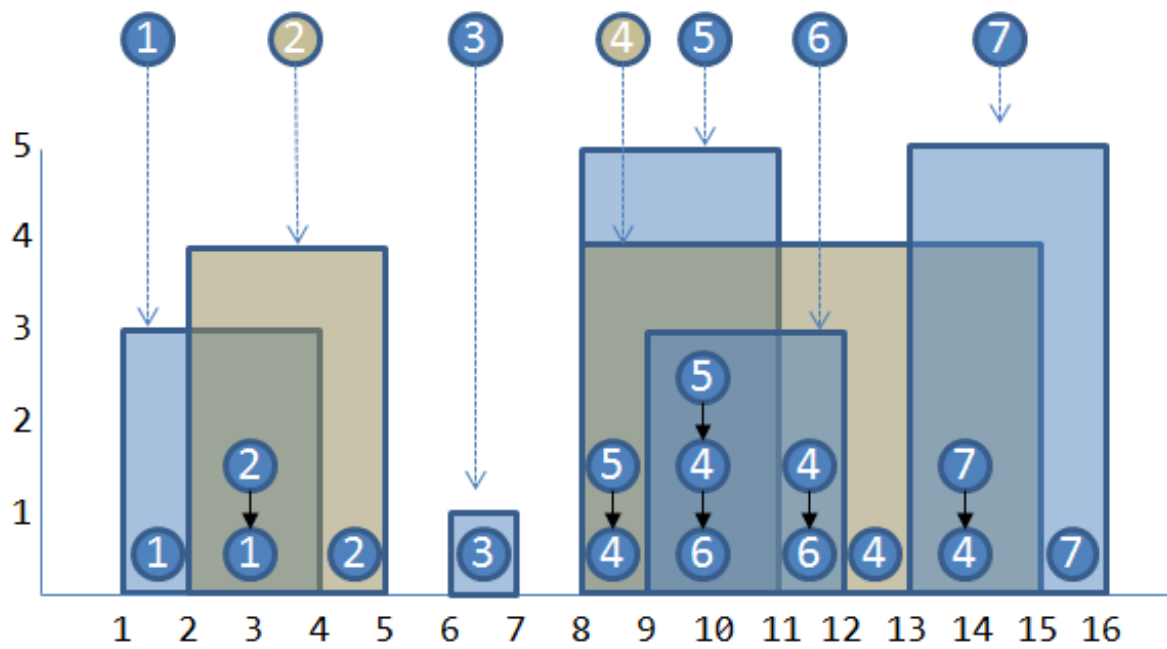


In **Case Four**, once you have processed the second building, the height of the next Skyline point will be at the original first building, not the third building processed. The original approach focused too much on the **left** coordinates of the buildings; now you can see that the **right** coordinate of the buildings is just as important.

Let's approach the problem from another perspective. Another way to define the Skyline for a set of buildings is to consider only the tops of buildings, that is, just the horizontal edges. The top of a building at a given x-coordinate is part of the Skyline *if no other building at that x-coordinate is taller*. This is a simpler way to approach the whole problem; often after you have worked on a problem for some time, a simpler solution will materialize. The earlier observation that the Skyline contained alternating vertical and horizontal edges was actually a distraction from this simpler approach.

The revised approach to this problem (as shown above in Case Four) still involves attempts to "sweep" the buildings from left to right, but now it gives equal weight to both the left- and right- coordinates of a building. Now you need to maintain an ordered *heightList* of buildings (from tallest down to smallest) as the x-coordinate sweeps from left to right. The ordered list (represented by a chain of circles) keeps track of the buildings. When you sweep *x* from left to right and discover the left edge of a building at that x-coordinate, add the building to this ordered list *at its proper location (according to its height)*. When the x-coordinate matches the right-coordinate of a building in *heightList*, **remove** that building from the list. Now comes the key observation: *Whenever the top of the ordered list changes, the Skyline changes as well.*

Here's the expected behavior when we manage the *heightList* structure for the sample input set described earlier:



Let's get started with adding code to sweep through the coordinates and constructs *heightList*. Modify the **Skyline** class as shown:

CODE TO TYPE: Modifications to Skyline class

```
package skyline;

import java.io.*;
import java.util.*;

public class Skyline {

    public static Collection<Building> retrieveInput(InputStream is) {
        ArrayList<Building> buildings = new ArrayList<Building>();
        Scanner sc = new Scanner (is);
        while (sc.hasNextLine()) {
            String s = sc.nextLine();
            if (s.equals("")) { break; }

            try {
                StringTokenizer st = new StringTokenizer(s);
                int left = Integer.valueOf(st.nextToken());
                int right = Integer.valueOf(st.nextToken());
                int height = Integer.valueOf(st.nextToken());

                Building b = new Building (left, right, height);
                buildings.add(b);
            } catch (NumberFormatException nfe) {
                System.err.println(" ** Ignoring " + s + ": all values must be integers.
");
            } catch (Exception e) {
                System.err.println(" ** Ignoring " + s + ": " + e.getMessage());
            }
        }

        return (buildings);
    }

    public static void compute(Collection<Building> buildings) {
        TreeSet<Integer> S = new TreeSet<Integer>();
        HashMap<Integer,ArrayList<Building>> lefts = new HashMap<Integer,ArrayList<Building>>();
        HashMap<Integer,ArrayList<Building>> rights = new HashMap<Integer,ArrayList<Building>>();
        ArrayList<Building> list = null;
        for (Building b : buildings) {
            S.add(b.left);
            list = lefts.get(b.left);
            if (list == null) {
                list = new ArrayList<Building>();
                lefts.put(b.left, list);
            }
            list.add(b);

            S.add(b.right);
            list = rights.get(b.right);
            if (list == null) {
                list = new ArrayList<Building>();
                rights.put(b.right, list);
            }
            list.add(b);
        }

        ArrayList<Building> heightList = new ArrayList<Building>();
        for (int x : S) {
            list = rights.get(x);
            if (list != null) {
                for (Building b : list) {
                    heightList.remove(b);
                }
            }
        }
    }
}
```

```

        list = lefts.get(x);
        if (list != null) {
            for (Building b : list) {
                int i;
                for (i = 0; i < heightList.size(); i++) {
                    if (heightList.get(i).height < b.height) {
                        heightList.add(i, b);
                        break;
                    }
                }
                if (i == heightList.size()) {
                    heightList.add(b);
                }
            }
        }

        System.out.println(x + ":" + heightList);
    }
}

public static void main(String[] args) {
    Collection<Building> buildings = retrieveInput(System.in);

    compute(buildings);
    for (Building b : buildings) {
        System.out.println("[ " + b.left + ", " + b.right + " ] @ " + b.height);
    }
}

```

Execute this program on the sample input from before:

INTERACTIVE SESSION: Maintaining HeightList

```

1 4 3
6 7 1
8 15 4
8 11 5
9 12 3
2 5 4
13 16 5

1: [[1,4] @ 3]
2: [[2,5] @ 4, [1,4] @ 3]
4: [[2,5] @ 4]
5: []
6: [[6,7] @ 1]
7: []
8: [[8,11] @ 5, [8,15] @ 4]
9: [[8,11] @ 5, [8,15] @ 4, [9,12] @ 3]
11: [[8,15] @ 4, [9,12] @ 3]
12: [[8,15] @ 4]
13: [[13,16] @ 5, [8,15] @ 4]
15: [[13,16] @ 5]
16: []

```

Each line of output shows the *heightList* of buildings, in reverse order of height. Compare the output to the *heightList* image we just saw; the results at each x-coordinate accurately reflect the order of buildings in *heightList* at each coordinate. Let's take a closer look at the code:

OBSERVE: Creating initial data structures S, lefts and rights

```
public static void compute(Collection<Building> buildings) {
    TreeSet<Integer> S = new TreeSet<Integer>();
    HashMap<Integer, ArrayList<Building>> lefts = new HashMap<Integer, ArrayList<Building>>();
    HashMap<Integer, ArrayList<Building>> rights = new HashMap<Integer, ArrayList<Building>>();
    ArrayList<Building> list = null;
    for (Building b : buildings) {
        S.add(b.left);
        list = lefts.get(b.left);
        if (list == null) {
            list = new ArrayList<Building>();
            lefts.put(b.left, list);
        }
        list.add(b);

        S.add(b.right);
        list = rights.get(b.right);
        if (list == null) {
            list = new ArrayList<Building>();
            rights.put(b.right, list);
        }
        list.add(b);
    }
}
```

This code constructs a set **S** from the left- and right- coordinates of the buildings so it can sweep through the coordinates from left to right. In the previous lesson, we learned that sets have no inherent ordering associated with them; they simply maintain a collection of unique elements. In practice, however, the **TreeSet** class in the Collections Framework can store the elements of a set efficiently *and allow you to iterate over these elements in sorted order*. The algorithm pseudocode shows that you need to be able to retrieve quickly, all buildings whose left- (or right-) coordinate is a specific value. This behavior calls for an associative Map of some kind. Here the code creates two **HashMap** objects. **lefts** enables the retrieval of an **ArrayList** of **Building** objects that all share the same left x-coordinate. Similarly, the **rights** **HashMap** stores an **ArrayList** of **Building** objects that all share the same right x-coordinate.

To compute the number of operations in the above code, consider these actions:

- The **for** loop executes n times.
- Each **add** to a **TreeSet** is guaranteed to perform with $O(\log n)$ behavior.
- Each **get** on a **HashMap** is $O(1)$ time.
- Each **put** operation on a **HashMap** is $O(1)$ time.
- Each **add** on a **ArrayList** is amortized constant time.

The total number of operations is $2*n*(O(\log n) + O(1) + O(1) + \textit{Amortized Constant})$. The above is classified as an $O(n \log n)$ algorithm because those are the dominant terms in the computation.

Throughout this course you will be asked to evaluate the run-time performance of an algorithm in the same manner. Make sure you understand the reason behind classifying the performance of this initialization code as $O(n \log n)$.

Once **lefts**, **rights**, and **S** are constructed, the **compute** method must sweep through the coordinates from left to right. It does so by iterating over all integer values in **S**, which are processed in ascending order.

OBSERVE: Manage heightList

```
ArrayList<Building> heightList = new ArrayList<Building>();  
for (int x : S) {  
    list = rights.get(x);  
    if (list != null) {  
        for (Building b : list) {  
            heightList.remove(b);  
        }  
    }  
  
    list = lefts.get(x);  
    if (list != null) {  
        for (Building b : list) {  
            int i;  
            for (i = 0; i < heightList.size(); i++) {  
                if (heightList.get(i).height < b.height) {  
                    heightList.add(i, b);  
                    break;  
                }  
            }  
            if (i == heightList.size()) {  
                heightList.add(b);  
            }  
        }  
    }  
  
    System.out.println(x + ":" + heightList);  
}
```

The **for loop** iterates over every coordinate value **x**. First it **removes from heightList** all buildings with the right-coordinate of **x**; these buildings can no longer affect the Skyline. Then the **for loop inserts into heightList** all of the buildings with a left-coordinate of **x**. **If any exist**, the **Building** objects in that list are inserted *at the proper location* in heightList. Observe how the above code keeps heightList in order (from tallest to shortest). The closing println statement outputs heightList so you can validate that the sweep is working properly.

Now that you have a working sweep that maintains the *heightList*, it's time to design the revised pseudocode for the algorithm.

OBSERVE: Pseudocode description of revised algorithm

```
compute ()
  S = set of integers containing all left- and right-coordinates of buildings
  lefts = HashMap of buildings by left-coordinate
  rights = HashMap of buildings by right-coordinate
  heightList = empty

  skyline = empty
  foreach x in S in sorted order do
    if heightList is empty then
      top = 0
    else
      top = tallest building in heightList

    foreach building b whose b.right=x do
      remove b from heightList
    foreach building b whose b.left=x do
      insert b into heightList at appropriate location

    if heightList is empty then
      newTop = 0
    else
      newTop = tallest building in heightList

    if top is 0 then
      left = x
    else if top != newTop then
      add edge (left, top) - (x, top) into skyline
      left = x

  return skyline
```

The pseudocode demonstrates how to generate a set of edges while sweeping the coordinates from left to right. With each pass through the foreach loop, the algorithm determines if the top of the tallest building in heightList changes because a building is removed from or added to the heightList. If a change happens, then **newTop != top** and a horizontal edge can be determined for the Skyline. With each pass through the **foreach** loop, *left* records the most recent x-coordinate for processing.

To complete the implementation, you need a class to represent the edges in the Skyline.

 Create an **Edge** class in the **skyline** package of the **/src** source folder:

CODE TO TYPE: Edge class

```
package skyline;

import java.awt.Point;

public class Edge {
    final Point start;
    final Point end;

    public Edge (Point start, Point end) {
        this.start = start;
        this.end = end;
    }

    public String toString() {
        return "[" + start.x + "," + start.y + ") - (" + end.x + "," + end.y + "]"
    }
}
```

The **Edge** class simply records an edge by using two **java.awt.Point** objects. It has a convenient **toString** method for debugging.

To compute the Skyline of horizontal edges you need to record whenever the *top of the ordered heightList changes*. Make these code modifications to **Skyline**:

CODE TO TYPE: Modifications to Skyline to compute edges of Skyline

```
package skyline;

import java.io.*;
import java.util.*;
import java.awt.Point;

public class Skyline {

    public static Collection<Building> retrieveInput(InputStream is) {
        ArrayList<Building> buildings = new ArrayList<Building>();
        Scanner sc = new Scanner (is);
        while (sc.hasNextLine()) {
            String s = sc.nextLine();
            if (s.equals("")) { break; }

            try {
                StringTokenizer st = new StringTokenizer(s);
                int left = Integer.valueOf(st.nextToken());
                int right = Integer.valueOf(st.nextToken());
                int height = Integer.valueOf(st.nextToken());

                Building b = new Building (left, right, height);
                buildings.add(b);
            } catch (NumberFormatException nfe) {
                System.err.println(" ** Ignoring " + s + ": all values must be integers.
");
            } catch (Exception e) {
                System.err.println(" ** Ignoring " + s + ": " + e.getMessage());
            }
        }

        return (buildings);
    }

    public static ArrayList<Edge> void compute(Collection<Building> buildings) {
        TreeSet<Integer> S = new TreeSet<Integer>();
        HashMap<Integer,ArrayList<Building>> lefts = new HashMap<Integer,ArrayList<B
uilding>>();
        HashMap<Integer,ArrayList<Building>> rights = new HashMap<Integer,ArrayList<
Building>>();
        ArrayList<Building> list = null;
        for (Building b : buildings) {
            S.add(b.left);
            list = lefts.get(b.left);
            if (list == null) {
                list = new ArrayList<Building>();
                lefts.put(b.left, list);
            }
            list.add(b);

            S.add(b.right);
            list = rights.get(b.right);
            if (list == null) {
                list = new ArrayList<Building>();
                rights.put(b.right, list);
            }
            list.add(b);
        }

        int left = 0, top = 0;
        ArrayList<Edge> skyline = new ArrayList<Edge>();
        ArrayList<Building> heightList = new ArrayList<Building>();
        for (int x : S) {
            if (heightList.isEmpty()) {
                top = 0;
            } else {
```

```

        top = heightList.get(0).height;
    }

    list = rights.get(x);
    if (list != null) {
        for (Building b : list) {
            heightList.remove(b);
        }
    }

    list = lefts.get(x);
    if (list != null) {
        for (Building b : list) {
            int i;
            for (i = 0; i < heightList.size(); i++) {
                if (heightList.get(i).height < b.height) {
                    heightList.add(i, b);
                    break;
                }
            }
            if (i == heightList.size()) {
                heightList.add(b);
            }
        }
    }

    int newTop;
    if (heightList.isEmpty()) {
        newTop = 0;
    } else {
        newTop = heightList.get(0).height;
    }

    if (top == 0) {
        left = x;
    } else if (top != newTop) {
        Edge e = new Edge(new Point (left, top), new Point (x, top));
        skyline.add(e);
        left = x;
    }
    System.out.println(x + ":" + heightList);
}
return (skyline);
}

public static void main(String[] args) {
    Collection<Building> buildings = retrieveInput(System.in);

    for (Edge e : compute(buildings)) {➤
        System.out.println(e);
    }
}
}

```

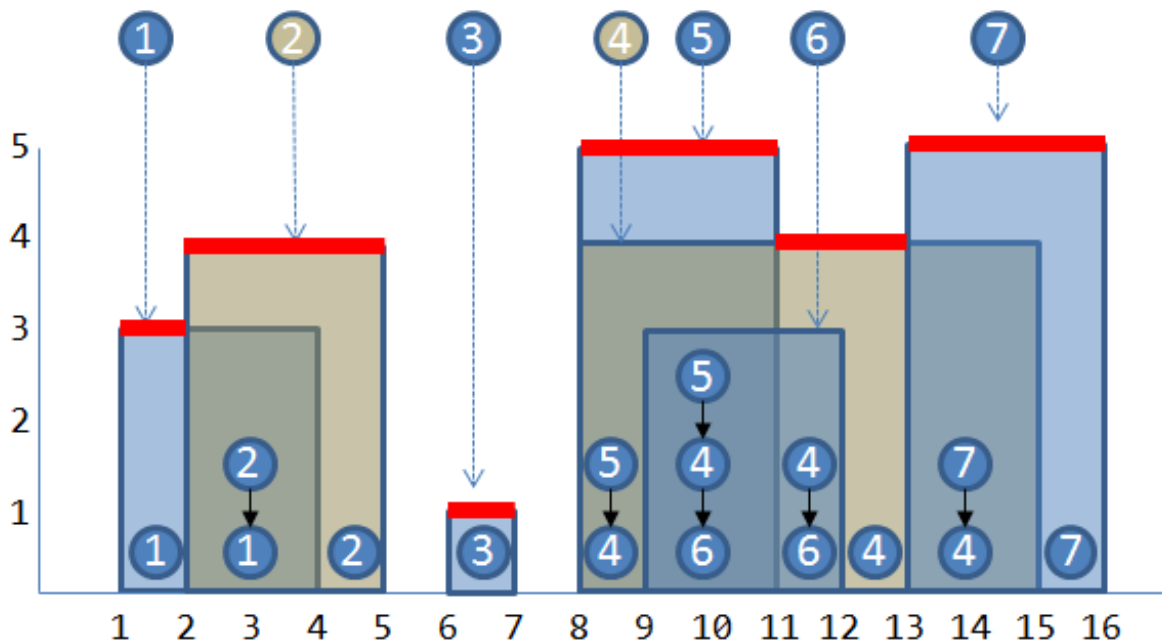
This approach will record all of the horizontal lines (shown in red in previous images) that form the tops of buildings. Execute this program on the original set of buildings to produce this set of horizontal edges:

INTERACTIVE SESSION: Output of horizontal edges in Skyline

```
1 4 3
6 7 1
8 15 4
8 11 5
9 12 3
2 5 4
13 16 5
```

```
[ (1,3) - (2,3) ]
[ (2,4) - (5,4) ]
[ (6,1) - (7,1) ]
[ (8,5) - (11,5) ]
[ (11,4) - (13,4) ]
[ (13,5) - (16,5) ]
```

Compare these edges with the image below that highlights the edges in the Skyline in red; these are the horizontal edges in the final Skyline:



To complete this algorithm, you add a helper method that completes the Skyline which contains only horizontal edges forming the tops of each building. Since the edges in the Skyline were added from left to right, you must "stitch together" vertical edges to connect them, but you must also handle gaps that form when there is a space between two buildings. The pseudocode below for **complete()** describes this process. In the pseudocode, **edge.start** refers to the left point of a horizontal edge and **edge.end** refers to its right point. Each point has an x-coordinate and a y-coordinate, so **edge.end.x** refers to the x-coordinate of the right point of the given edge:

OBSERVE: Pseudocode for complete() method

```
complete(skyline)
  skylinepoints = empty
  left = leftmost coordinate of edges in skyline
  right = rightmost coordinate of edges in skyline

  append (left, 0) to skylinepoints
  foreach edge in skyline do
    append edge.start to skylinepoints
    append edge.end to skylinepoints

    nextEdge = next edge in skyline
    if nextEdge exists then
      if edge.end does not have same x-coordinate as nextEdge.start then
        append (edge.end.x, 0) to skylinepoints
        append (nextEdge.start.x, 0) to skylinepoints

  append (right, 0) to skylinepoints
  return skylinepoints
```

The following implementation shows the final modifications to complete the Skyline problem.

Skyline Final Implementation

```
package skyline;

import java.io.*;
import java.util.*;
import java.awt.Point;

public class Skyline {

    public static Collection<Building> retrieveInput(InputStream is) {
        ArrayList<Building> buildings = new ArrayList<Building>();
        Scanner sc = new Scanner (is);
        while (sc.hasNextLine()) {
            String s = sc.nextLine();
            if (s.equals("")) { break; }

            try {
                StringTokenizer st = new StringTokenizer(s);
                int left = Integer.valueOf(st.nextToken());
                int right = Integer.valueOf(st.nextToken());
                int height = Integer.valueOf(st.nextToken());

                Building b = new Building (left, right, height);
                buildings.add(b);
            } catch (NumberFormatException nfe) {
                System.err.println(" ** Ignoring " + s + ": all values must be integers.
");
            } catch (Exception e) {
                System.err.println(" ** Ignoring " + s + ": " + e.getMessage());
            }
        }

        return (buildings);
    }

    public static ArrayList<Edge> compute(Collection<Building> buildings) {
        TreeSet<Integer> S = new TreeSet<Integer>();
        HashMap<Integer,ArrayList<Building>> lefts = new HashMap<Integer,ArrayList<Building>>();
        HashMap<Integer,ArrayList<Building>> rights = new HashMap<Integer,ArrayList<Building>>();
        ArrayList<Building> list = null;
        for (Building b : buildings) {
            S.add(b.left);
            list = lefts.get(b.left);
            if (list == null) {
                list = new ArrayList<Building>();
                lefts.put(b.left, list);
            }
            list.add(b);

            S.add(b.right);
            list = rights.get(b.right);
            if (list == null) {
                list = new ArrayList<Building>();
                rights.put(b.right, list);
            }
            list.add(b);
        }

        int left = 0, top = 0;
        ArrayList<Edge> skyline = new ArrayList<Edge>();
        ArrayList<Building> heightList = new ArrayList<Building>();
        for (int x : S) {
            if (heightList.isEmpty()) {
                top = 0;
            } else {
```

```

        top = heightList.get(0).height;
    }

    list = rights.get(x);
    if (list != null) {
        for (Building b : list) {
            heightList.remove(b);
        }
    }

    list = lefts.get(x);
    if (list != null) {
        for (Building b : list) {
            int i;
            for (i = 0; i < heightList.size(); i++) {
                if (heightList.get(i).height < b.height) {
                    heightList.add(i, b);
                    break;
                }
            }
            if (i == heightList.size()) {
                heightList.add(b);
            }
        }
    }

    int newTop;
    if (heightList.isEmpty()) {
        newTop = 0;
    } else {
        newTop = heightList.get(0).height;
    }

    if (top == 0) {
        left = x;
    } else if (top != newTop) {
        Edge e = new Edge(new Point (left, top), new Point (x, top));
        skyline.add(e);
        left = x;
    }
}
return (skyline);
}

public static Collection<Point> complete(ArrayList<Edge> skyline) {
    ArrayList<Point> skylinepoints = new ArrayList<Point>();
    if (skyline.isEmpty()) { return skylinepoints; }

    int left = skyline.get(0).start.x;
    int right = skyline.get(skyline.size()-1).end.x;

    skylinepoints.add(new Point (left, 0));

    for (int i = 0; i < skyline.size(); i++) {
        Edge edge = skyline.get(i);
        skylinepoints.add(edge.start);
        skylinepoints.add(edge.end);

        Edge nextEdge = null;
        if (i+1 < skyline.size()) {
            nextEdge = skyline.get(i+1);

            if (edge.end.x != nextEdge.start.x) {
                skylinepoints.add(new Point (edge.end.x, 0));
                skylinepoints.add(new Point (nextEdge.start.x, 0));
            }
        }
    }
}

```

```

        skylinepoints.add(new Point (right, 0));
        return (skylinepoints);
    }

    public static void main(String[] args) {
        Collection<Building> buildings = retrieveInput(System.in);

        ArrayList<Edge> skyline = compute(buildings);
        Collection<Point> skylinepoints = complete(skyline);
        for (Point p : skylinepoints) {
            System.out.print("(" + p.x + "," + p.y + ") ");
        }
        for (Edge e : compute(buildings)) {
            System.out.println(e);
        }
    }
}

```

Execute the final program to validate that it works on the sample input set:

INTERACTIVE SESSION: Final output from Skyline program

```

1 4 3
6 7 1
8 15 4
8 11 5
9 12 3
2 5 4
13 16 5

(1,0) (1,3) (2,3) (2,4) (5,4) (5,0) (6,0) (6,1) (7,1) (7,0) (8,0) (8,5) (11,5) (
11,4)
(13,4) (13,5) (16,5) (16,0)

```

Lessons Learned

This lesson demonstrates an iterative approach to algorithm development. The challenge is to identify milestones along the way where you can validate your progress. Instead of trying to solve the whole problem all at once, find ways to break the problem into sub-tasks. Our first attempt to solve the Skyline problem identified a number of cases that we believed to be every possible way that the Skyline would grow when buildings intersected with each other. In retrospect, this ad hoc solution didn't capture all the ways that n buildings could intersect each other. You need to find meaningful milestones that represent the different *stages* with the processing phase of an algorithm. Each milestone has a well-defined validation condition that you could test using test cases. In Skyline, everything started to work once the *heightList* abstraction was identified; that's allowed us to identify the horizontal edges in the Skyline. Once that work was completed and validated, the second stage of the algorithm just stitched the edges together to form the Skyline.

- 1. When you cannot fully order the elements, try to find a way to sweep from left to right across a partially ordered set:** the sweeping technique described in this lesson can be used to fully process each element in a set that cannot be completely ordered.
- 2. Even though a set is inherently unordered, the TreeSet allows you to iterate over its elements in order:** while **Set** implementations cannot be sorted in the same way that **List** implementations can, you can use its efficient iterator to inspect each of the elements in sorted order.
- 3. Use ArrayList when you need to maintain a list in some sorted order and then insert new elements into their proper locations within the list:** while **LinkedList** and **ArrayList** are both **Lists** that can be sorted, you cannot insert the element into its proper location efficiently in **LinkedList** because its **get(idx)** method executes in $O(n)$ time where n represents the number of elements in the list. Only **ArrayList** guarantees a constant time performance for this operation.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Working With Big Data

Lesson Objectives


When you finish this lesson you will be able to:

- characterize the storage requirement for an algorithm.
- access the contents of a structured binary file in the same way that you would an array stored in main memory.
- read from and write to a memory-mapped file.

Working with Big Data

What if you had to sort a collection of integers? The following example shows how to use the built-in sorting capabilities provided by the JDK.

Create a **BigData** project, and assign it to the **Java6_Lessons** working set.

 Then, create a **SortRandomIntegers** class in the default package of the **/src** source folder:

CODE TO TYPE: SortingExample

```
import java.util.Arrays;

public class SortRandomIntegers {
    public static void main(String[] args) {
        int numIntegers = 1000;
        int[] group = new int[numIntegers];

        for (int i = 0; i < numIntegers; i++) {
            group[i] = (int) (Math.random() * numIntegers);
        }

        Arrays.sort(group);

        for (int i = 0; i < 10; i++) {
            System.out.println(group[i]);
        }
    }
}
```

 Run the code to verify that it prints out ten numbers in sorted order.

This small program generates a random array containing 1000 integers, sorts them, and prints out the smallest ten in the array. You should always use the **Arrays.sort** built-in methods to sort arrays because it provides tuned algorithms with a performance that is nearly always $O(n \log n)$. For *comparison-based* sorting algorithms (where you can only sort the elements by directly comparing the individual elements) this is the best we've got.

Now what if you have a large collection of integers? Like 450 million? If you modify the settings of the above program to generate **450000000** integers instead of **1000** integers, you'll see this error message:

OBSERVE: Unable to create large arrays in memory

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at SortRandomIntegers.main(SortRandomIntegers.java:8)

The primary issue with this code is that it is simply impossible to create a contiguous array to contain a large collection of elements. You can try to increase the heap space available to your Java virtual machine, but eventually the computer on which you are running will exhaust its available memory. So how is it possible to deal with extremely large data sets? You will need to develop techniques that manage the transfer of data from external storage (such as a hard disk) into main memory (what is commonly called RAM). In the early days of computing, main memory was measured in kilobytes (not gigabytes!) and programmers learned how to work within these constraints. In this era of "Big Data" where data can be measured in terabytes and petabytes, even modern programmers have to make some fundamental

adjustments.

In this lesson, you'll learn how to sort large sets stored on disk, sets that may be too large to store in main memory. We'll show examples using small data sets, but they can scale to much larger data sets as needed.

Sorting Large Sets Using External Storage

Most sorting algorithms operate over an array of values, swapping elements in the array until the elements are in order. The earlier **sort** method does that. However, when the number of elements being sorted is too large to store in main memory, there are sorting algorithms that allow us to use external storage. The fundamental algorithm to learn is called *MergeSort*. You've probably used this technique in the real world already. Suppose that you had a stack of 50 notecards, each containing a single number. To sort the whole stack, divide it into two stacks of 25 notecards each. Sort each of these two stacks individually, which results in two sorted stacks of notecards where you can see the topmost visible card in each stack. You can "merge" these two smaller stacks into a third sorted stack by repeatedly taking the card whose visible number is the smaller of the two. This merging process gives the algorithm its name.

MergeSort is recursive, since it breaks up a problem instance into two smaller instances of half the size. To stop the recursion, consider two cases:

1. Sorting a collection of two values: swap the first value with the second if they are out of order.
2. Sorting a collection with a single value: the collection is already sorted, so stop.

You have enough information to write the pseudocode now. The notation $|A|$ represents the size of the collection A:

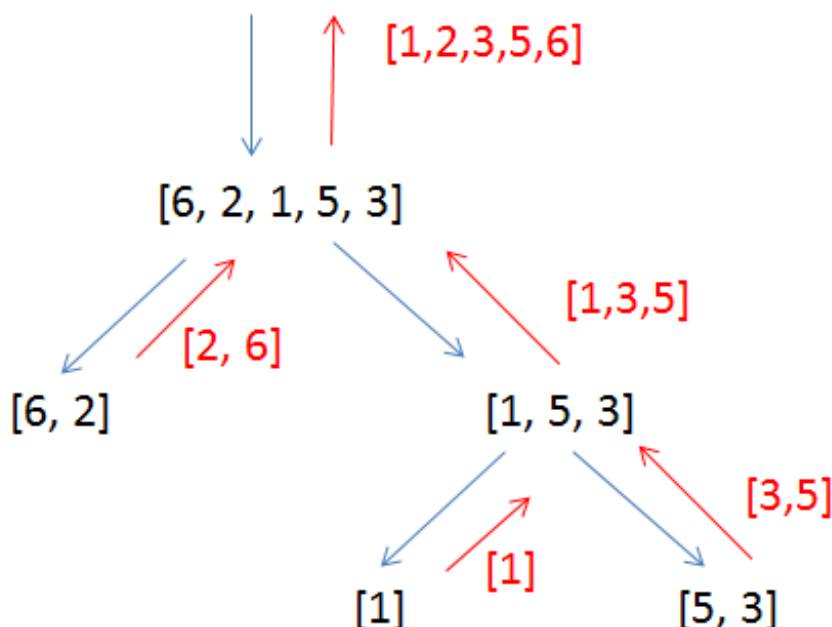
OBSERVE: pseudocode for Mergesort

```
MergeSort (A)
  if  $|A| < 2$  then return A
  if  $|A| = 2$  then
    swap elements of A if out of order
    return A


  sub1 = MergeSort(left half of A)
  sub2 = MergeSort(right half of A)


  merge sub1 and sub2 into a new array B
  return B
```

Try this out by manually executing MergeSort on the collection [6, 2, 1, 5, 3]. In the following graphic, the blue arrows represent invocations of *MergeSort* and the red arrows represent the returned sorted arrays. The newly created arrays are depicted in red and contain three or more elements (that is, [1,3,5] and [1,2,3,5,6]):



Now it's your turn to implement this algorithm:

 Create a **sort** package in the **/src** source folder.

 Create a **CopyMergeSort** class in the **sort** package as shown.

CODE TO TYPE: CopyMergeSort class

```
package sort;

import java.util.Arrays;

public class CopyMergeSort {
    public static void main(String[] args) {
        int[] group = new int []{6, 2, 1, 5, 3};
        group = copymergesort(group);
        for (int i : group) {
            System.out.print (i + " ");
        }
    }

    static int[] copymergesort(int[] A) {
        if (A.length < 2) {
            return A;
        }

        if (A.length == 2) {
            if (A[0] > A[1]) {
                int tmp = A[0];
                A[0] = A[1];
                A[1] = tmp;
            }
            return A;
        }

        int mid = A.length/2;
        int[] left = Arrays.copyOfRange(A, 0, mid);
        int[] right = Arrays.copyOfRange(A, mid, A.length);

        left = copymergesort(left);
        right = copymergesort(right);

        for (int i = 0, j = 0, idx=0; idx < A.length; idx++) {
            if (j >= right.length || (i < left.length && left[i] < right[j])) {
                A[idx] = left[i++];
            } else {
                A[idx] = right[j++];
            }
        }

        return A;
    }
}
```

Run this code to verify that it works. You might experiment with different initial arrays to see how the code handles arrays with just 1 or 2 (or larger number of) elements:

OBSERVE: Output of CopyMergeSort

1 2 3 5 6

Let's take a closer look at this code. The base cases of the recursion are as follows:

OBSERVE: Base cases of CopyMergeSort recursion

```
static int[] copymergesort(int[] A) {
    if (A.length < 2) {
        return A;
    }

    if (A.length == 2) {
        if (A[0] > A[1]) {
            int tmp = A[0];
            A[0] = A[1];
            A[1] = tmp;
        }
        return A;
    }

    ...
}
```

The **copymergesort** method must return an **int[]** array, so these **if** statements both return the input array, **A**. When there are two elements in the array, the second **if** statement **swaps the two elements** if they are out of order.

OBSERVE: Recursive steps

```
int mid = A.length/2;
int[] left = Arrays.copyOfRange(A, 0, mid);
int[] right = Arrays.copyOfRange(A, mid, A.length);

left = copymergesort(left);
right = copymergesort(right);
```

The true logic of this algorithm occurs when the array **A** is subdivided into two arrays, **left** and **right**, which are then recursively sorted using **copymergesort**.

OBSERVE: Merging two sorted arrays

```
for (int i = 0, j = 0, idx=0; idx < A.length; idx++) {
    if (j >= right.length || (i < left.length && left[i] < right[j])) {
        A[idx] = left[i++];
    } else {
        A[idx] = right[j++];
    }
}

return A;
}
```

Once the two recursive calls return, **left** and **right** will be sorted (this is the fundamental property of any recursive function). All that remains is the process of selecting the smaller of the two elements while merging these two lists. The code above reuses array **A** to store the sorted values. Variable **i** will iterate over the indices in **left**, while **j** will iterate over the indices in **right**. **idx** identifies the index location in **A** into which the smaller value of **left[i]** or **right[j]** will be written. The loop terminates once all values have been transferred into **A** (that is, when **idx = A.length**). Once **right** has exhausted its elements (because **j >= right.length**), elements of **left** are transferred to **A**. Similarly, once **left** has exhausted its elements (because **i >= left.length**), elements of **right** are transferred to **A**.

This code works, and it's reasonably efficient on small sets of numbers, but we also need space for the **left** and **right** arrays. To address this issue, you need to learn how to characterize the storage requirements for an algorithm.

Characterizing Storage Requirements for an Algorithm

Throughout this course, you have characterized the running time of an algorithm to determine its efficiency. This is how algorithms are most often compared. You can also compare algorithms by their storage requirements. There is a "Time vs. Space" tradeoff in programming that explains many of the design

decisions that a programmer must make. For example, each Java class that is used as a key value in a **HashMap** must implement a **hashCode()** method as part of the Collections Framework. As we've mentioned, if two objects are equal to each other, then the value returned by **hashCode** must also be the same. For immutable classes (such as **String**), a program can save computation time by computing the hash value just once and then caching the result for subsequent invocations. Here is the code from **java.lang.String**:

OBSERVE: **String.hashCode()** method

```
public int hashCode() {
    int h = hash;
    int len = count;

    if (h == 0 && len > 0) {
        int off = offset;
        char val[] = value;

        for (int i = 0; i < len; i++) {
            h = 31*h + val[off++];
        }
        hash = h;
    }

    return h;
}
```

Whenever **hashCode** is executed, it checks to see if the cached value **hash** is equal to zero; only then does it compute and store the value in the **hash** class attribute. This code is more efficient because of the extra integer being stored. How much extra storage is required? In this case it's a fixed amount of storage—just one additional **int** value. When addressing more complicated algorithms, you will need to determine whether the amount of extra storage is fixed, or is based on the size of the problem instance. For example, if you needed $2*n$ additional stored array elements to sort an existing array of n elements, you would characterize the storage requirements as being $O(n)$. If, however, you needed $n*n$ additional array elements to sort an existing array of n elements, the required storage is $O(n^2)$. We use the following notation in this course. $T(n)$ refers to the *running time* characterization of an algorithm, $S(n)$ refers to the *storage requirements* of an algorithm. Modify the **CopyMergeSort** code as shown:

CODE TO TYPE: Modifications to CopyMergeSort to compute storage requirements

```
package sort;
import java.util.Arrays;

public class CopyMergeSort {
    static int total=0;
    public static void main(String[] args) {
int[] group = new int []{6, 2, 1, 5, 3};
        int numIntegers = 512;
        for (; numIntegers < 65536; numIntegers *= 2) {
            int[] group = new int[numIntegers];

            for (int i = 0; i < numIntegers; i++) {
                group[i] = (int) (Math.random()*numIntegers);
            }
            total = 0;
            group = copymergesort(group);
            System.out.println(total + " locations for " + numIntegers);
        }
for (int i : group) {
    System.out.print (i + " ");
}
    }

    static int[] copymergesort(int[] A) {
        if (A.length < 2) {
            return A;
        }
        if (A.length == 2) {
            if (A[0] > A[1]) {
                int tmp = A[0];
                A[0] = A[1];
                A[1] = tmp;
            }
            return A;
        }

        int mid = A.length/2;
        int[] left = Arrays.copyOfRange(A, 0, mid);
        int[] right = Arrays.copyOfRange(A, mid, A.length);

        left = copymergesort(left);
        right = copymergesort(right);

        for (int i = 0, j = 0, idx=0; idx < A.length; idx++) {
            if (j >= right.length || (i < left.length && left[i] < right[j])) {
                A[idx] = left[i++];
            } else {
                A[idx] = right[j++];
            }
        }

        total += A.length;
        return A;
    }
}
```

Execute this revised code to produce this table:

OBSERVE: Output showing storage requirements for CopyMergeSort

```
4096 locations for 512
9216 locations for 1024
20480 locations for 2048
45056 locations for 4096
98304 locations for 8192
212992 locations for 16384
458752 locations for 32768
```

When sorting 512 elements you need 8 times as much temporary storage; worse, when sorting 2,048 elements you need 10 times as much temporary storage. Based on the above table, when sorting n elements you need $2 \cdot n \cdot \log_2(n)$ temporary storage where $\log_2(n)$ is the logarithm of n in base 2. So, the storage requirement for CopyMergeSort is $O(n \log n)$. Even though CopyMergeSort executes efficiently, there is a serious issue regarding its storage requirements. Can something be done to remedy this? Yes.

MergeSort with $O(n)$ Storage Requirements

Most sorting algorithms already perform "in place" with no additional storage requirements, so you might think that some intermediate compromise can be reached to reduce the storage requirements. You don't need to instantiate two sub-arrays **left** and **right** if you **instead pass parameters that refer to subranges within the array itself**. Let's start by revising the pseudocode for *MergeSort* to create a method that takes an array, **A**, and two internal indices, **[start, end]** where index location **start** is **inclusive** in the range **0 .. A.length-1** while **end** is **exclusive** in the range **0 .. A.length**. So, to sort an array one would invoke **MergeSort(A, 0, A.length)**. Note that the sorting is done "in place" so an array is no longer returned by this function.

OBSERVE: potential revised pseudocode for Mergesort

```
MergeSort (A, start, end)
  if end - start < 2 then return
  if end - start = 2 then
    swap elements of A if out of order

  mid = (end + start)/2;
  MergeSort(A, start, mid);
  MergeSort(A, mid, end);

  merge A's left- and right- sorted sub-arrays
```

The trouble with this approach is that merging in place will ultimately require just as many comparisons (and possibly more element swaps) as sorting in place. To avoid this situation, consider making these change to the pseudocode which introduces a copy of the initial array being sorted, which means the storage requirement is $O(n)$:

OBSERVE: final pseudocode for Mergesort

```
MergeSort (A)
  copy = copy of A
  MergeSort (copy, A, 0, |A|)

MergeSort (A, result, start, end)
  if end - start < 2 then return
  if end - start = 2 then
    swap elements of result if out of order


  mid = (end + start)/2;
  MergeSort(result, A, start, mid);
  MergeSort(result, A, mid, end);

merge A's left- and right- sorted sub-arrays
merge left- and right- of A into result
```

Because **copy** is a true copy of the entire array, the terminating base cases of the recursion will work *because they reference the original elements of the array directly at their respective index locations*. This observation is a sophisticated one; when you run this implementation in the debugger, you can validate it for yourself. In

addition, the final merge step requires only $O(n)$ operations.

Now it's your turn to implement this pseudocode.

 In the **sort** package of the **/src** source folder, create a **MergeSortInteger** class as shown.

COE TO TYPE: MergeSortInteger class

```
package sort;

import java.util.Arrays;

public class MergeSortInteger {
    public static void main(String[] args) {
        int numIntegers = 1024;
        int[] group = new int[numIntegers];
        for (int i = 0; i < numIntegers; i++) {
            group[i] = (int) (Math.random()*numIntegers);
        }
        mergesort(group);

        for (int i = 0; i < 10; i++) {
            System.out.println(group[i]);
        }
    }


    static void mergesort (int[] A) {
        int[] copy = Arrays.copyOf(A, A.length);
        mergesort (copy, A, 0, A.length);
    }

    static void mergesort(int[] A, int[] result, int start, int end) {
        if (end - start < 2) {
            return;
        }

        if (end - start == 2) {
            if (result[end-2] > result[end-1]) {
                int tmp = result[end-2];
                result[end-2] = result[end-1];
                result[end-1] = tmp;
            }
            return;
        }

        int mid = (end + start)/2;
        mergesort(result, A, start, mid);
        mergesort(result, A, mid, end);

        for (int i = start, j = mid, idx=start; idx < end; idx++) {
            if (j >= end || (i < mid && A[i] < A[j])) {
                result[idx] = A[i++];
            } else {
                result[idx] = A[j++];
            }
        }
    }
}
```

 Run this code; you see the first ten randomly generated integers in sorted order. Let's review this code more closely:

OBSERVE: MergeSort invocation

```
static void mergesort (int[] A) {  
    int[] copy = Arrays.copyOf(A, A.length);  
    mergesort (copy, A, 0, A.length);  
}
```

To sort the array, we make a full copy and then internally invoke **mergesort** to sort the copy with **A** as the ultimate destination. Note that the arguments to pass in are **0** and **A.length**, which reflect the index values into A, namely inclusive on the left side with **0** and exclusive on the right side with **A.length**.

All logic once again resides in the recursive method. Let's review the base cases:

OBSERVE: Recursive base case of MergeSort

```
static void mergesort(int[] A, int[] result, int start, int end) {  
    if (end - start < 2) {  
        return;  
    }  
  
    if (end - start == 2) {  
        if (result[end-2] > result[end-1]) {  
            int tmp = result[end-2];  
            result[end-2] = result[end-1];  
            result[end-1] = tmp;  
        }  
        return;  
    }  
}
```

If **end - start** is less than 2, there is either no element or a single element to be sorted, which means nothing needs to be done. When **end-start** equals 2, there are two elements to be sorted. This code executes only as a base case in the recursion, which means that it's the first time the method is inspecting the array subrange of **[start,end)**. Because the result must be stored in the **result** array, this code reorders the values it finds there.

The final elements in **mergesort** show how to merge the sorted left and right sub-arrays:

OBSERVE: Merging in O(n) time

```
int mid = (end + start)/2;  
mergesort(result, A, start, mid);  
mergesort(result, A, mid, end);  
  
for (int i = start, j = mid, idx=start; idx < end; idx++) {  
    if (j >= end || (i < mid && A[i] < A[j])) {  
        result[idx] = A[i++];  
    } else {  
        result[idx] = A[j++];  
    }  
}
```


This code first **recursively sorts the left half and right half of the range [start, end)**, placing the properly ordered elements in the array referenced as **A**. Then it uses two indices, **i** and **j**, to iterate over each of these sub-ranges, always copying the smaller of **A[i]** and **A[j]** into the properly located **result[idx]**. There are three cases to consider:

1. The right side is exhausted (**j >= end**), in which case you can grab the remaining elements from **A[i]**.
2. The left side is exhausted (**i >= mid**), in which case you can grab the remaining elements from **A[j]**.
3. The left and right side have elements; if **A[i] < A[j]**, insert **A[i]**, otherwise insert **A[j]**.

Once the for loop completes, **result** has the merged (and sorted) elements from the subarray **[start, end)** of the original array A.

Working with Large Datasets

You are going to use *MergeSort* to sort large collections of values. You're going to need additional storage for that to work. You don't actually need to store the entire collection in main memory to sort its contents. Let's start by defining the problem instance. The input of n integers will be stored in a binary file containing $4*n$ bytes. Practice using this structure by writing this sample program:

 In the **sort** package of the **/src** source folder, create a class **BinaryIntegerFile** as shown:

CODE TO TYPE: BinaryIntegerFile


```
package sort;

import java.io.*;

public class BinaryIntegerFile {
    public static void main(String[] args) throws IOException {
        int numIntegers = 4096;
        File f = new File ("IntegerFile.bin");
        DataOutputStream dos = new DataOutputStream(new FileOutputStream(f));
        for (int i = 0; i < numIntegers; i++) {
            dos.writeInt((int) (Math.random()*numIntegers));
        }
        dos.close();

        DataInputStream dis = new DataInputStream(new FileInputStream(f));

        System.out.println("First five sorted numbers");
        for (int i = 0; i < 5; i++) {
            System.out.println(dis.readInt());
        }
        dis.skipBytes(4*(numIntegers-10));
        System.out.println("Last five sorted numbers");
        for (int i = 0; i < 5; i++) {
            System.out.println(dis.readInt());
        }
        dis.close();
    }
}
```

 Run this program; you'll see something like this (your numbers will be different because they are randomly generated):

INTERACTIVE SESSION: Output from BinaryIntegerFile. Note that sort function is not yet implemented.

```
First five sorted numbers
2935
3918
245
2885
2496
Last five sorted numbers
2748
3716
1972
2086
1350
```

Let's take a closer look at this code. The **java.io** package contains a number of classes to read and write information to the file system. The fundamental abstraction is a *stream*, which represents a sequence of data. An **InputStream** reads data from a source and an **OutputStream** writes data to a source. In the code, a **DataInputStream** is used to read primitive Java data types from the input stream (such as **int** and **float** values) while a **DataOutputStream** writes primitive Java data types to an output stream.

OBSERVE: Creating a random binary file of integers

```
int numIntegers = 4096;
File f = new File ("IntegerFile.bin");
DataOutputStream dos = new DataOutputStream(new FileOutputStream(f));
for (int i = 0; i < numIntegers; i++) {
    dos.writeInt((int) (Math.random()*numIntegers));
}
dos.close();
```

Using a **DataOutputStream** object, the above code writes 4,096 integers in binary format to the file and closes it. Once created, this file will contain 16,384 bytes because the integers are written in binary format where each integer value requires four bytes. Don't bother trying to open this file in Eclipse because the data is stored in binary format so Eclipse will just present you with the raw data. You can retrieve the integer values that were stored using **DataInputStream**, which properly decodes the binary formatted encoding of the integer values in the file. The second part of the code reads in this file and prints the first five integers and the last five integers in the file.


OBSERVE: Read integers from file

```
DataInputStream dis = new DataInputStream(new FileInputStream(f));

System.out.println("First five sorted numbers");
for (int i = 0; i < 5; i++) {
    System.out.println(dis.readInt());
}
dis.skipBytes(4*(numIntegers-10));
System.out.println("Last five sorted numbers");
for (int i = 0; i < 5; i++) {
    System.out.println(dis.readInt());
}
dis.close();
```

This code uses a **DataInputStream** to retrieve the values from the file. Note that it reads the first five integers and then **skips the requisite number of bytes** (there are four bytes for each integer) so it can then read the last five numbers from the file. Clearly this file isn't sorted; you'll solve this by implementing a *MergeSort* that operates over a **File** containing integer values, rather than an in-memory array of integer values.

To make this work, you have to access a file in the same way that you would otherwise access an array. You know the structure of *MergeSort* from the implementation you completed earlier, all you need to do now is map those concepts to a file. Consider using **RandomAccessFile**, provided by the **java.io** package, which allows you to access any byte within a file randomly. Knowing that the file contains a collection of integers in 4-byte format, you can determine that to read the *n*th int value from the file, you need to start reading 4 bytes from position $n*4$. Similar logic is used to write an integer to replace the *n*th int value in the file. All of these operations will succeed with index values of type **long**, which means you can process extremely large files if you want.

 In the **sort** package of the **/src** source folder, create a **MergeSortFile** class as shown:

CODE TO TYPE: MergeSortFile class

```
package sort;

import java.io.*;

public class MergeSortFile {

    static void mergesort (File A) throws IOException {
        File copy = new File (A.getPath() + ".tmp");
        copyFile(A, copy);

        // TBA: invoke MergeSort
    }

    static void copyFile(File src, File dest) throws IOException {
        FileInputStream fis = new FileInputStream(src);
        FileOutputStream fos = new FileOutputStream (dest);
        byte[] bytes = new byte[4*1048576];
        int numRead;
        while ((numRead = fis.read(bytes)) > 0) {
            fos.write(bytes, 0, numRead);
        }
        fis.close();
        fos.close();
    }
}
```

The **mergesort** method prepares for the algorithm by making a full copy of the source file, **A**. For demonstration purposes, the file copy is created in your workspace, but normally you would use the static method **File.createTempFile** instead to create a temporary file in the default temporary directory. The **copyFile** method copies bytes in chunks of four megabytes to replicate the file. To test out the above code, modify **BinaryIntegerFile** to use the **mergesort** method in **MergeSortFile**.

CODE TO TYPE: Modified BinaryIntegerFile

```
package sort;

import java.io.*;

public class BinaryIntegerFile {
    public static void main(String[] args) throws IOException {
        int numIntegers = 4096;
        File f = new File ("IntegerFile.bin");
        DataOutputStream dos = new DataOutputStream(new FileOutputStream(f));
        for (int i = 0; i < numIntegers; i++) {
            dos.writeInt((int) (Math.random()*numIntegers));
        }
        dos.close();

        long now = System.currentTimeMillis();
        MergeSortFile.mergesort(f);
        System.out.println((System.currentTimeMillis() - now) + " ms.");

        DataInputStream dis = new DataInputStream(new FileInputStream(f));

        System.out.println("First five sorted numbers");
        for (int i = 0; i < 5; i++) {
            System.out.println(dis.readInt());
        }
        dis.skipBytes(4*(numIntegers-10));
        System.out.println("Last five sorted numbers");
        for (int i = 0; i < 5; i++) {
            System.out.println(dis.readInt());
        }
        dis.close();
    }
}
```

Now execute **BinaryIntegerFile** and refresh your workspace. You will see two top-level files: **IntegerFile.bin** and **IntegerFile.bin.tmp**. Select both of these files in the Java package browser (holding down the **Shift** key and click each icon), and right-click on either file to select **Compare With | Each Other**. The files are identical, because you haven't yet written any code to sort the data.

You are now ready to complete the *MergeSort* implementation. Modify **MergeSortFile** as follows:

Modified MergeSortFile

```
package sort;

import java.io.*;

public class MergeSortFile {

    static void mergesort (File A) throws IOException {
        File copy = new File (A.getPath() + ".tmp");
        copyFile(A, copy);

        // TBA: invoke MergeSort
        RandomAccessFile src = new RandomAccessFile(A, "rw");
        RandomAccessFile dest = new RandomAccessFile(copy, "rw");

        mergesort (dest, src, 0, A.length());
        src.close();
        dest.close();
        copy.delete();
    }

    static void copyFile(File src, File dest) throws IOException {
        FileInputStream fis = new FileInputStream(src);
        FileOutputStream fos = new FileOutputStream (dest);
        byte[] bytes = new byte[4*1048576];
        int numRead;
        while ((numRead = fis.read(bytes)) > 0) {
            fos.write(bytes, 0, numRead);
        }
        fis.close();
        fos.close();
    }

    static void mergesort(RandomAccessFile A, RandomAccessFile result,
        long start, long end) throws IOException {

        if (end - start < 8) {
            return;
        }

        if (end - start == 8) {
            result.seek(end-8);
            int left = result.readInt();
            int right = result.readInt();
            if (left > right) {
                result.seek(end-8);
                result.writeInt(right);
                result.writeInt(left);
            }
            return;
        }

        long mid = (end + start)/8*4;
        mergesort(result, A, start, mid);
        mergesort(result, A, mid, end);

        result.seek(start);
        for (long i = start, j = mid, idx=start; idx < end; idx += 4) {
            A.seek(i);
            int Ai = A.readInt();
            int Aj = 0;
            if (j < end) { A.seek(j); Aj = A.readInt(); }
            if (j >= end || (i < mid && Ai < Aj)) {
                result.writeInt(Ai);
                i += 4;
            } else {
                result.writeInt(Aj);
            }
        }
    }
}
```

```

        j += 4;
    }
}
}

```

The modified **mergesort** method now opens two **RandomAccessFile** objects on the two files and they are both opened in *read/write* mode because they will both be updated during the *MergeSort* algorithm. The **mergesort** method is invoked by requesting to sort the contents of the copied file into the original file. Once done, the copied file can be deleted.

The `mergesort(RandomAccessFile A, RandomAccessFile result, long start, long end)` method performs the recursive *MergeSort* of the given range **[start, end)** of the underlying files. These parameters are both of type **long** to enable this method to sort files that can be several gigabytes in size. For this lesson, the files will only be several megabytes in size; feel free to generate files of this size on your home computer!

The structure of this method follows the earlier examples.

Go back and execute **BinaryIntegerFile** and the output will appear properly (though your random numbers will be different):

Output from BinaryIntegerFile

```

First five sorted numbers
1
3
4
6
6
Last five sorted numbers
4091
4091
4093
4093
4095

```

Refresh the files in your workspace; the temporary file used during the sort has been deleted. Let's take a closer look at this code. First, let's inspect the base cases of the recursion:

OBSERVE: mergesort base cases for recursion

```

static void mergesort(RandomAccessFile A, RandomAccessFile result,
                      long start, long end) throws IOException {

    if (end - start < 8) {
        return;
    }

    if (end - start == 8) {
        result.seek(end-8);
        int left = result.readInt();
        int right = result.readInt();
        if (left > right) {
            result.seek(end-8);
            result.writeInt(right);
            result.writeInt(left);
        }
        return;
    }
}

```

Recall that integers are stored using 4 bytes. The offsets **start** and **end** are index locations within the **RandomAccessFile**; in addition, **start** and **end** must be evenly divisible by 4. The condition **end - start < 8** determines when the subrange **[start, end)** contains zero or one element; when this occurs, no sorting needs to take place and the method can simply return.

The second base case needs to swap the two neighboring integers in **result** if they are out of order. When **end - start == 8** you know that **[start, end)** contains two elements exactly. The above code uses the **seek**

method to find that location in the file for the first of these two integers. It then reads in two integers sequentially from the 8 bytes stored at that position in that file. If these two numbers are out of order, it goes back to the beginning of that range in the file and writes the two integers in the proper order.

The final case demonstrates how to merge the two sub-ranges together. It starts with a little mathematical optimization. Mergesort must divide the range into two parts, but each sub-range must contain a number of bytes that is divisible by four. For example, if the range contained 7 integers for a total of 28 bytes, it might be represented as **[0,28)**. Simply dividing $(0+28)/2$ would give 14, which is not divisible by 4. Instead, divide $(0+28)/8$ (to get 3 using integer division) and then multiply by 4 to get 12, which is roughly half of the range.

OBSERVE: Merging case in MergeSort

```
long mid = (end + start)/8*4;
mergesort(result, A, start, mid);
mergesort(result, A, mid, end);

result.seek(start);
for (long i = start, j = mid, idx=start; idx < end; idx += 4) {
    A.seek(i);
    int Ai = A.readInt();
    int Aj = 0;
    if (j < end) { A.seek(j); Aj = A.readInt(); }
    if (j >= end || (i < mid && Ai < Aj)) {
        result.writeInt(Ai);
        i += 4;
    } else {
        result.writeInt(Aj);
        j += 4;
    }
}
```


The above code recursively **invokes mergesort on the left and right sub-ranges**, after which the file on disk referenced by **A** will contain the two sorted sub-ranges waiting to be merged. The code takes advantage of a nice optimization in that the integers written to **result** will be written sequentially, so it only needs to **seek to the starting location** of that output sequence in **result** before starting the **for** loop. When the code determines the **Ai** and **Aj** values, it must seek the proper file position within **A** and then read the integer encoded there.

Note that, in this **for** loop, the index values *i*, *j*, and *idx* are all incremented by 4 because they reference positions inside the file that contains the 4-byte integer encodings.

The condition $(j \geq \text{end} \parallel (i < \text{mid} \ \&\& \ A_i < A_j))$ takes advantage of "short-circuit" logical evaluation. That is, if $j \geq \text{end}$, the second part of the condition (after the " \parallel ") is not executed. However, if $j < \text{end}$, you can retrieve **Aj** from the file. For this reason, the code first loads up **Aj** if it exists to prepare for the short-circuit conditional.

Never Be Satisfied

Despite the success of the code, it still feels like it takes too long to complete. The problem is likely that as the files get larger, the number of disk accesses begins to dominate the performance of the algorithm. Fortunately there is a "drop-in replacement" for file access based on an operating-system capability known as "Memory Mapped Files." The implementation is found in the **java.nio** package, known as the "new input/output" Java package that contains many high-performance classes.

 In the **sort** package of the **/src** source folder, create a class named **MergeSortFileMapped**. Much of this code will be familiar to you because it follows the exact implementation style of the earlier MergeSort.

CODE TO TYPE: MergeSortFileMapped class

```
package sort;

import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MergeSortFileMapped {

    static void mergesort (File A) throws IOException {
        File copy = File.createTempFile("Mergesort", ".bin");
        MergeSortFile.copyFile(A, copy);

        RandomAccessFile src = new RandomAccessFile(A, "rw");
        RandomAccessFile dest = new RandomAccessFile(copy, "rw");
        FileChannel srcC = src.getChannel();
        FileChannel destC = dest.getChannel();
        MappedByteBuffer srcMap = srcC.map(FileChannel.MapMode.READ_WRITE, 0, src.length());
        MappedByteBuffer destMap = destC.map(FileChannel.MapMode.READ_WRITE, 0, dest.length());

        mergesort (destMap, srcMap, 0, (int) A.length());
        src.close();
        dest.close();
    }

    static void mergesort(MappedByteBuffer A, MappedByteBuffer result,
                          int start, int end) throws IOException {

        if (end - start < 8) {
            return;
        }

        if (end - start == 8) {
            result.position(start);
            int left = result.getInt();
            int right = result.getInt();
            if (left > right) {
                result.position(start);
                result.putInt(right);
                result.putInt(left);
            }
            return;
        }

        int mid = (end + start)/8*4;
        mergesort(result, A, start, mid);
        mergesort(result, A, mid, end);

        result.position(start);
        for (int i = start, j = mid, idx=start; idx < end; idx += 4) {
            int Ai = A.getInt(i);
            int Aj = 0;
            if (j < end) { Aj = A.getInt(j); }
            if (j >= end || (i < mid && Ai < Aj)) {
                result.putInt(Ai);
                i += 4;
            } else {
                result.putInt(Aj);
                j += 4;
            }
        }
    }
}
```

To execute this code instead of the earlier version, modify **BinaryIntegerFile** as shown:

CODE TO TYPE: Modifications to BinaryIntegerFile


```
package sort;

import java.io.*;

public class BinaryIntegerFile {
    public static void main(String[] args) throws IOException {
        int numIntegers = 655364096;
        File f = new File ("IntegerFile.bin");
        DataOutputStream dos = new DataOutputStream(new FileOutputStream(f));
        for (int i = 0; i < numIntegers; i++) {
            dos.writeInt((int) (Math.random()*numIntegers));
        }
        dos.close();

        long now = System.currentTimeMillis();
MergeSortFile.mergesort(f);
        MergeSortFileMapped.mergesort(f);
        System.out.println((System.currentTimeMillis() - now) + " ms.");

        DataInputStream dis = new DataInputStream(new FileInputStream(f));
        System.out.println("First five sorted numbers");
        for (int i = 0; i < 5; i++) {
            System.out.println(dis.readInt());
        }
        dis.skipBytes(4*(numIntegers-10));
        System.out.println("Last five sorted numbers");
        for (int i = 0; i < 5; i++) {
            System.out.println(dis.readInt());
        }
        dis.close();
    }
}
```

 Run it to sort just over 65,000 integer values. The code spends most of its time writing the random numbers to the disk file to prepare for the algorithm. The execution now takes far less time (32 milliseconds instead of 10276 milliseconds). This code executes so much faster because when you're working with data on disk, you need to limit the frequency of disk access to maximize the efficiency of your code. Inside the Java Virtual Machine, the **java.nio** package is integrated with the virtual memory manager of the operating system. Memory-mapped files are loaded into memory one entire page at a time, and each operating system is fine-tuned so these operations execute as efficiently as possible. When you modify information in a memory-mapped file, it will be written out to the file one page at a time; the operating system is responsible for carrying this operation out efficiently as well. Now your program is no longer in charge of reading and writing bytes from a file directly; it updates memory directly, as managed by the **MappedByteBuffer** class. Ultimately this class determines when the updated memory is written to the file.

Let's review this code:

OBSERVE: Using MappedByteBuffer to access file data

```
static void mergesort (File A) throws IOException {
    File copy = File.createTempFile("Mergesort", ".bin");
    MergeSortFile.copyFile(A, copy);

    RandomAccessFile src = new RandomAccessFile(A, "rw");
    RandomAccessFile dest = new RandomAccessFile(copy, "rw");
    FileChannel srcC = src.getChannel();
    FileChannel destC = dest.getChannel();
    MappedByteBuffer srcMap = srcC.map(FileChannel.MapMode.READ_WRITE, 0, src.length());
    MappedByteBuffer destMap = destC.map(FileChannel.MapMode.READ_WRITE, 0, dest.length());

    mergesort (destMap, srcMap, 0, (int) A.length());
    src.close();
    dest.close();
}
```

One unfortunate drawback with using MappedByteBuffer is that on many operating systems (and on Windows in particular) once a file has been mapped, it cannot be deleted from within the Java program. The above code, therefore, **creates a temporary file in the designated default temporary directory** which will eventually be cleaned up by the user. From a **RandomAccessFile**, it is **possible to retrieve its FileChannel descriptor**, which is used to **construct the respective MappedByteBuffer objects**.

Changes to the **mergesort** method are more subtle:

OBSERVE: mergesort revised to use MappedByteBuffer

```
static void mergesort(MappedByteBuffer A, MappedByteBuffer result,
                      int start, int end) throws IOException {

    if (end - start < 8) {
        return;
    }

    if (end - start == 8) {
        result.position(start);
        int left = result.getInt();
        int right = result.getInt();
        if (left > right) {
            result.position(start);
            result.putInt(right);
            result.putInt(left);
        }
        return;
    }

    int mid = (end + start) / 8 * 4;
    mergesort(result, A, start, mid);
    mergesort(result, A, mid, end);

    result.position(start);
    for (int i = start, j = mid, idx = start; idx < end; idx += 4) {
        int Ai = A.getInt(i);
        int Aj = 0;
        if (j < end) { Aj = A.getInt(j); }
        if (j >= end || (i < mid && Ai < Aj)) {
            result.putInt(Ai);
            i += 4;
        } else {
            result.putInt(Aj);
            j += 4;
        }
    }
}
```

start and **end** are int values again. The MappedByteBuffer class only supports integer indexing, which means the files to be sorted cannot be greater than 2^{32} bytes in size (roughly 4 gigabytes).

Let's review the base cases of the recursion:

OBSERVE: MergeSortFileMapped base recursive cases

```
if (end - start < 8) {
    return;
}

if (end - start == 8) {
    result.position(start);
    int left = result.getInt();
    int right = result.getInt();
    if (left > right) {
        result.position(start);
        result.putInt(right);
        result.putInt(left);
    }
    return;
}
```

When asked to sort two elements, the code uses the **getInt** method of the MappedByteBuffer class to retrieve the integer stored at the proper offset of **start**. If the MappedByteBuffer does not have this information in main memory, it will read the information into memory one page at a time. If this memory is updated (using the **putInt** methods) it won't be written to disk until the **MappedByteBuffer** determines that it can be written efficiently. As a programmer, you no longer know whether a **getInt** or **end** method accesses the file system; you can simply program it correctly while leaving MappedByteBuffer responsible for the persistent storage of the information.

OBSERVE: Completing the mergesort

```
int mid = (end + start)/8*4;
mergesort(result, A, start, mid);
mergesort(result, A, mid, end);

result.position(start);
for (int i = start, j = mid, idx=start; idx < end; idx += 4) {
    int Ai = A.getInt(i);
    int Aj = 0;
    if (j < end) { Aj = A.getInt(j); }
    if (j >= end || (i < mid && Ai < Aj)) {
        result.putInt(Ai);
        i += 4;
    } else {
        result.putInt(Aj);
        j += 4;
    }
}
```

Despite the large number of read and write statements that access the **result** and **A** files, the MappedByteBuffer class ensures that these operations act on information stored in main memory. You can't predict when (Java) will write the info to a file, so the MappedByteBuffer class ensures that data is read from memory (instead of the file). It'll be more efficient because it's faster to read from memory than from a file (in this case, almost 300 times faster).

Lessons Learned

Often you can improve the efficiency of an algorithm by storing additional state information. You see this on a small scale in **java.lang.String**, which caches its computed hash value to improve the performance of **hashCode**. Often you can achieve efficient $O(n \log n)$ performance by storing additional $O(n)$ storage information.

To determine the appropriate algorithm to use, be sure to characterize the storage requirements in addition to the run-time performance. In most cases, the additional storage will be $O(n)$, which typically is an acceptable

trade-off to make.

Accessing information on disk is typically *thousands of times slower than accessing information in main memory*. When designing algorithms that access data on disk, you must find ways to reduce the number of individual reads and writes, choosing instead to let the operating system optimize input/output access.

Practice some of the things you learned in this lesson in the project. See you in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Representing Graph Data Structures

Lesson Objectives

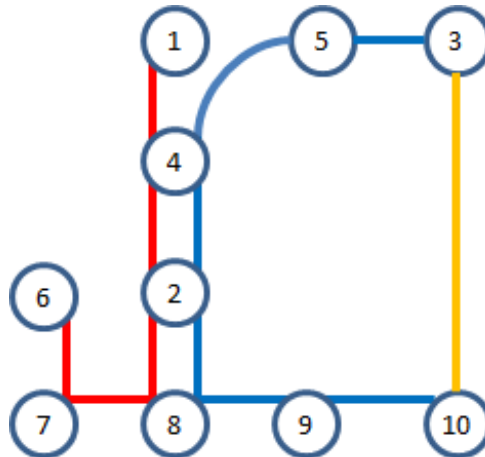
After completing this lesson, you will be able to:

- compute the adjacency matrix for any graph.
- explain the concept of backtracking.
- use a backtracking depth-first search algorithm to traverse a graph.

Representing Graphs

You've already seen how to use **Set** and **List** classes from the Java Collections Framework to represent unordered sets or lists ordered linearly. However, sometimes you need to represent more than a collection of items; you also need to capture *relationships between the items themselves*. One common way to accomplish this is to define a *Graph* construct composed of a set of vertices, **V**, representing a set of items, and a set of edges, **E**, connecting pairs of these vertices, representing the relationships. In this lesson, we'll focus on learning how to represent graphs. Along the way, you'll also learn how to explore a graph by traversing its edges from one vertex to another.

Let's start with an example. Assume you live in a city with a subway system with three subway lines and ten stations as shown:



You would like to determine a route from a given starting station to a destination station. For example, to go from station 4 to 10 you can likely see two distinct paths to take: (4, 5, 3, 10) and (4, 2, 8, 9, 10). If the subway system has dozens of stations, the problem increases in complexity and you might not always be able to "see a path" at a glance. To write a program that solves this problem, you need to develop an algorithm to traverse the subway from a starting station to an ending station. The algorithm needs a data structure that represents the subway system so it can execute efficiently. None of the classes provided by the JDK can be used "as is" to solve this problem, so you have to do this yourself.

Using Adjacency Matrix To Represent Graph

Since you only need to know whether two stations are connected with each other, you could create a two-dimensional array **boolean matrix[][]** in which a given element **matrix[i][j]** is **true** when there is a direct connection between stations **i** and **j**. This array is symmetric so **matrix[i][j]** equals **matrix[j][i]**.

Note

This representation doesn't include part of the structure of a subway system, namely the multiple subway lines that may traverse the same link between two stations. However, for the purposes of this lesson, it's okay.



Create a new Java Project named **Graphs** and assign it to the **Java6_Lessons** working set.



In the **/src** folder of the **Graphs** project, create a **subway** package.



In the **subway** package of the **/src** source folder, create a **SubwayMatrix** class. This class will represent a

graph using an adjacency matrix.

CODE TO TYPE: SubwayMatrix class

```
package subway;

public class SubwayMatrix {
    final int n;
    final boolean matrix[][];

    public SubwayMatrix(int numStations) {
        n = numStations;
        matrix = new boolean[n+1][n+1];
    }
}
```

The **SubwayMatrix** constructor requires the total number of vertices so it can construct the **matrix[][]** two-dimensional array. Because stations (hence vertices) are numbered from **1 .. numVertices**, this array is one number larger than it needs to be; this makes the code easier to read and write.

The example subway system could be represented by auto-initializing the matrix in **SubwayMatrix** as shown:

OBSERVE: potential compiled initialization of matrix for subway problem

```
boolean matrix[][] = new boolean[][] {
/*          1      2      3      4      5      6      7      8      9
10 */
    {false, false, false, false, false, false, false, false, false, false},
/* 1 */ {false, false, false, false, true,  false, false, false, false, false},
/* 2 */ {false, false, false, false, true,  false, false, false, true,  false},
/* 3 */ {false, false, false, false, false, true,  false, false, false, false},
/* 4 */ {false, true,  true,  false, false, true,  false, false, false, false},
/* 5 */ {false, false, false, true,  true,  false, false, false, false, false},
/* 6 */ {false, false, false, false, false, false, false, true,  false, false},
/* 7 */ {false, false, false, false, false, false, true,  false, true,  false},
/* 8 */ {false, false, true,  false, false, false, false, true,  false, true},
/* 9 */ {false, false, false, false, false, false, false, false, true,  false},
/* 10 */ {false, false, false, true,  false, false, false, false, false, true},
    false};
}
```

Instead of defining the subway system in this way, add the method below to the end of **SubwayMatrix**, which will allow you to update *matrix* dynamically, given an array of **int** values in sequence:

CODE TO STYLE: Method to dynamically add stations in a line

```
public void addLine(int[] stations) {
    for (int i = 1; i < stations.length; i++) {
        matrix[stations[i-1]][stations[i]] = true;
        matrix[stations[i]][stations[i-1]] = true;
    }
}
```

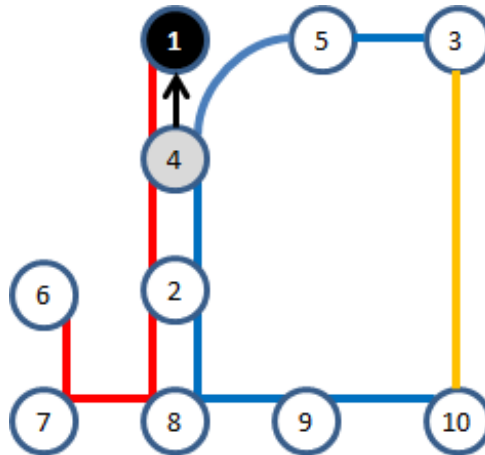
We prefer this approach because you can construct subway lines dynamically without compilation.

Searching a Graph

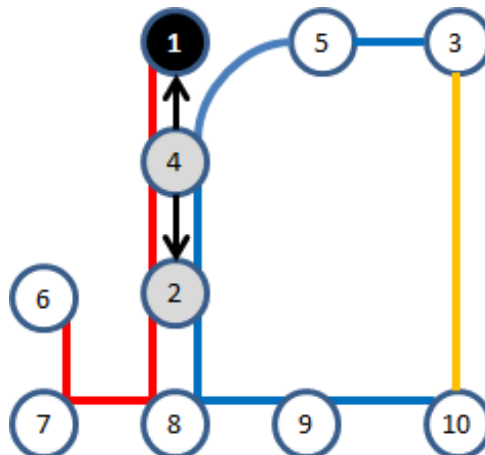
To compute a path in a graph from any vertex X to another vertex Y, make these assumptions:

- The graph is connected; that is, it is possible to travel from any vertex to any other vertex by following the edges in the graph.
- The path must not visit the same vertex twice.

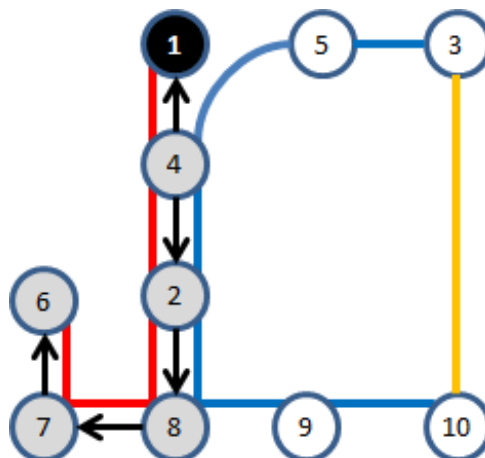
To find a route from station 4 to 10, for example, imagine that you have a copy of the map that you can mark up with a pencil. Start by shading station 4 in gray, and then consider traveling next to one of its unvisited neighbors, such as station 1. Draw an arrow connecting stations 4 and 1. Once you see that station 1 has no neighboring station that you haven't visited, color station 1 in black to indicate that there is no need to consider that station again. Now, like encountering a dead end in a maze, you have to "backtrack" to the previous station 4 to see if another route is possible:



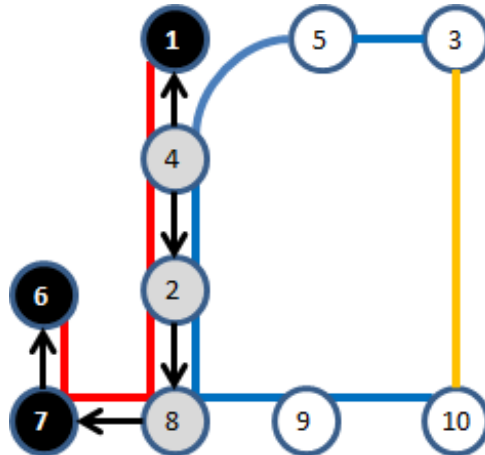
Continue the search by moving on from station 4 to station 2, shading station 2 in gray:



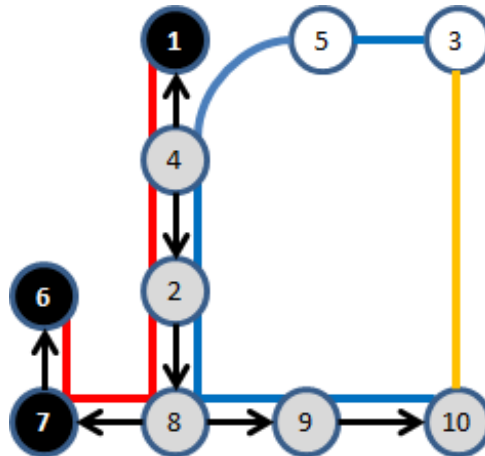
Repeat this procedure with stations 8, 7, and then 6:



Station 6 is a dead end because there are no neighboring stations that you have not already visited, so you can color station 6 black and backtrack to station 7. You get the same result at station 7, so color 7 black and backtrack to station 8.



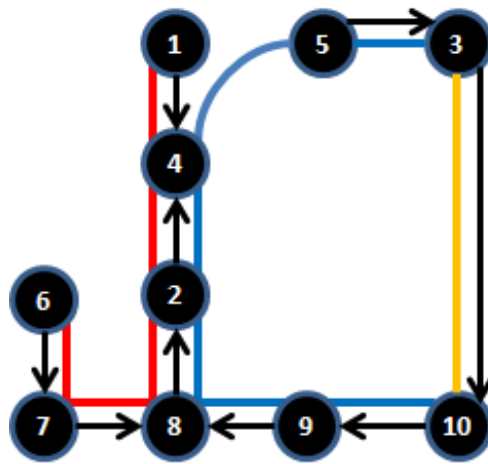
Observe that station 8 still has an unvisited neighbor (station 9), so head in that direction and eventually you will reach station 10, your destination:



In the above graphic, you can see that you have three different station colors:

1. Black vertices have been visited and are fully processed.
2. Gray vertices have been visited, but they may have *an unvisited neighbor*.
3. White vertices have not been visited at all yet.

Instead of stopping when you reach the destination vertex (at station 10), it's just to let the algorithm explore the entire graph such that, when it's finished, all vertices are colored black. You need to make one more enhancement since this algorithm is trying to find a path from the designated start vertex (station 4) to a destination vertex (station 10). In the images above, you connected stations with arrows in the direction of the search. However, if instead you "flip" the arrows so they record *where the search came from*, you can recreate the path from the start vertex to any other vertex in the graph by following the arrows in reverse:



You can reconstruct the path from the start vertex to any destination vertex quickly by starting at a destination vertex and following the black "previous" arrows all the way back to the start vertex. The computed path here is **4, 2, 8, 9, 10**. This algorithm is not designed to compute the shortest path between two vertices. For example, although stations 4 and 5 are directly connected by the blue subway line, the computed path is **4, 2, 8, 9, 10, 3, 5**.

This brief example highlights a *Depth-First Search* over a graph. When faced with that decision, try *visiting some vertex that you haven't yet visited*; when you reach a dead end, *backtrack to the previous vertex* to see if you missed a route to an unvisited vertex. Continue this approach until *all vertices are visited*.

It's time to apply these concepts to your program. Modify **SubwayMatrix** as shown:

CODE TO TYPE: SubwayMatrix class

```
package subway;

public class SubwayMatrix {
    final static int White = 0;
    final static int Gray = 1;
    final static int Black = 2;

    final int n;
    final boolean matrix[][];
    int src;
    final int previous[];
    final int color[];

    public SubwayMatrix(int numStations) {
        n = numStations;
        matrix = new boolean[n+1][n+1];
        previous = new int[n+1];
        color = new int[n+1];
        src = 0;
    }

    public void addLine(int[] stations) {
        for (int i = 1; i < stations.length; i++) {
            matrix[stations[i-1]][stations[i]] = true;
            matrix[stations[i]][stations[i-1]] = true;
        }
    }
}
```

The **SubwayMatrix** constructor requires the total number of vertices so it can construct the **previous** and **color** arrays, as well as the **matrix[][]** two-dimensional array. Because stations (hence vertices) are numbered from **1 .. numVertices** (which makes the code easier to read and write), these arrays are all one size larger than they need to be. SubwayMatrix also stores the source vertices, **src**, from which the desired search is made. This is important because this algorithm ultimately determines the path between the source vertex, **src**, and every other vertex to which it is connected in the graph. Initially we see, **src=0**, which means that the algorithm has not yet executed.

To implement Depth-First Search over a graph, you need to know about recursion. Instead of trying to solve a problem all at once, recursion breaks a problem into smaller pieces. For example, instead of trying to find the full path, start by visiting the start vertex. To visit a vertex **u** you shade it gray to remember that vertex **u** is no longer unvisited. Then, *recursively visit all neighboring vertices of u*. Once these recursive tasks are done, you shade **u** black to indicate that you are done with the vertex. This approach works because you use the color of the vertices to record your progress. When visiting a neighbor **v** of **u**, be sure to record that **previous[v]=u** so you can reconstruct the path from the source vertex to any other connected vertex in the graph. Note that in a connected graph, after completing the search, only the starting vertex has no computed **previous** vertex.

The following pseudocode describes the Depth-First Search algorithm:

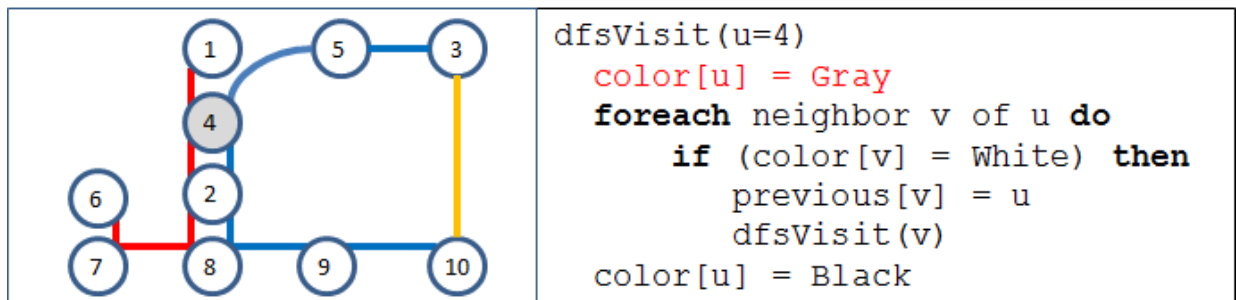
OBSERVE: pseudocode for Depth-First Search

```
dfsSearch(s)
  foreach v in V do
    color[v] = White
  dfsVisit(s)

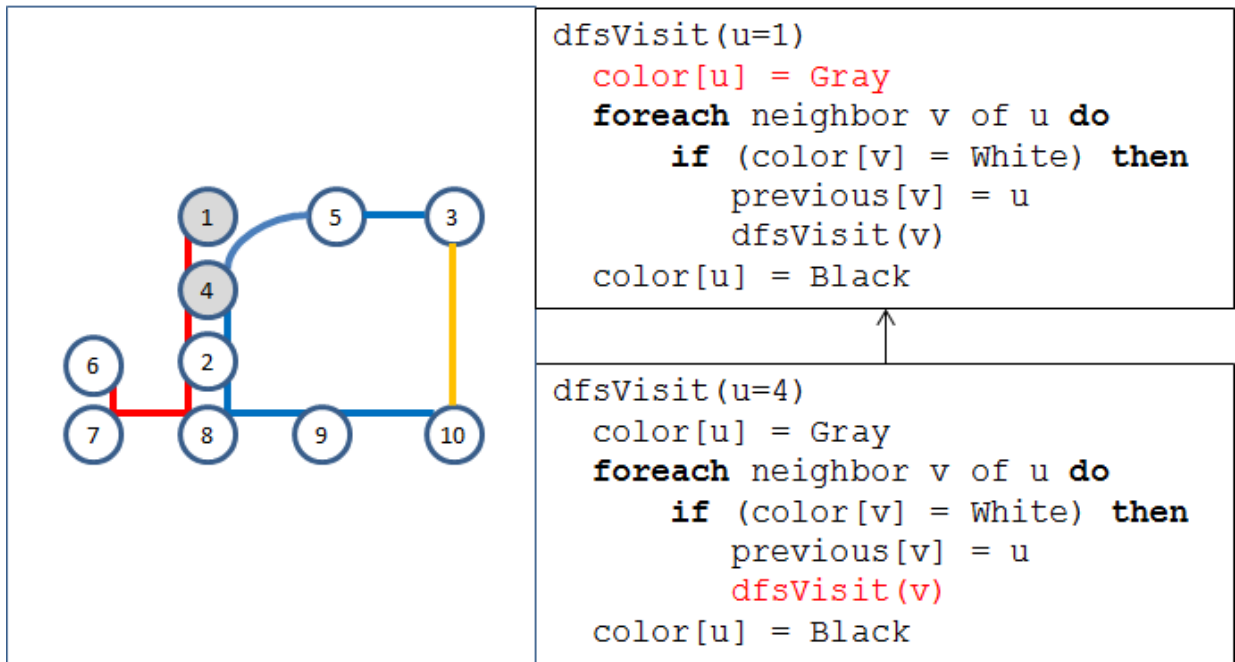
dfsVisit(u)
  color[u] = Gray
  foreach neighbor v of u do
    if (color[v] = White) then
      previous[v] = u
      dfsVisit(v)
  color[u] = Black
```

The algorithm starts by coloring every vertex in the graph **white** before it visits the starting vertex, **s**. The visit function, **dfsVisit(u)**, is a recursive function which invokes **dfsVisit(v)** on each unvisited neighbor **v** of **u**.

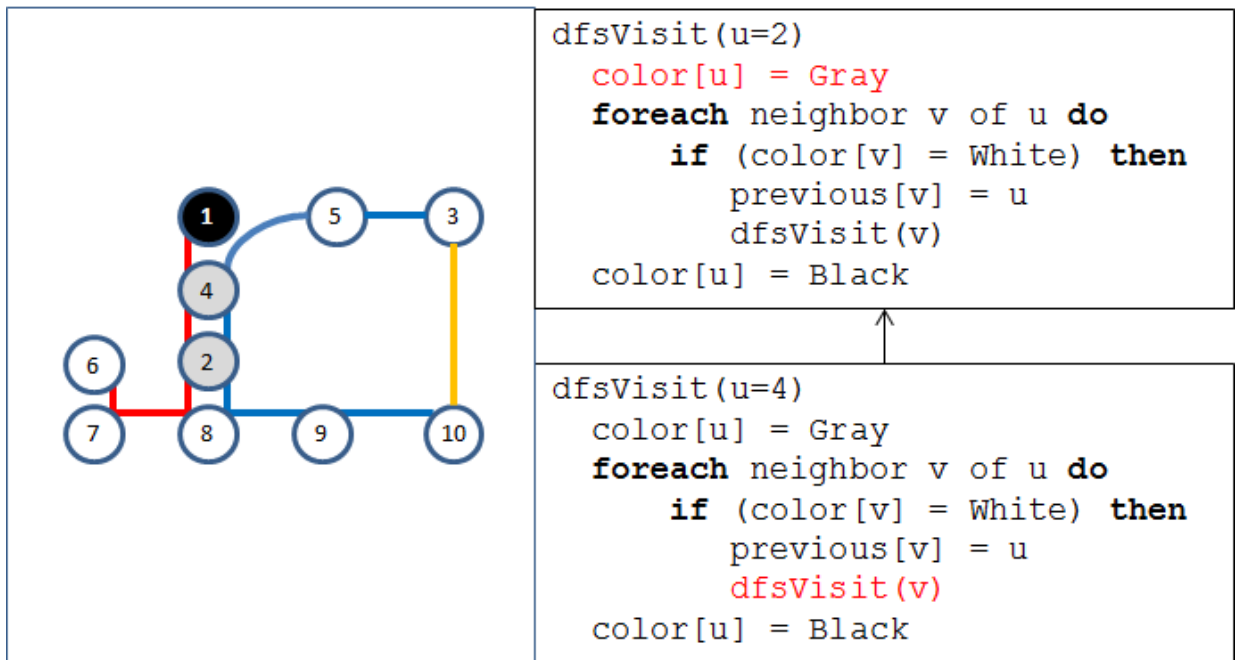
As with previous lessons, it is worth "stepping through" the execution to make sure that it will work properly. In doing so, you will see exactly how recursion allows you to backtrack in your search. Let's start by graphically representing the state of the algorithm and its progress through the pseudocode when **dfsVisit(4)** is called. Each executing pseudocode statement is shown in red on the right.



Once vertex 4 is colored gray, the **foreach** loop processes each of its neighbors; let's start with vertex 1. Since its color is **White**, set **previous[1]=4** and recursively call **dfsVisit(1)**. When this call returns, the **foreach** loop will continue where it left off, and then process the other two neighbors of 4 (namely vertex 2 and 5). In other words, the algorithm will backtrack to vertex 4. This graphic shows the "call stack" and the second invocation of **dfsVisit()**:



The second **dfsVisit(1)** function first colors vertex 1 gray. It then tries to find a neighbor of vertex 1 that is white (indicating that it remains unvisited). Since there are no unvisited neighbors of vertex 1, the function colors vertex 1 **black** and then returns. This is the key backtracking step—going back to an earlier point in the computation. The call stack shows that **dfsVisit(4)** is still waiting for **dfsVisit(1)** to complete so it can move on to the other neighbors of 4. Assuming that vertex 2 is visited next (after vertex 1), the next recursive call (and corresponding graph state) looks like this:



You can continue this exercise as long as you want, ultimately producing the final graphic described earlier where every vertex is colored black. With this pseudocode in hand, you're ready to begin programming.

Modify **SubwayMatrix** as shown to implement Depth-First Search over a graph represented using an adjacency matrix representation:

CODE TO TYPE: Modifications to SubwayMatrix

```
package subway;

import java.util.*;

public class SubwayMatrix {
    final static int White = 0;
    final static int Gray = 1;
    final static int Black = 2;

    final int n;
    final boolean matrix[][];
    int src;
    final int previous[];
    final int color[];

    public SubwayMatrix(int numStations) {
        n = numStations;
        matrix = new boolean[n+1][n+1];
        previous = new int[n+1];
        color = new int[n+1];
        src = 0;
    }

    public void dfsSearch(int s) {
        for (int v = 1; v <= n; v++) {
            color[v] = White;
            previous[v] = 0;
        }

        dfsVisit(s);
        src = s;
    }

    void dfsVisit(int u) {
        color[u] = Gray;

        for (int v = 1; v <= n; v++) {
            if (matrix[u][v] && color[v] == White) {
                previous[v] = u;
                dfsVisit (v);
            }
        }

        color[u] = Black;
    }

    public void addLine(int[] stations) {
        for (int i = 1; i < stations.length; i++) {
            matrix[stations[i-1]][stations[i]] = true;
            matrix[stations[i]][stations[i-1]] = true;
        }
    }
}
```

This code follows the pseudocode fairly faithfully. Let's investigate more closely.

OBSERVE: Initializing and executing search

```
public void dfsSearch(int s) {  
    for (int v = 1; v <= n; v++) {  
        color[v] = White;  
        previous[v] = 0;  
    }  
  
    dfsVisit(s);  
    src = s;  
}
```

The **dfsSearch(int s)** method first initializes the algorithm's state by **resetting the color of each vertex white and clearing the previous links**. Using an array-based storage of the graph allows you to write a simple **for** loop to iterate over all vertices in the graph. Recall that one of the fundamental tasks of a *depth first search* algorithm is to identify the neighbors for a vertex u . With an array-based storage, you only need to use a **for (int v = 1; v <= n; v++)** loop to locate the **true matrix[u][v]** entries for different v values. Remember that the vertices are numbered from 1 to n . Once the search is complete, it **sets the src variable to record the source vertex used for the search**. Now let's investigate the recursive **dfsVisit(int u)** method:

OBSERVE: Recursive dfsVisit method

```
void dfsVisit(int u) {  
    color[u] = Gray;  
  
    for (int v = 1; v <= n; v++) {  
        if (matrix[u][v] && color[v] == White) {  
            previous[v] = u;  
            dfsVisit (v);  
        }  
    }  
  
    color[u] = Black;  
}
```

dfsVisit(int u) must first **color vertex u gray** to signal that the vertex is no longer unvisited. Remember that the algorithm recursively visits all unvisited neighbors of this vertex. With the array-based implementation, you only need to **iterate through all possible vertices, v**, to see if **matrix[u][v] is non-zero (which means there is an edge between u and v) and that color[v] is White** (which means vertex v has not yet been visited). As its final act, **dfsVisit(u)** **colors u black** to signal that vertex u is fully processed. With recursion, you just have to trust that it will visit all vertices required. At this point, you could insert a mathematical proof to show that this algorithm will terminate; instead, let me convince you not to by sharing two observations. First, **dfsVisit** starts by coloring a vertex gray. Second, **dfsVisit** recursively calls **dfsVisit** only on vertices that are colored white. If you put these two observations together, **dfsVisit** will never be called twice on the same vertex. Since there is a fixed number of vertices in the graph, eventually **dfsVisit** will run out of unvisited vertices to process.

To complete the implementation, add a **path(d)** method that returns a **List** of integers representing the stations between the original source vertex and the given destination vertex, **d**. This method traverses the **previous** links and *prepends each vertex identifier* to ensure proper ordering. Add this method to the end of **SubwayMatrix**:

CODE TO TYPE: Add path() method to SubwayMatrix

```
public List<Integer> path (int d) {  
    LinkedList<Integer> path = new LinkedList<Integer>();  
    if (src != 0 && src != d) {  
        while (d != 0) {  
            path.add(0, d);  
            d = previous[d];  
        }  
    }  
  
    return path;  
}
```

To validate the above code, write some performance tests:



Create a new **performance** source folder:



Create a **subway** package in the **/performance** source folder:



In the **/performance** folder **subway** package, create a **Demonstrate** class as shown:

CODE TO TYPE: Demonstrate class

```
package subway;  
  
public class Demonstrate {  
    public static void main(String[] args) {  
        SubwayMatrix sm = new SubwayMatrix(10);  
        sm.addLine(new int[]{1, 4, 2, 8, 7, 6});  
        sm.addLine(new int[]{3, 5, 4, 2, 8, 9, 10});  
        sm.addLine(new int[]{3, 10});  
  
        sm.dfsSearch(4);  
  
        for (int i = 1; i <= 10; i++) {  
            System.out.println("4-" + i + " : " + sm.path(i));  
        }  
    }  
}
```



Save and run it. This code creates the subway described earlier and prints out the path one would follow from station 4 to all other stations in the subway system:

OBSERVE: Sample Output From Demonstrate

```
4-1 : [4, 1]  
4-2 : [4, 2]  
4-3 : [4, 2, 8, 9, 10, 3]  
4-4 : []  
4-5 : [4, 2, 8, 9, 10, 3, 5]  
4-6 : [4, 2, 8, 7, 6]  
4-7 : [4, 2, 8, 7]  
4-8 : [4, 2, 8]  
4-9 : [4, 2, 8, 9]  
4-10 : [4, 2, 8, 9, 10]
```

You can verify that all of these are valid paths in the subway system; this code even handles the case where the source and destination stations are the same by producing the empty path.

Practical Application

Let's put the Depth-First Search Algorithm to use on a related problem: generating a rectangular grid maze. It's a related problem because you can frame the problem starting with a rectangular grid maze with every interior wall present. It's not much of a maze though since you can't move through it. Now start with a cell on

the topmost row of the maze. If you can randomly move in one of the (potentially four) valid directions, either horizontally or vertically, to a cell *that has not yet been visited*, then do so and remove the wall in between. Repeat this process until all cells have been visited.



In the `/src` source folder **subway** package, create a **MazeApplet** class as shown:

CODE TO TYPE: MazeApplet class

```
package maze;

import javax.swing.*;
import java.awt.*;
import java.util.*;

public class MazeApplet extends JApplet {
    int size    = 10, offset = 50;
    int width   = 500, height = 500;

    final static int White = 0;
    final static int Gray  = 1;
    final static int Black = 2;

    int color[][];
    LinkedList<Point> neighbors[][];
    boolean hasEastWall[][];
    boolean hasSouthWall[][];

    void clearWall(int fromR, int fromC, int toR, int toC) {
        if (fromC == toC) {
            hasSouthWall[Math.min(fromR, toR)][fromC] = false;
        } else {
            hasEastWall[fromR][Math.min(fromC, toC)] = false;
        }
    }

    public MazeApplet() {
        hasEastWall = new boolean[height/size][width/size];
        hasSouthWall = new boolean[height/size][width/size];
        color       = new int[height/size][width/size];
        neighbors    = new LinkedList[height/size][width/size];

        for (int r = 0; r < height/size; r++) {
            for (int c = 0; c < width/size; c++) {
                hasEastWall[r][c] = true;
                hasSouthWall[r][c] = true;
                neighbors[r][c] = new LinkedList<Point>();

                if (r != 0) { neighbors[r][c].add(new Point(r-1, c)); }
                if (r != height/size-1) { neighbors[r][c].add(new Point(r+1, c)); }
                if (c != 0) { neighbors[r][c].add(new Point(r, c-1)); }
                if (c != width/size-1) { neighbors[r][c].add(new Point(r, c+1)); }

                Collections.shuffle(neighbors[r][c]);
            }
        }

        dfsVisit(0,width/size/2);
        hasSouthWall[height/size-1][width/size/2] = false;
    }

    void dfsVisit(int r, int c) {
        color[r][c] = Gray;
        while (!neighbors[r][c].isEmpty()) {
            Point cell = neighbors[r][c].remove();
            if (color[cell.x][cell.y] == White) {
                clearWall(r,c, cell.x,cell.y);
                dfsVisit(cell.x, cell.y);
            }
        }
        color[r][c] = Black;
    }

    public void paint(Graphics g) {
        g.drawLine(offset, offset, offset, offset+(height/size)*size);
    }
}
```

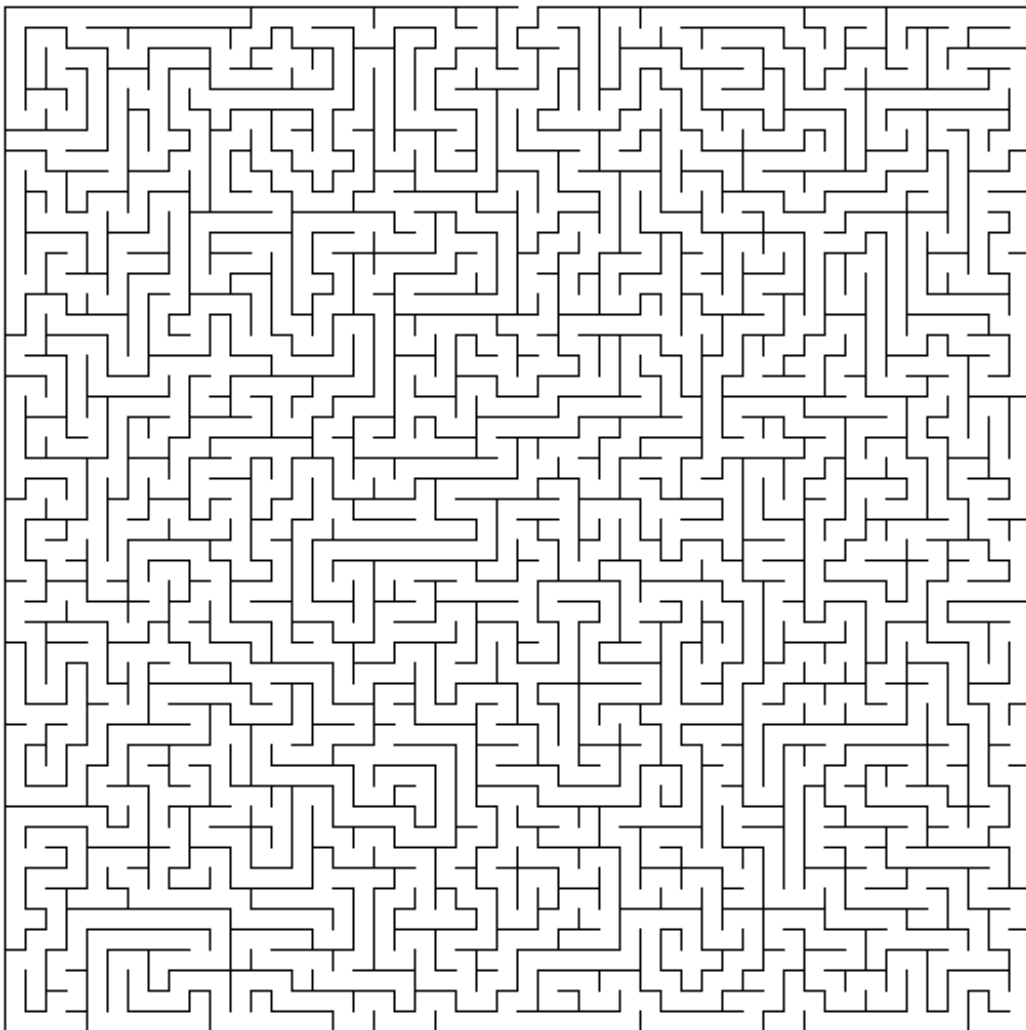
```

        g.drawLine(offset, offset, offset + (width/size/2)*size, offset);
        g.drawLine(offset + size*(1+(width/size)/2), offset, offset+(width/size)*size, offset);

        for (int r = 0; r < height/size; r += 1) {
            for (int c = 0; c < width/size; c += 1) {
                if (hasSouthWall[r][c]) {
                    g.drawLine (offset+c*size, offset + (r+1)*size, offset+(c+1)*size, offset + (r+1)*size);
                }
                if (hasEastWall[r][c]) {
                    g.drawLine (offset+(c+1)*size, offset + r*size, offset+(c+1)*size, offset + (r+1)*size);
                }
            }
        }
    }
}

```

- Save and run this applet and you'll see a window like this:



Let's take a closer look at the code. It has the skeletal structure of Depth-First Search:

OBSERVE: dfsVisit(int r, int c) method

```

void dfsVisit(int r, int c) {
    color[r][c] = Gray;

    while (!neighbors[r][c].isEmpty()) {
        Cell cell = neighbors[r][c].remove();
        if (color[cell.row][cell.col] == White) {
            clearWall(r,c, cell.row,cell.col);
            dfsVisit(cell.row, cell.col);
        }
    }
    color[r][c] = Black;
}

```

Here the **color** array is two-dimensional because each cell is identified by a row and a column. The code uses a **java.awt.Point** class to record a given cell position. First it marks the designated cell as Gray and then, **as long as there is an unvisited neighbor remaining for that cell**, it **clears the wall** between the cell **r,c** and the neighbor **cell.x, cell.y**, before **recursively visiting that cell**. Once all recursions have completed, it sets **color[r][c]** to **Black** because it has completed all processing for that cell.

OBSERVE: clearWall method

```

void clearWall(int fromR, int fromC, int toR, int toC) {
    if (fromC == toC) {
        hasSouthWall[Math.min(fromR, toR)][fromC] = false;
    } else {
        hasEastWall[fromR][Math.min(fromC, toC)] = false;
    }
}

```

The maze contains two arrays, **hasSouthWall** and **hasEastWall**, that determine whether there is a wall at a given row and column. There is no need to worry about north or west walls, because the maze is symmetric (meaning if you can get from cell *u* to *v*, you can get from *v* back to *u*). Using **Math.min**, the above code clears the south or east walls as required. Drawing takes place like this:

OBSERVE: paint(Graphics) method

```

public void paint(Graphics g) {
    g.drawLine(offset, offset, offset, offset+(height/size)*size);
    g.drawLine(offset, offset, offset + (width/size/2)*size, offset);
    g.drawLine(offset + size*(1+(width/size)/2), offset, offset+(width/size)*size, offset);

    for (int r = 0; r < height/size; r += 1) {
        for (int c = 0; c < width/size; c += 1) {
            if (hasSouthWall[r][c]) {
                g.drawLine (offset+c*size, offset + (r+1)*size, offset+(c+1)*size, offset + (r+1)*size);
            }
            if (hasEastWall[r][c]) {
                g.drawLine (offset+(c+1)*size, offset + r*size, offset+(c+1)*size, offset + (r+1)*size);
            }
        }
    }
}

```

The **first g.drawLine invocation** draws the vertical line representing the "western" vertical line of the maze. The **next two invocations of g.drawLine** leave a space at the top of the maze where the start cell exists—exactly half-way through the first row of the maze. The nested **for** loops iterate over all possible cells in the maze and draw the southern and/or eastern walls for those cells if necessary.

OBSERVE: Create Maze

```
public MazeApplet() {
    hasEastWall = new boolean[height/size][width/size];
    hasSouthWall = new boolean[height/size][width/size];
    color = new int[height/size][width/size];
    neighbors = new LinkedList[height/size][width/size];

    for (int r = 0; r < height/size; r++) {
        for (int c = 0; c < width/size; c++) {
            hasEastWall[r][c] = true;
            hasSouthWall[r][c] = true;
            neighbors[r][c] = new LinkedList<Point>();

            if (r != 0) { neighbors[r][c].add(new Point(r-1, c)); }
            if (r != height/size-1) { neighbors[r][c].add(new Point(r+1, c)); }
            if (c != 0) { neighbors[r][c].add(new Point(r, c-1)); }
            if (c != width/size-1) { neighbors[r][c].add(new Point(r, c+1)); }

            Collections.shuffle(neighbors[r][c]);
        }
    }

    dfsVisit(0,width/size/2);
    hasSouthWall[height/size-1][width/size/2] = false;
}
```

The **MazeApplet** constructor creates the **color** storage array used for Depth-First Search. It also creates the arrays for whether **hasSouthWall** and **hasEastWall** exist. Finally, each cell has a number of **neighbors**—between 2 and 4, depending on where that cell exists in the maze. There is a two-dimensional array, **neighbors**, where each element is a **LinkedList** of **Cell** objects. The **nested for loop instantiates the list of all neighbors for each cell** and then uses **Collections.shuffle** to ensure that when **dfsVisit(0,0)** executes, it will search through the maze randomly.

The search starts in the middle of the first row, at cell $(0, \text{width}/\text{size}/2)$. The end point of the maze is identified by removing the south wall of the cell in the middle of the final row.

Lessons Learned

- **Two-dimensional boolean matrices can capture relationships between n elements.** A simple graph is composed of unique edges between any two elements in a set. The range of the matrix represents the vertices, and each value in the matrix is a boolean that represents the existence of an edge between two vertices in the graph.
- **A matrix of complex types can capture metadata about the edges.** Instead of simply recording the existence of an edge, the value in *matrix[u][v]* can contain information about the relationship, including real-world properties such as distance or cost.
- **Depth-First Search is blind and needs to know the target destination so it knows when it is done.** Instead of trying to conduct an intelligent search, Depth-First Search tries each available choice, relying on recursion and backtracking to ensure that the entire space will be searched in pursuit of the goal.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Graph Adjacency List and Shortest Path Algorithms

Lesson Objectives

After completing this lesson, you will be able to:

- represent graphs using Adjacency Lists.
 - explain how *Breadth-First Search* uses a Queue to search a graph.
-

Searching For Optimal Paths


In the last lesson, you applied a *Depth-First Search* algorithm to traverse a graph. However, Depth-First Search will not help you compute the **shortest** path between two vertices. In this lesson, we'll learn how to compute the path with the fewest number of edge traversals between a given source and destination vertex. In tackling this problem, you'll also revise the way that graphs are stored.

Representing Graph By Adjacency List

The **SubwayMatrix** class you designed in the prior lesson represents a graph using a two-dimensional array known as the *adjacency matrix*. An alternate representation for graphs is an *adjacency list*, which is a more efficient data structure to use for sparse graphs. A graph with **n** vertices may potential have $n*(n-1)/2$ edges (which demonstrates quadratic growth), but a sparse graph has much fewer edges. For example, suppose you want to use *Breadth-First Search* to determine the fewest number of subway stations to visit in the New York City subway system given a source and destination station. Start by constructing a graph where the vertices represent the 421 subway stations (468 if you individually count the subway stations that belong to one of the 32 station complexes). Theoretically, a graph with 421 vertices could have up to $421*420/2$, or 88,410, individual tracks connecting pairs of stations. The actual number of station pairings will be much smaller, given the physical reality of subway construction.

Now you'll develop an adjacency list implementation that stores a collection of neighboring vertices for each vertex. As graphs become larger (and sparser) this form of representation will decrease the storage requirements of a graph significantly. Also, instead of being forced to use a **for** loop to iterate over all possible edges that might exist, code using an adjacency list will only iterate over the existing known neighbors. The performance benefits will be negligible for small graphs, but when you tackle larger graphs, you'll see the improved performance.

We'll continue working in the **Graphs** project for this lesson.

 In the **/src** source folder **subway** package, create a **SubwayList** class. This class borrows much of the implementation from **SubwayMatrix**:

CODE TO TYPE: SubwayList

```
package subway;

import java.util.*;

public class SubwayList {
    final static int White = 0;
    final static int Gray = 1;
    final static int Black = 2;

    final int n;
    final Set<Integer>[] neighbors;
    int src;
    final int previous[];
    final int color[];

    public SubwayList(int numStations) {
        n = numStations;
        neighbors = new TreeSet[n+1];
        for (int i = 1; i <= n; i++) {
            neighbors[i] = new TreeSet<Integer>();
        }

        previous = new int[n+1];
        color = new int[n+1];
        src = 0;
    }

    public void addLine(int[] stations) {
        for (int i = 0; i < stations.length-1; i++) {
            neighbors[stations[i]].add(stations[i+1]);
            neighbors[stations[i+1]].add(stations[i]);
        }
    }

    public ArrayList<Integer> path (int d) {
        ArrayList<Integer> path = new ArrayList<Integer>();
        if (src != 0 && src != d) {
            while (d != 0) {
                path.add(0, d);
                d = previous[d];
            }
        }
        return path;
    }
}
```

Let's take a closer look at this class:

OBSERVE: SubwayList Structure

```
public class SubwayList {
    final static int White = 0;
    final static int Gray = 1;
    final static int Black = 2;

    final int n;
    final Set<Integer>[] neighbors;
    int src;
    final int previous[];
    final int color[];

    public SubwayList(int numStations) {
        n = numStations;
        neighbors = new TreeSet[n+1];
        for (int i = 1; i <= n; i++) {
            neighbors[i] = new TreeSet<Integer>();
        }

        previous = new int[n+1];
        color = new int[n+1];
        src = 0;
    }

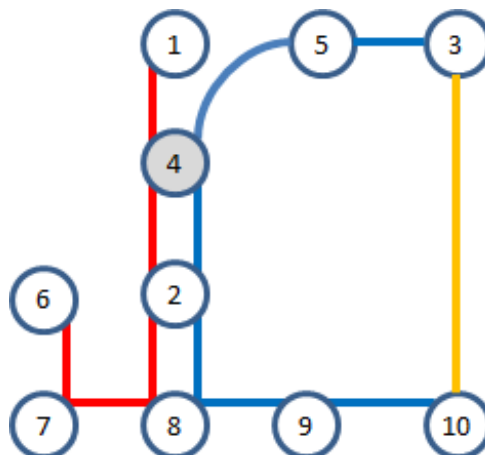
    ...
}
```

Instead of using a two-dimensional array, we use a single array, **neighbors**, to represent the **Set** of neighboring vertices for each vertex in the graph. The index into **neighbors** is the vertex identifier (a number from 1 .. n). Note that each element in the **neighbors** array is a **Set<Integer>**. With this change, the **addLine** method now invokes the **add** method to insert each vertex. You don't need to check whether two vertices are already connected because the **TreeSet** implements set-based semantics, so duplicates are prevented. The same **White**, **Gray**, and **Black** constants are used, in addition to the *color* and *previous* arrays.

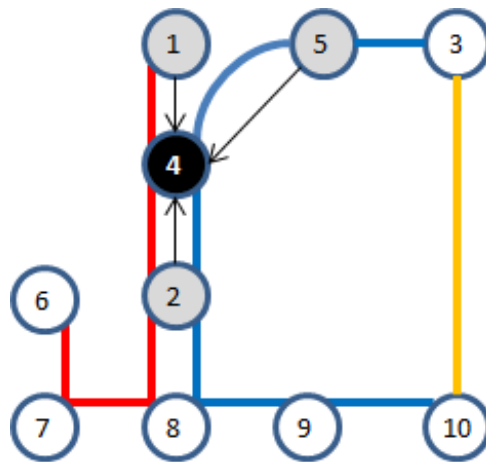
Breadth-First Search

While Depth-First Search computes valid paths between two vertices in a connected graph, there is no guarantee that the computed path is the shortest that exists. You'll need to try another approach. A Breadth-First Search through a graph starts at a source vertex, **s**, then proceeds to visit all vertices that are one edge away from **s**, then vertices no more than two edges away, then vertices no more than three edges away, and so on. The search proceeds methodically from the source vertex, radiating outwards until all vertices in the connected graph are visited.

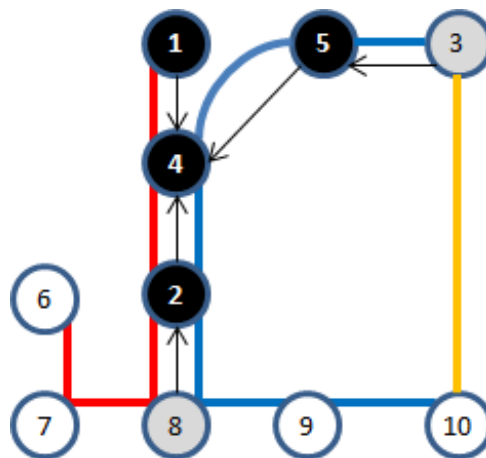
Using the same subway system from the previous lesson, let's compute the shortest path from station 4 to all other stations in the system. Start by coloring vertex 4 Gray:



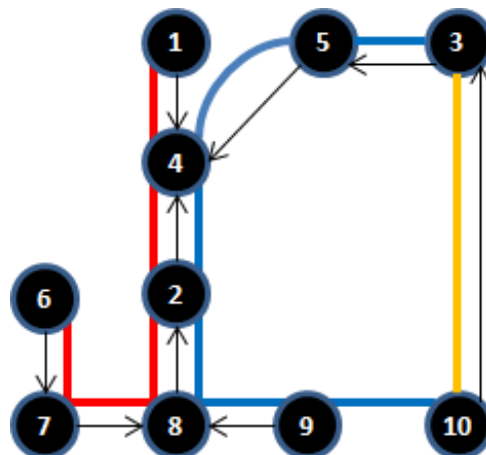
Now three stations are directly connected to station 4, so they are just one edge away. Color station 4 Black (to signal that it's done) and color in Gray stations 1, 2, and 5. Record the previous station in the path (in this case, station 4) using an arrow for each of these stations:



At this point, station 1 is a dead end because it has no unvisited neighbors. However, you can continue to extend the search outwards from stations 2 and 5. Be sure to color stations 2 and 5 Black because they are now fully processed and update previous links for stations 3 and 8:



Continue this process until all vertices are colored Black and all previous links are assigned. Trace a path from any destination station to station 4 (the source station for the search) and you won't be able to find a shorter path than the one computed by Breadth-First Search:



We've reused the concept of coloring vertices developed in the last lesson for Depth-First Search. Specifically,:

- black vertices have been visited and are fully processed.
- gray vertices have been visited but they may have an *unvisited neighbor*.
- white vertices have not been visited yet at all.

The fundamental question for implementing Breadth-First Search is how to keep track of the state of the algorithm as it progresses. Depth-First Search maintains only one "current vertex" as it searches through the

graph, backtracking to overcome dead ends. However, Breadth-First Search needs to keep track of the Gray vertices that it has identified for exploration. It also must make sure to process the vertices in order. In the subway system above, the shortest path from station 4 to station 9 contains three edges (**4, 2, 8, 9**); another longer path exists (**4, 5, 3, 10, 9**).

To enable Breadth-First Search to keep track of the Gray vertices, let's review the behavior of a First-in First-out Queue, a versatile data structure that stores an ordered sequence of items. Using the terminology from the Java Collections Framework, a *Queue* is a Collection that supports this behavior:

- Items are added to the *tail* of a Queue using the **add** operation.
- Items are removed from the *head* of a Queue using the **remove** operation.

Breadth-First Search uses a Queue to maintain all Gray vertices, which represents the "boundary" of the search radiating outwards from the initial source vertex, **s**. While this Queue is not empty, there may still be other unvisited vertices to be processed. This pseudocode describes the Breadth-First Search algorithm:

OBSERVE: pseudocode for Breadth-First Search

```
bfsSearch(s)
  foreach v in V do
    previous[v] = 0
    color[v] = White

  color[s] = Gray
  Q = empty Queue
  add s to Q

  while (Q is not empty) do
    u = remove head of Q
    foreach neighbor v of u do
      if (color[v] = White) then
        previous[v] = u
        color[v] = Gray
        add v to Q
    color[u] = Black
```

In this lesson, we'll complete the Breadth-First Search and Depth-First Search implementations in both the **SubwayMatrix** and **SubwayList** classes.

Modify the existing **SubwayMatrix** implementation described in the previous lesson, to convert this pseudocode to Java:

CODE TO TYPE: Modifications to SubwayMatrix class

```
package subway;

import java.util.*;

public class SubwayMatrix {
    final static int White = 0;
    final static int Gray = 1;
    final static int Black = 2;

    final int n;
    final boolean matrix[][];
    int src;
    final int previous[];
    final int color[];

    public SubwayMatrix(int numStations) {
        n = numStations;
        matrix = new boolean[n+1][n+1];
        previous = new int[n+1];
        color = new int[n+1];
        src = 0;
    }

    public void addLine(int[] stations) {
        for (int i = 1; i < stations.length; i++) {
            matrix[stations[i-1]][stations[i]] = true;
            matrix[stations[i]][stations[i-1]] = true;
        }
    }

    public void dfsSearch(int s) {
        for (int v = 1; v <= n; v++) {
            color[v] = White;
            previous[v] = 0;
        }

        dfsVisit(s);
        src = s;
    }

    void dfsVisit(int u) {
        color[u] = Gray;

        for (int v = 1; v <= n; v++) {
            if (matrix[u][v] && color[v] == White) {
                previous[v] = u;
                dfsVisit (v);
            }
        }

        color[u] = Black;
    }

    public List<Integer> path (int d) {
        LinkedList<Integer> path = new LinkedList<Integer>();
        if (src != 0 && src != d) {
            while (d != 0) {
                path.add(0, d);
                d = previous[d];
            }
        }

        return path;
    }

    public void bfsSearch(int s) {
```



```

    for (int v = 1; v <= n; v++) {
        color[v] = White;
        previous[v] = 0;
    }

    Queue<Integer> q = new LinkedList<Integer>();
    color[s] = Gray;
    q.add(s);

    while (!q.isEmpty()) {
        int u = q.remove();

        for (int v = 1; v <= n; v++) {
            if (matrix[u][v] && color[v] == White) {
                previous[v] = u;
                color[v] = Gray;
                q.add(v);
            }
        }

        color[u] = Black;
    }

    src = s;
}
}

```

Let's review this code in more detail:

OBSERVE: Initializing Breadth-First Search

```

public void bfsSearch(int s) {
    for (int v = 1; v <= n; v++) {
        color[v] = White;
        previous[v] = 0;
    }

    Queue<Integer> q = new LinkedList<Integer>();
    color[s] = Gray;
    q.add(s);

    ...
}

```

The **for loop** iterates over all vertices in the graph to reset their **color** and **previous** values. It then **constructs a Queue of integers starting with s as its initial element**. When analyzing an algorithm, it's helpful to identify some invariants that are always true. From the pseudocode you saw earlier, observe that **any vertex in the Queue is colored Gray**.

There are many classes in the Java Collections Framework that implement the Queue interface; we chose the **LinkedList** class because it implements **add** (to the tail of the queue) and **remove** (from the head of the queue) efficiently. Also, observe a common idiom when using the Collections Framework: referring to the instantiated object, **q**, by its interface **Queue** rather than the instantiating class **LinkedList**:

OBSERVE: Computing Breadth-First Search

```
while (!q.isEmpty()) {
    int u = q.remove();

    for (int v = 1; v <= n; v++) {
        if (matrix[u][v] && color[v] == White) {
            previous[v] = u;
            color[v] = Gray;
            q.add(v);
        }
    }

    color[u] = Black;
}
src = s;
```

The algorithm proceeds by **removing the head element, u, from the Queue** and **adding to the tail those unvisited vertices that are neighbors of u**.

As long as there are vertices in the Queue that need to be processed, the **while loop** will remove the head vertex from the Queue. At some point the **while loop** must terminate because only the unvisited vertices (colored White) are ever considered for addition to the Queue, and there are a finite number of vertices in the graph. Note that this code maintains the invariant that only Gray vertices are added to the Queue. The **add** method properly inserts the vertex *at the tail* to maintain proper ordering of the vertices within the Queue. That is, there is no other Gray or White vertex in the graph that is closer to the source vertex, **s**.

To validate this implementation, write this performance code:

 In the **/performance** source folder **subway** package, create a **Compare** class as shown:

CODE TO TYPE: Compare Class

```
package subway;


import java.util.*;

public class Compare {
    public static void main(String[] args) {
        SubwayMatrix sm = new SubwayMatrix(10);
        sm.addLine(new int[]{1, 4, 2, 8, 7, 6});
        sm.addLine(new int[]{3, 5, 4, 2, 8, 9, 10});
        sm.addLine(new int[]{3, 10});

        sm.dfsSearch(4);
        List dfsPaths[] = new List[11];
        for (int i = 1; i <= 10; i++) {
            dfsPaths[i] = sm.path(i);
        }

        sm.bfsSearch(4);
        List bfsPaths[] = new List[11];
        for (int i = 1; i <= 10; i++) {
            bfsPaths[i] = sm.path(i);
        }

        for (int i = 1; i <= 10; i++) {
            if (bfsPaths[i].size() < dfsPaths[i].size()) {
                System.out.println("4-" + i + " : " + bfsPaths[i] + " (instead of " + dfsPaths[i] + ")");
            }
        }
    }
}
```

 Save and run it. As you can see, this code directly compares Breadth-First Search against Depth-First

Search on the same subway system and prints only the paths that are shorter when computed by Breadth-First Search. The output below is computed and you can verify that it finds three shorter paths in the graph:

OBSERVE: Comparison of Breadth-First and Depth-First Search on Subway System
--

4-3 : [4, 5, 3] (instead of [4, 2, 8, 9, 10, 3]) 4-5 : [4, 5] (instead of [4, 2, 8, 9, 10, 3, 5]) 4-10 : [4, 5, 3, 10] (instead of [4, 2, 8, 9, 10])
--

Now you can complete **SubwayList** by implementing the Breadth-First Search algorithm also:.

CODE TO TYPE: Modifications to SubwayList class

```
package subway;

import java.util.*;

public class SubwayList {
    final static int White = 0;
    final static int Gray = 1;
    final static int Black = 2;

    final int n;
    final Set<Integer>[] neighbors;
    int src;
    final int previous[];
    final int color[];

    public SubwayList(int numStations) {
        n = numStations;
        neighbors = new TreeSet[n+1];
        for (int i = 1; i <= n; i++) {
            neighbors[i] = new TreeSet<Integer>();
        }

        previous = new int[n+1];
        color = new int[n+1];
        src = 0;
    }

    public void addLine(int[] stations) {
        for (int i = 1; i < stations.length; i++) {
            neighbors[stations[i-1]].add(stations[i]);
            neighbors[stations[i]].add(stations[i-1]);
        }
    }

    public ArrayList<Integer> path (int d) {
        ArrayList<Integer> path = new ArrayList<Integer>();
        if (src != 0 && src != d) {
            while (d != 0) {
                path.add(0, d);
                d = previous[d];
            }
        }
        return path;
    }

    public void dfsSearch(int s) {
        for (int v = 1; v <= n; v++) {
            color[v] = White;
            previous[v] = 0;
        }

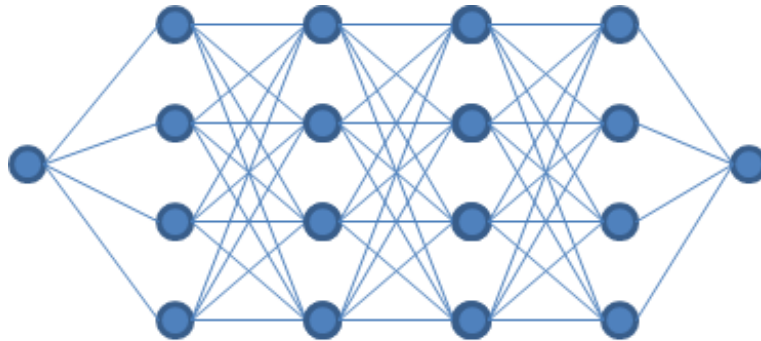
        dfsVisit(s);
        src = s;
    }


    void dfsVisit(int u) {
        color[u] = Gray;
        for (int v : neighbors[u]) {
            if (color[v] == White) {
                previous[v] = u;
                dfsVisit(v);
            }
        }
        color[u] = Black;
    }
}
```

Because you can iterate over just the neighbors of a given vertex, there is don't have to use a **for** loop within **dfsVisit** to check for all possible edges that might exist like you did when the graph was represented as an Adjacency Matrix.

To understand which representation option to choose (Adjacency Matrix or Adjacency List), figure out which types of graphs you'll be processing. In a dense graph, the number of edges can grow proportional to the square of the number of vertices. In a sparse graph, the number of edges grows linearly with the number of vertices.

The following performance code generates stylized graphs (representing subway lines) on which to test these algorithms. Specifically, these graphs have $n=k^2+2$ vertices and k^3-k^2+2k edges. The number of edges is roughly $n^{1.5}$, where n is the number of vertices. The following is the example for $k=4$ which contains $n=18$ vertices and 56 edges (vertex 1 is the leftmost vertex and vertex 18 is the rightmost one):



 In the **/performance** source code folder **subway** package, create a **StylizedDemonstration** class as shown:

CODE TO TYPE: StylizedDemonstration

```
package subway;

public class StylizedDemonstration {
    static int Max = 20;
    static int m = 1000000;
    static SubwayMatrix mat;
    static SubwayList list;
    static int numVertices;


    public static void main(String[] args) {
        System.out.println("n\tMatrix\t\tList");
        for (int k = 2; k <= 64; k *= 2) {
            float totalMatrix = 0, totalList = 0;
            for (int numTrials = 1; numTrials <= Max; numTrials++) {
                generate(k);

                System.gc();
                long now = System.nanoTime();
                mat.dfsSearch(1);
                totalMatrix += (System.nanoTime()-now);

                System.gc();
                now = System.nanoTime();
                list.dfsSearch(1);
                totalList += (System.nanoTime()-now);
            }
            System.out.println(numVertices + "\t" + totalMatrix/Max/m + "\t" + totalList/Max/m);
        }
    }

    public static void generate(int k) {
        int n = k*k;
        numVertices = n+2;
        mat = new SubwayMatrix(n+2);
        list = new SubwayList(n+2);
        int[] pairs;

        for (int i = 2; i <= k+1; i++) {
            pairs = new int[]{1,i};
            list.addLine(pairs);
            mat.addLine(pairs);
        }
        for (int i = n-k+2; i <= n+1; i++) {
            pairs = new int[]{n+2,i};
            list.addLine(pairs);
            mat.addLine(pairs);
        }
        for (int i = 0; i < k-1; i++) {
            for (int j = 0; j < k-1; j++) {
                int u = 2 + i*k + j;
                for (int m = 0; m < k; m++) {
                    pairs = new int[]{u, 2+(i+1)*k+m};
                    list.addLine(pairs);
                    mat.addLine(pairs);
                }
            }
        }
    }
}
```

 Save and run it; you see output similar to this:

OBSERVE: Comparing Adjacency List with Adjacency Matrix		
n	Matrix	List
6	0.0047069504	0.031076651
18	0.0098863	0.05587125
66	0.033456147	0.0704818
258	0.24323966	0.18596876
1026	3.4497294	1.659615
4098	52.31373	26.572315

The Adjacency Matrix implementation initially outperforms the Adjacency List implementation, but the situation changes quickly, and the Adjacency Matrix implementation progresses twice as slowly. Note that the algorithm has not changed, but rather the structural representation of the graph.

Lessons Learned

- **Representation of a data structure impacts the performance for an algorithm.** Even when you have identified the proper algorithm to use, make sure that you are not using a sub-optimal data structure. The Adjacency List is preferred for sparse graphs while Adjacency Matrix is optimal for dense graphs. Only you know which types of graphs that you intend to process, so choose wisely!
- **Stacks support last-in, first-out while Queues support first-in, first-out.** The difference between Depth-First Search and Breadth-First Search can be traced directly to the data structures used to represent the active search. Depth-First Search uses the call stack to store progress, backtracking whenever it hits a dead end; *Breadth-first Search* uses a queue to methodically search a graph.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Priority Queues

Lesson Objectives

After completing this lesson you will be able to:

- write your own heap implementation using array-based storage.
- describe two distinct implementations of priority queues.
- compute the resulting heap structure after a number of insertions and removals.
- compute a Minimum Spanning Tree for a graph using Prim's Algorithm.

Priority Queue Data Structure

A *Queue* is the data structure used when you need First-in, First-out behavior as items are added to, and removed from, a collection. Normally a queue is used to model a sequence of items using the insertion time as the comparator between elements. The *Priority Queue* is a related structure that behaves like a queue, except that the items in the queue all have an associated *priority value* (typically an integer). In a priority queue, each item is added with an associated priority. When removing an element from the priority queue, the item with the smallest priority value is removed first. That is, the most important item in the priority queue is the one with the smallest priority value. Typically, priority values are non-negative, so zero has the highest importance while +Infinity has the lowest priority.

Note

If two or more items have the same lowest priority value, either one may be returned when you request the removal of an item from the priority queue.

To see a priority queue in operation, let's introduce the **Pair** class, which contains an integer key value and its associated integer priority.



Create a new Java Project named **Priority** and assign it to the **Java6_Lessons** working set.



In the **/src** source folder, create a package named **mst**.



In the **mst** package, create a class named **Pair** as shown:

CODE TO TYPE: Pair class

```
package mst;

public class Pair {
    int key;
    int priority;

    public Pair (int k, int p) {
        key = k;
        priority = p;
    }

    public String toString() {
        return "(" + key + ",p=" + priority + ")";
    }
}
```

The **Pair** class associates a priority value with each key value. The class below codifies that *smaller priority values represent more important items in the priority queue*.



In the **mst** package, create a class **PriorityComparator** as shown:

CODE TO TYPE: PriorityQueue class

```
package mst;

import java.util.Comparator;

public class PriorityQueue implements Comparator<Pair> {
    public int compare(Pair first, Pair second) {
        return first.priority - second.priority;
    }
}
```

This **Comparator** class determines how to compare two **Pair** objects in the **PriorityQueue**. **Pair** objects with lower priority values are considered to be more important, so the **compare** method must return zero when the two objects have the same priority and a negative number when the first object's priority value is smaller than the second object's priority value (in other words, when the first object has higher importance).

Now you can create a **PriorityQueue** object to which you add **Pair** objects and from which you retrieve **Pair** objects, in order of their importance:

 In the **mst** package, create a class **SamplePriorityQueue** as shown:

CODE TO TYPE: SamplePriorityQueue class


```
package mst;

import java.util.*;

public class SamplePriorityQueue {
    public static void main(String[] args) {
        PriorityQueue<Pair> pq = new PriorityQueue<Pair>(10, comp);

        pq.add(new Pair(1000, 5));
        pq.add(new Pair(2000, 10));
        pq.add(new Pair(3000, 7));
        pq.add(new Pair(4000, 3));

        while (!pq.isEmpty()) {
            System.out.println(pq.remove());
        }
    }
}
```

 Save and run it; you see this output.

OBSERVE: Output of SamplePriorityQueue

```
(4000,p=3)
(1000,p=5)
(3000,p=7)
(2000,p=10)
```

Your first thought might be that a **PriorityQueue** only sorts its elements by their respective priorities. However, it isn't mandatory to sort queue elements by their respective priority values. You can make your sort more efficient by removing the item with highest priority from the queue. Once again, efficiency will be based on achieving $O(\log n)$ performance on both the **add** and **remove** methods.

In this lesson, you'll learn how to apply priority queues to compute a *Minimum Spanning Tree* (MST) of a graph. Computing the MST is central to many network problems, because it determines the lowest aggregate total for a set of edges that maintains the connected property of a graph. Solving MST is useful for chip design and the telecommunications industry because they are often concerned with computing a connectivity scheme that uses the lowest total length of wire. In this lesson, you'll learn some limitations of the existing **PriorityQueue** implementation as found in the Java Collections Framework, and you'll develop your own priority queue class using a novel data structure, known as a *heap*.

Minimum Spanning Tree

A connected graph allows you to go from any vertex in the graph to any other vertex in the graph over its edges. Given a connected graph $G=(V,E)$, it's possible that the graph will still be connected even if you discard many of its edges. A *Spanning Tree* for a graph is simply a graph $STG=(V,SE)$ where SE is a subset of the original edges in the graph such that the removal of any edge in SE from STG results in a disconnected graph. There are many such spanning trees for a given graph. If each edge in the graph is associated with a positive weight, you might want to find a minimum spanning tree for a graph whose accumulated edge weights is minimum for all possible spanning trees.

To solve this problem efficiently, you can't just generate all possible spanning trees and select the one whose accumulated edge weights is minimum; there are simply too many possible spanning trees. At the same time, you are not trying to find a unique spanning tree; there may be many spanning trees with accumulated edge weights that all match the same minimum value.

Prim's Algorithm is an elegant solution to constructing a minimum spanning tree (MST) for a given graph by using a greedy approach in which each step of the algorithm makes forward progress towards a solution without reversing earlier decisions; That is, no backtracking is necessary.

This pseudocode describes the algorithm:

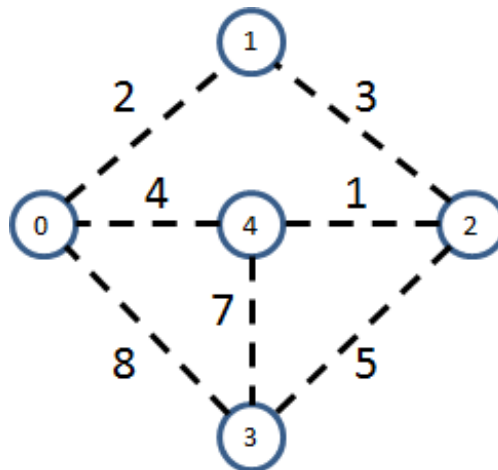
OBSERVE: pseudocode for Prim's Algorithm

```
computeMST(G)
  MST = empty
  S = some vertex in V
  T = V - S

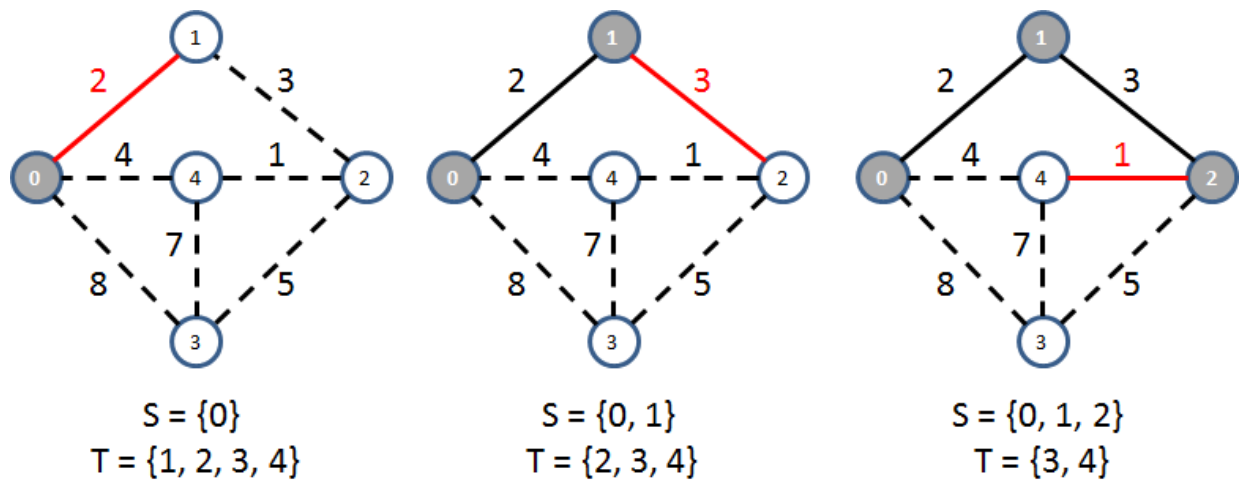
  while (T is not empty) do
    find edge (u,v) with lowest weight such that (u in S) and (v in T)
    add (u,v) to MST
    remove v from T
    add v to S
  return MST
```

At each step through the **while loop**, the algorithm finds the edge with lowest weight that crosses the boundary from the set **S** (representing the vertices in the MST) and **T** (representing the vertices still to be processed). This greedy approach will ensure that the accumulated weights of the MST grows by the smallest possible increment with each step, ultimately resulting in a minimum spanning tree for the graph.

Let's go over this pseudocode on a specific example to make sure it's designed properly. Here is a sample graph for which you will compute a minimum spanning tree:

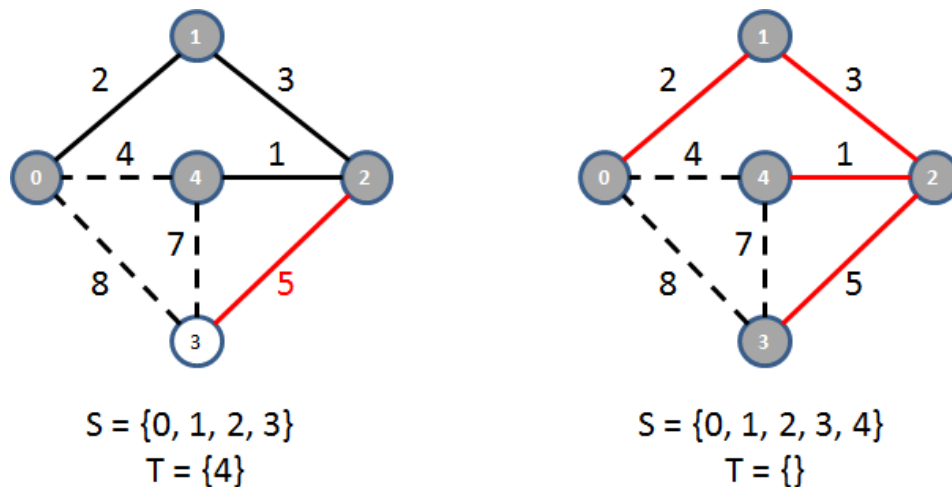


Start with vertex 0 as the initial vertex in set S , which means $T = \{1, 2, 3, 4\}$. The edge $(0,1)$ highlighted in red is the edge between S and T with the lowest edge weight.



The edge (0,1) is added to the MST and S and T are updated accordingly. In the middle graph above, the edge (1,2) is the edge between S and T with the lowest edge weight, so this edge is added to the MST, and S and T are updated. In the third image above, the edge (2,4) is the edge between S and T with the lowest edge weight, so it is added to the MST.

The final step in the algorithm below shows that edge (2,3) is the edge between S and T with the lowest edge weight, so it is added to the MST and the algorithm completes. You won't be able to find another spanning tree for this graph with accumulated weights that is lower than the 11 computed here.



What data structures should you use to implement this algorithm? Well, you can use **Collection** objects to represent sets S and T, but the most costly operation is within the **while** loop where you have to efficiently find the edge with the lowest weight for all edges (u,v) where u belongs to S and v belongs to T. It seems like, as S grows in size, it will be increasingly complicated to compute this edge. If you had to check each edge that exists between S and T, the performance of the algorithm would suffer.

How can a priority queue be used to solve this problem? The MST above is computed by starting at vertex 0, so the three edges being inspected first are those directly connected to vertex 0: the edges to vertices 1, 3, and 4. Note that vertex 2 is not yet "on the search horizon" so the distance from vertex 0 to vertex 2 must be considered to be +Infinity. So, what if you were able to maintain a priority queue that contained all vertices with a computed priority of the current shortest distance from **any vertex in S**? For example, at the start you could insert the vertices 1, 3, and 4 into the priority queue with priorities 2, 8, and 4 respectively; vertex 2 would also be in the priority queue, but its priority would be +Infinity. The priority queue would look like this: **(1, p=2) > (4, p=4) > (3, p=8) > (2, p=INF)**. The ordering in the priority queue is done according to increasing distance, so the first vertex to be removed from the priority queue would be vertex 1. At this point, you could review the neighbors of vertex 1 (in this case vertex 2) and determine to insert (2, p=3) into the priority queue; however, this vertex already exists in the priority queue. Somehow you need to decrease the priority value associated with a vertex that already exists in the priority queue. Technically, you want to *find the key value associated with vertex 2 in the queue and decrease its priority value from +INF to 3*, such that the resulting priority queue is **(2, p=3) > (4, p=4) > (3, p=8)**.

Continuing from this priority queue, remove vertex 2 since it has the smallest priority value and you can observe from the presence of the edge (2,4) that you can connect to vertex 4 with a distance of 1 *instead of the current distance of 4 as maintained in the priority queue*. Similarly, with edge (2,3) you can connect to vertex 3

with a distance of 4 *instead of the current distance of 8 as shown in the priority queue*. To do that, you need to *find the key value associated with these two vertices in the queue and decrease that key value so that the resulting priority queue is (4, p=1) > (3, p=5)*.

Now you're faced with a dilemma: locating a given element in a priority queue is potentially an $O(n)$ operation. Altering the priority associated with an element in the priority queue seems like it can only be done safely by removing the element first and then reinserting it with the new priority.

Start by creating a class to represent an edge in the computed minimum spanning tree.

 In the **mst** package, create an **Edge** class as shown:

CODE TO TYPE: Edge class

```
package mst;

public class Edge {
    int start;
    int end;

    public Edge (int s, int e) {
        start = s;
        end = e;
    }

    public String toString() {
        return "" + start + "-" + end;
    }
}
```

The code below demonstrates the naive use of **PriorityQueue** from the Java Collections Framework. The initial graph is represented as a two-dimensional adjacency matrix where the value of **graph[i][j]** is the weight associated with the edge (i,j) ; if no such edge exists, then **graph[i][j] = 0**.

 In the **mst** package, create a **Driver** class as shown:

CODE TO TYPE: Driver

```
package mst;

public class Driver {
    public static void main (String[] args) {
        int[][] graph = new int[][] {
            {0, 2, 0, 8, 4},
            {2, 0, 3, 0, 0},
            {0, 3, 0, 5, 1},
            {8, 0, 5, 0, 7},
            {4, 0, 1, 7, 0}};

        Edge[] mst = MST.compute(graph);

        for (Edge e : mst) {
            System.out.println(e + "(" + graph[e.start][e.end] + ")");
        }
    }
}
```

The above code represents the graph used in the earlier example by an adjacency matrix. It computes a minimum spanning tree (using the MST class that you will write shortly). The returned array **mst[k]** represents the $n-1$ edges in the computed minimum spanning tree.

 In the **mst** package, create an **MST** class as shown:

CODE TO TYPE: MST class

```
package mst;

import java.util.*;


public class MST {
    static Edge[] compute(int[][] graph) {
        int n = graph.length;
        Edge[] mst = new Edge[n-1];

        PriorityQueue<Pair> pq = new PriorityQueue<Pair>(n, new PriorityComparator()
    );
        for (int i = 1; i < n; i++) {
            pq.add(new Pair(i, Integer.MAX_VALUE));
            mst[i-1] = new Edge(i, -1);
        }
        pq.add(new Pair(0, 0));

        while (!pq.isEmpty()) {
            int u = pq.remove().key;

            for (int v = 0; v < n; v++) {
                int weight = graph[u][v];
                if (weight > 0) {
                    for (Pair pv : pq) {
                        if ((pv.key == v) && (weight < pv.priority)) {
                            mst[v-1].end = u;
                            pq.remove(pv);
                            pv.priority = weight;
                            pq.add(pv);
                            break;
                        }
                    }
                }
            }
        }

        return mst;
    }
}
```

 Save all of the new files and run the **Driver** class; you see output which corresponds to the manual computation from earlier. When given a connected graph of n vertices, the corresponding output will have $n-1$ edges:

INTERACTIVE SESSION: Output of Driver

```
1-0 (2)
2-1 (3)
3-2 (5)
4-2 (1)
```

Let's look at this code more closely:

OBSERVE: MST initialization

```
int n = graph.length;
Edge[] mst = new Edge[n-1];

PriorityQueue<Pair> pq = new PriorityQueue<Pair>(n, new PriorityComparator());
for (int i = 1; i < n; i++) {
    pq.add(new Pair(i, Integer.MAX_VALUE));
    mst[i-1] = new Edge(i, -1);
}
pq.add(new Pair(0, 0));
```

This code initializes the data structures used by the algorithm. Prim's Algorithm starts at some vertex—for the implementation, you will start at vertex 0, so the priority queue will initially contain **Pair** objects for each of the other $n-1$ vertices with +Infinity as the computed minimum distance. The final initialization code **inserts vertex 0 into the priority queue** with a priority of 0, which ensures that this **Pair** object will be the first one removed from the priority queue.

All of the logic is contained in this **while** loop:

OBSERVE: Main loop of Prim's Algorithm

```
while (!pq.isEmpty()) {
    int u = pq.remove().key;

    for (int v = 0; v < n; v++) {
        int weight = graph[u][v];
        if (weight > 0) {
            for (Pair pv : pq) {
                if ((pv.key == v) && (weight < pv.priority)) {
                    mst[v-1].end = u;
                    pq.remove(pv);
                    pv.priority = weight;
                    pq.add(pv);
                    break;
                }
            }
        }
    }
}
```

This code implements Prim's Algorithm. **While the priority queue pq is not empty**, the **Pair object with lowest priority is removed** and its **vertex u is identified**. The first **for loop** checks all other vertices, **v**, **to see if there is a direct edge connecting u and v**. If a direct edge is found, **weight will be a value greater than zero**, so you have to determine if this new distance is smaller than the current shortest distance to **v** already being maintained in **pq**. If it turns out that **weight** is smaller, **the old pair pv must be removed from pq and reinserted with the lower priority**.

You can evaluate the performance of this implementation by reviewing the number of loops in the code. Essentially there is a triply nested loop, each of which iterates over all n vertices. This gives the worst-case estimate of $O(n^3)$ for how frequently elements in the priority queue are adjusted. The **add** and **remove** operations on a PriorityQueue perform in $O(\log n)$, so the performance for the entire algorithm is $O(n^3 \log n)$. Surely we can do better!

Heap Data Structure

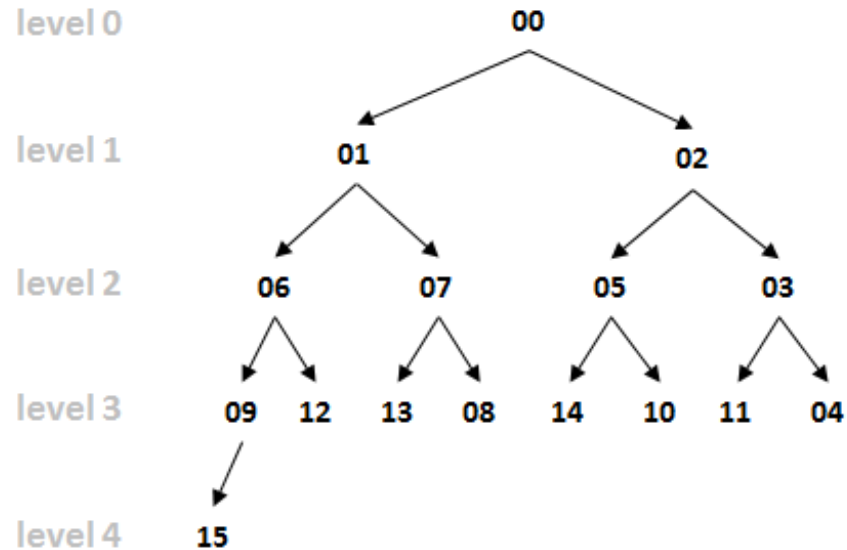
The previous section requires a data structure that returns the smallest element of a collection in constant time; but it also needs to be able to locate a particular element in the collection and reprioritize it efficiently (which means in $O(\log n)$ time).

There is an interesting data structure known as a *heap* that can serve our purposes. A heap is a binary tree with a structure that ensures two properties:

- **Shape property:** A leaf node at depth $k > 0$ can exist only if all 2^{k-1} nodes at the previous level $k-1$ exist. Additionally, nodes at a partially filled level must be added from left to right.
- **Heap property:** Each node in the tree contains a value smaller than or equal to either of its two

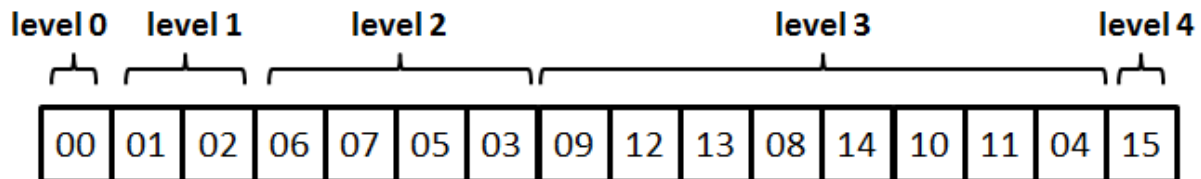
children (if it has any).

The image below represents a sample heap of 16 integer values from 0 to 15:




The heap consists of a number of levels. The value associated with each node in the heap is guaranteed to be smaller than or equal to both of its children. For this reason, the root of the heap always contains the smallest value in the entire heap. Note that each level in the heap is fully filled before new elements are added to the next level.

Given the rigid structure imposed by the shape property, a heap can be implemented efficiently within an array *A* without losing any of its structural information. The image below demonstrates how a heap can be stored in an array by storing the element value for a node in the array position identified by the node's label. The order of the elements within *A* can be read from left to right as deeper levels of the tree are explored.



Develop code that allows you to create a heap efficiently. In this lesson, we assume that you know the maximum size of the heap *in advance* when you construct it. Let's get started.

 In the **mst** package, create a new class named **Heap** as shown:

CODE TO TYPE: Heap class

```
package mst;

public class Heap {
    int n = 0;
    Pair[] elements;

    public Heap(int n) {
        elements = new Pair[n];
    }

    public boolean isEmpty() {
        return (n == 0);
    }

    public void insert (int key, int priority) {
        int idx = n++;
        while (idx > 0) {
            int parent = (idx-1)/2;
            Pair p = elements[parent];

            if (priority >= p.priority) { break; }

            elements[idx] = p;
            idx = parent;
        }

        elements[idx] = new Pair (key, priority);
    }
}
```

Let's take a closer look at this code:

OBSERVE: Construct Heap Storage

```
int n = 0;
Pair[] elements;

public Heap(int n) {
    elements = new Pair[n];
}

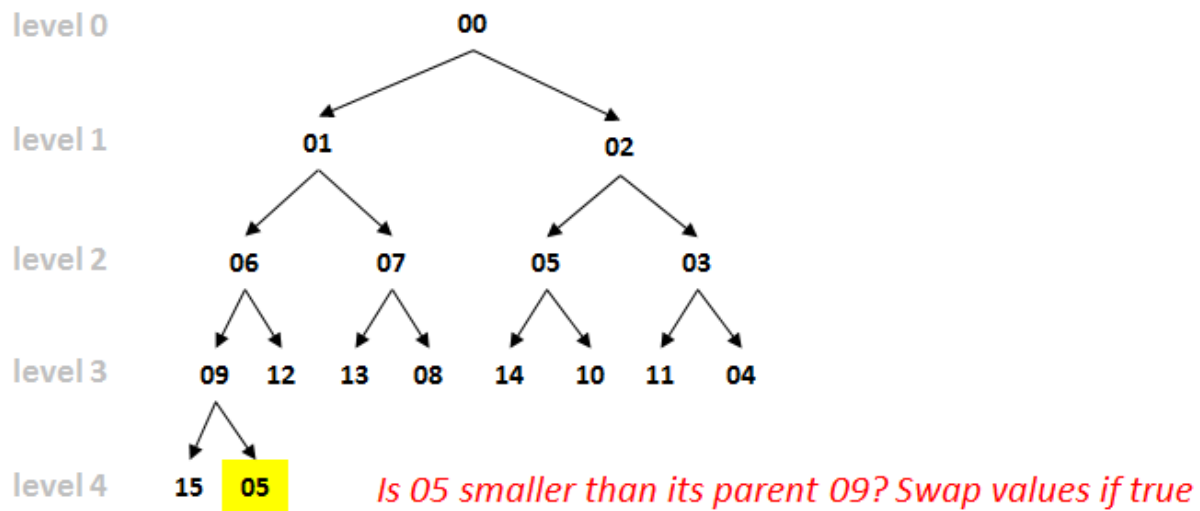
public boolean isEmpty() {
    return (n == 0);
}
```

The **elements** array will store the **Pair** objects representing the elements in the priority queue. Attribute **n** counts the number of elements in the priority queue.

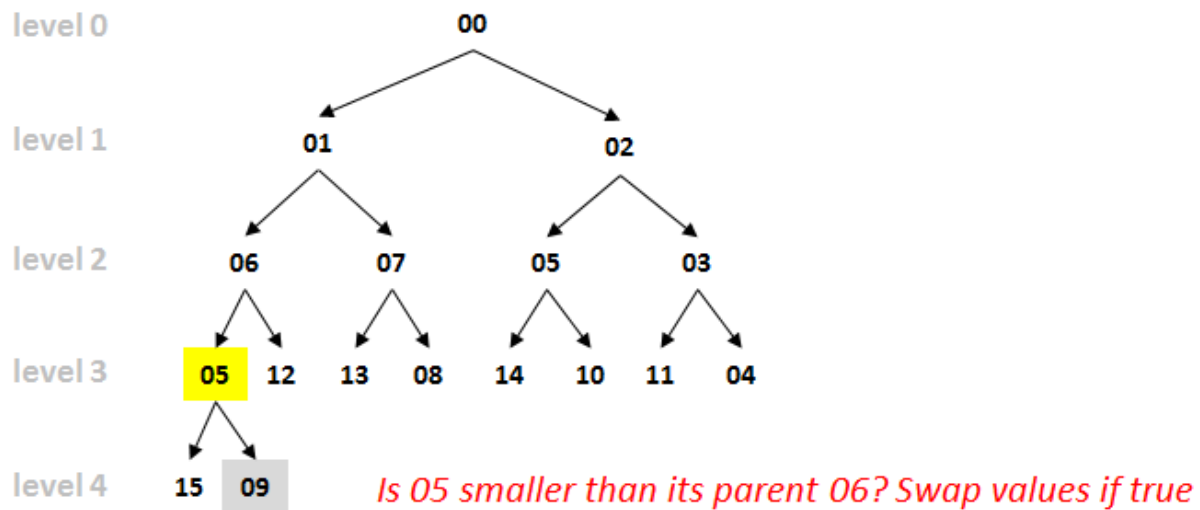
This heap will be used as a priority queue so each element in the heap stores a **Pair** of values, where each *key* has an associated *priority* with lower integer values that represent greater importance. To construct a heap of a maximum size you just need to reserve room for **n** potential **Pair** elements. The heap is empty when its **n** attribute is 0.

To insert an element, recognize that the values in the heap are not fully ordered. Rather, the only global property is that the values on any path of nodes from the root to a leaf stay the same or increase. The heap was created with sufficient space for all values that you will insert, so when it comes time to insert a value, you can place it in the "next" array location. In doing so, you continue to conform to the *Shape Property* of the heap. However, when inserting an element at this location, you may violate the *Heap Property*, so you'll have to make some adjustments. The good news is that you don't have to reorder all elements in the heap; rather, you need to focus on the ancestor nodes of the new element, going all the way back to the root of the heap.

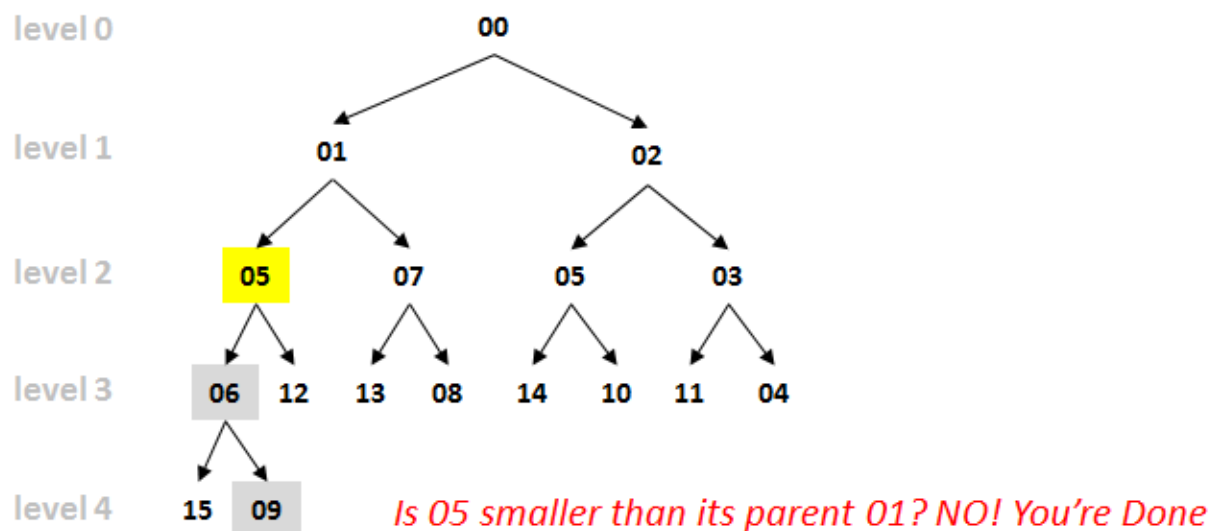
Suppose you insert "05" into the heap of 16 elements shown earlier. First, it's placed in the 17th location:



Since 05 is smaller than 09, the two values in the nodes are swapped. Continue up to the parent node on level 2 to see if its contents are smaller than 05.



Once again, the two values in the nodes are swapped. Keep going, checking with the parent node on level 1 to see if its contents are smaller than 05.



Once you hit a node with a value that's smaller than the newly added element, you're done. The heap is now guaranteed to have both its *Heap* and *Shape* properties. In the *worst case*, you only have to check and

potentially swap $O(\log n)$ values in the heap, so inserting an element is $O(\log n)$ and constructing a heap of n elements in the is $O(n \log n)$.

Let's look closer at the **insert** method:

OBSERVE: Insert (key, priority) into Heap

```
public void insert (int key, int priority) {
    int idx = n++;
    while (idx > 0) {
        int parent = (idx-1)/2;
        Pair p = elements[parent];

        if (priority >= p.priority) { break; }

        elements[idx] = p;
        idx = parent;
    }

    elements[idx] = new Pair (key, priority);
}
```

The heap is aware that you are trying to add a key value with an associated priority. The **insert** method first **increments the count of elements n** and considers the new **Pair** location at index location **idx**. Given an index location in the heap of **idx**, the **parent node is computed as $(idx-1)/2$** . **If the new priority being added is larger than this Pair's priority, you're done**, and the final line of the method **creates a Pair object in location idx to contain the associated key and priority values**. If, however, you still have to swap node values, move the parent **Pair p** within the **while** loop into the child's location **idx** and repeat, setting **idx** to the *parent* location to move up a level to check once again for the *Heap Property*. If you continue to swap values all the way up to the root (which has index location 0), the **while** loop will exit and the new **Pair** will be placed there. In the worst case, the **while** loop requires $O(\log n)$ iterations.

To use the heap data structure as a priority queue, you must be able to locate and remove the element with the lowest priority. As stated earlier, the root of the heap will always contain the **Pair** of the lowest priority. However, you can't simply remove this node because that would violate both the Heap and Shape properties. Instead, remove the root and replace it with *the last Pair in the heap*. In doing so, you will maintain the Shape property, but now you'll have to manipulate the heap to restore the Heap property, which states that each node must be smaller than both of its children.

Add this method to the end of the **Heap** class:

CODE TO TYPE: smallest() method for Heap

```
public int smallest () {
    int key = elements[0].key;
    Pair last = elements[--n];
    elements[0] = last;

    int idx = 0;
    int child = 2*idx+1;
    while (child <= n) {
        Pair smaller = elements[child];
        if (child < n) {
            if (smaller.priority > elements[child+1].priority) {
                smaller = elements[++child];
            }
        }

        if (last.priority <= smaller.priority) { break; }

        elements[idx] = smaller;
        idx = child;
        child = 2*idx+1;
    }

    elements[idx] = last;
    return key;
}
```

Let's look at this code more closely.

OBSERVE: smallest() method for Heap

```
public int smallest () {
    int key = elements[0].key;
    Pair last = elements[--n];
    elements[0] = last;


    int idx = 0;
    int child = 2*idx+1;
    while (child <= n) {
        Pair smaller = elements[child];
        if (child < n) {
            if (smaller.priority > elements[child+1].priority) {
                smaller = elements[++child];
            }
        }

        if (last.priority <= smaller.priority) { break; }

        elements[idx] = smaller;
        idx = child;
        child = 2*idx+1;
    }

    elements[idx] = last;
    return key;
}
```

It starts by **remembering the key of the Pair with lowest priority** (that is, the root of the heap), since that's the value being returned by the method. Then it takes the **last Pair in the heap and moves it into the root position**. Now the **while** loop is similar to the loop within the **insert** method. The difference is that you are starting from **idx=0** (the root) and working down some path in the heap, to find the smaller of the two children of position **idx**. The first child is found at index location $2*idx+1$ and the second at $idx*2+2$. The above code **finds the child with the lowest priority and breaks out of the loop if the Heap property is still maintained**. Otherwise it swaps the smaller value with the parent location of **idx** and continues down that child path. This **while** loop will never iterate more than $\log n$ times.

 To test the heap in use, create a new class named **HeapDriver** in the **mst** package as shown:


CODE TO TYPE: HeapDriver class

```
package mst;

public class HeapDriver {
    public static void main(String[] args) {
        Heap heap = new Heap(16);
        for (int i = 15; i >= 0; i--) {
            heap.insert(i, i);
        }

        for (Pair p : heap.elements) {
            System.out.print(p.priority + " ");
        }
        System.out.println();

        while (!heap.isEmpty()) {
            System.out.println(heap.smallest());
        }
    }
}
```

 Save and run this class; it constructs a heap of 16 values and repeatedly adds the integers from 15 down to 0 to fill the heap. The program prints out a representation of the array-based storage of the heap (shown earlier) and then demonstrates that it can return the smallest element in the heap, one at a time.

INTERACTIVE SESSION: Output of HeapDriver

```
0 1 2 6 7 5 3 9 12 13 8 14 10 11 4 15
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Prim's Algorithm Implementation

To complete this lesson, you need to modify the **Heap** to be able to support Prim's Algorithm, which needs to locate a value within the heap and decrease its priority value. To make this work, you need to make a number of modifications to **Heap**:

CODE TO TYPE: Modifications to Heap

```
package mst;

public class Heap {
    int n = 0;
    Pair[] elements;
    int[] positions;

    public Heap(int n) {
        elements = new Pair[n];
        positions = new int[n];
    }

    public boolean isEmpty() {
        return (n == 0);
    }

    public void insert (int key, int priority) {
        int idx = n++;
        while (idx > 0) {
            int parent = (idx-1)/2;
            Pair p = elements[parent];

            if (priority >= p.priority) { break; }

            elements[idx] = p;
            positions[p.key] = idx;
            idx = parent;
        }

        elements[idx] = new Pair (key, priority);
        positions[key] = idx;
    }

    public int smallest () {
        int key = elements[0].key;
        Pair last = elements[--n];
        elements[0] = last;

        int idx = 0;
        int child = 2*idx+1;
        while (child <= n) {
            Pair smaller = elements[child];
            if (child < n) {
                if (smaller.priority > elements[child+1].priority) {
                    smaller = elements[++child];
                }
            }

            if (last.priority <= smaller.priority) { break; }

            elements[idx] = smaller;
            positions[smaller.key] = idx;
            idx = child;
            child = 2*idx+1;
        }

        elements[idx] = last;
        positions[last.key] = idx;
        return key;
    }


    void decreasePriority (int key, int newPriority) {
        int size = n;
        n = positions[key];
        insert(key, newPriority);
        n = size;
    }
}
```

```
}  
}
```

The essence of the change is to be able to store the location in the array-based heap of each key value in the heap. This works because the key values themselves are integers in the range $[0, n)$. Every time a **Pair** object p is inserted into $elements[idx]$, there is a corresponding $positions[p.key] = idx$ to record that fact.

The reason to maintain positions is evident in the final method being added to **Heap**, which decreases the priority for a given key value found in the heap. In order to decrease priority of a **Pair**, you need to reduce the associated priority value with the **Pair**. This method only works if you are truly decreasing the priority (increasing the priority would break the Heap Property). This method works by reusing the **insert** method. First it truncates the heap up to, but not including the location where key is currently stored in the heap (it does this by **setting n to be the key's location in the heap**). Then it **invokes insert using the original key value, but the new priority**; as discussed earlier, this will reestablish the Heap Property. Finally, since the newPriority must be lower, you can **expand the size of the heap back to its original size** and still have a working heap.

Now you're ready to write a revised Prim's Algorithm that uses the heap data structure as a priority queue.

 In the **mst** package, create a **HeapMST** class as shown:

CODE TO TYPE: HeapMST class

```
package mst;  
  
public class HeapMST {  
    PriorityComparator comp = new PriorityComparator();  
  
    static Edge[] compute(int [][] graph) {  
        int n = graph.length;  
        Edge[] mst = new Edge[n-1];  
  
        boolean inQueue[] = new boolean [n];  
        Heap heap = new Heap(n);  
        int[] priorities = new int[n];  
  
        heap.insert(0,0);  
        for (int i = 1; i < n; i++) {  
            priorities[i] = Integer.MAX_VALUE;  
            heap.insert(i, priorities[i]);  
            inQueue[i] = true;  
            mst[i-1] = new Edge(i, -1);  
        }  
  
        while (!heap.isEmpty()) {  
            int u = heap.smallest();  
            inQueue[u] = false;  
  
            for (int v = 0; v < n; v++) {  
                int weight = graph[u][v];  
                if (weight > 0 && inQueue[v]) {  
                    if (weight < priorities[v]) {  
                        mst[v-1].end = u;  
                        priorities[v] = weight;  
                        heap.decreasePriority(v, weight);  
                    }  
                }  
            }  
        }  
  
        return mst;  
    }  
}
```

The structure of this code is probably familiar to you from the **MST** class. Let's take a closer look:

OBSERVE: Initializing for Prim's Algorithm

```
static Edge[] compute(int [][] graph) {
    int n = graph.length;
    Edge[] mst = new Edge[n-1];

    boolean inQueue[] = new boolean [n];
    BinaryHeap heap = new BinaryHeap(n);
    int[] priorities = new int[n];

    heap.insert(0,0);
    for (int i = 1; i < n; i++) {
        priorities[i] = Integer.MAX_VALUE;
        heap.insert(i, priorities[i]);
        inQueue[i] = true;
        mst[i-1] = new Edge(i, -1);
    }
}
```

This implementation uses two additional arrays: **priorities** stores the current best distance from any vertex in set *S* to the vertices remaining in *T*. **inQueue** determines whether a given vertex is currently in the priority queue. Initially the heap is constructed with the **designated start vertex 0 being inserted with greatest importance (a priority of 0)**, while the other *n-1* vertices are inserted with least importance (maximum priority). **inQueue** is set to true for the *n-1* vertices in the priority queue. The **mst** array will store the computed edges for each vertex. Currently their end values are -1 to declare that they have yet to be computed.

The real logic occurs in the **while** loop:

OBSERVE: Prim's Algorithm Implementation

```
while (!heap.isEmpty()) {
    int u = heap.smallest();
    inQueue[u] = false;

    for (int v = 0; v < n; v++) {
        int weight = graph[u][v];
        if (weight > 0 && inQueue[v]) {
            if (weight < priorities[v]) {
                mst[v-1].end = u;
                priorities[v] = weight;
                heap.decreasePriority(v, weight);
            }
        }
    }
}
```

As long as the heap is not empty, it **retrieves the vertex *u* with the smallest distance to any vertex in set *S***. Now that this vertex is out of the queue, **inQueue[*u*] is set to false**. The inner **for** loop still must iterate over all the other *n* vertices to find if there is an edge (*u,v*) with a distance that's shorter than previously recorded. The **priorities** array lets the code determine this quickly. If so, the priority is adjusted in the **priorities** array and the **element's location is adjusted in the priority queue using the decreasePriority method**.

To evaluate the performance of this algorithm, assume there are *n* vertices and *k* edges in the graph. During the initialization phase, each vertex is inserted into the priority queue for a total cost of $O(n \log n)$. The **decreasePriority** method requires no less than $O(\log n)$ time. It can be called $2*k$ times at most since each vertex is removed once from the priority queue and each edge in the graph is visited exactly twice. So, total performance is $O((n+2*k) \log n)$. If the graph is dense, *k* can be as high as $n*(n-1)/2$, so the worst case performance is $O(n^2 \log n)$. If the graph is really sparse, then *k* is on the order of $O(n)$ which results in $O(n \log n)$ performance.

Evaluating Minimum Spanning Tree Implementations

Compare these two implementations head to head on the same graph.

 In the **mst** package, create a **Comparison** class as shown:

CODE TO TYPE: Comparison class

```
package mst;

public class Comparison {
    public static void main (String[] args) {
        System.out.println("n\tHeapMST\t\tMST");
        for (int n = 16; n <= 1024; n*= 2) {
            int[][] graph = new int[n][n];
            for (int i = 0; i < n-1; i++) {
                for (int j = i+1; j < n; j++) {
                    int w = (int) (Math.random()*n);
                    graph[i][j] = w;
                    graph[j][i] = w;
                }
            }

            System.gc();
            long now = System.nanoTime();
            Edge[] mst = MST.compute(graph);
            long then = System.nanoTime();
            System.gc();
            Edge[] mst2 = HeapMST.compute(graph);
            long last = System.nanoTime();

            for (int i = 0; i < mst.length; i++) {
                if ((mst[i].start != mst2[i].start) ||
                    (mst[i].end != mst[i].end)) {
                    System.err.println("ERROR");
                    System.exit(0);
                }
            }

            float m = 1000000;
            System.out.println(n + "\t" + (last-then)/m + " \t" + (then-now)/m);
        }
    }
}
```



Save and run the code:

INTERACTIVE SESSION: Comparison Output

k	HeapMST	MST
16	4.720959	2.177451
32	3.310271	2.00772
64	3.199684	1.886284
128	4.528132	10.268524
256	5.046072	76.359505
512	5.648354	596.32855
1024	12.14046	4683.4565

For small values of n , the original **MST** implementation outperforms the **HeapMST** implementation. However, the true nature of the heap implementation demonstrates rapidly that it's incredible efficiency when compared against the obvious counterpart.

Lessons Learned

The Collection Framework offers a well-designed set of classes that will be useful when you apply the common data list, hash, and tree data structures. At the same time, the designers created a uniform interface to all Collection classes. In doing so, they created methods that do not behave as efficiently as other specialized data structures. In particular, the **PriorityQueue** Collection class will provide the exact behavior for priority queues with elements that cannot change priority once they've been inserted into the queue. Prim's Algorithm demands this behavior, so the default **PriorityQueue** implementation will not suffice. You must

always read the documentation that accompanies the Collection Framework classes, because each class provides information about the performance of their important methods.

You can improve performance by storing additional information to reduce the number of computations needed. The **HeapMST** implementation is able to reduce performance time with only a modest investment in storage. In your own algorithms, try to make this tradeoff to achieve the same benefits.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Binary Tree Data Structure

Lesson Objectives

After completing this lesson, you will be able to:

- describe the structure of a Binary Search Tree (BST).
- draw the BST after inserting a number of elements in a specific order.
- demonstrate how to rebalance an AVL tree after inserting a node.

Binary Tree Data Structure

A Binary Search Tree is a recursive data structure central to computer science. In earlier lessons we saw how Linked Lists provide dynamic behavior that improves on contiguous arrays. However, Linked Lists only provide $O(n)$ behavior for determining whether an element is in the list.

Given an array of sorted items, you can use *Binary Array Search* to determine efficiently whether the array contains a given item in $O(\log n)$ time. This code shows how to implement the algorithm:



Create a new Java project named **BinaryTree** for this lesson's work, and assign it to the **Java6_Lessons** working set.



In the **BinaryTree** project's **/src** source folder, create a **binary** package.



In the **binary** package, create a **BinaryArraySearch** class as shown:

CODE TO TYPE: BinaryArraySearch

```
package binary;

public class BinaryArraySearch {
    public static void main (String[] args) {
        int[] vals = new int [] { 2, 5, 8, 11, 15, 17 };

        System.out.println("7 goes in position " + binarySearch (vals, 7));
        System.out.println("2 found in position " + binarySearch (vals, 2));
    }

    public static int binarySearch(int[] A, int val) {
        int low = 0;
        int high = A.length-1;
        while (low <= high) {
            int mid = (low + high)/2;
            if (val < A[mid]) {
                high = mid-1;
            } else if (val > A[mid]) {
                low = mid + 1;
            } else {
                return mid;
            }
        }

        return -(low + 1);
    }
}
```



Save and run it:

OBSERVE: Sample execution of BinaryArraySearch

```
7 goes in position -3  
2 found in position 0
```

Because 7 does not appear in the array, you can use the return value from **binarySearch** to determine where in the array 7 *could be inserted* to maintain the sorted order. When the return value is less than zero, negate it and subtract 1 to find the correct location to store the value in **A**. In this case, 7 should be inserted at index location 2, which would place it between 5 and 8 as it should be. The second line of output shows that the search is able to locate element 2 at index location 0.

Arrays are unable to delete and insert items efficiently while maintaining a specific ordering of elements. However, Linked Lists can insert elements anywhere in the collection, but then searching for a given item will require $O(n)$ time. Binary Search Trees offer the impressive ability to maintain items in a structured order and on average, it can support adding, removing, and searching for items in $O(\log n)$ time.

In this lesson, you will construct a Binary Search Tree implementation from scratch. Through various performance code you'll see that the naive implementation can lead to worst-case performance of $O(n)$ for all key operations. At the end of the lesson, you'll learn how to "balance" the tree to improve the average case performance.

Naive Binary Tree Implementation

A Binary Search Tree is a finite set of nodes where each node stores a typed value known as the *key* for the node. A non-empty BST contains a special *root* node that is the ancestor of all other nodes in the BST. Each node n in the BST refers to two binary search subtrees, *left* and *right*, and obeys the property that if k is the key for node n , then all keys in *left* are $\leq k$ and all keys in *right* are $> k$. This property is known as the *binary search tree property*. If subtrees *left* and *right* are null, the node is called a *leaf node*.

Given a BST, the three primary operations are:

- Add a new key.
- Remove a key.
- Determine if a key value exists.

There can be two or more nodes in the search tree with the same *key* value, but if you want to restrict the tree to conform to Set-based semantics as defined in the Java Collections Framework while ensuring that the same implementation will work, you need to prevent the insertion of duplicate keys. For this lesson, assume that duplicate keys may exist in the BST.

 In the `/src` source folder **binary** package, create a **BinaryNode** class as shown:

CODE TO TYPE: BinaryNode class

```
package binary;  
  
public class BinaryNode<E extends Comparable<E>> {  
    final E key;  
    BinaryNode<E> left;  
    BinaryNode<E> right;  
  
    public BinaryNode(E k) {  
        this.key= k;  
    }  
  
    public int size() {  
        return 1 + size(left) + size(right);  
    }  
  
    int size(BinaryNode<E> n) {  
        if (n == null) { return 0; }  
        return n.size();  
    }  
}
```

A **BinaryNode<E>** class represents a node in the BST with a corresponding **key** value. BinaryNode is a generic class with the parameter, **E**, that determines the type of the **key** attribute. The only restriction is that the

type of the key must implement **Comparable**, otherwise there will be no way to order the key values.

BinaryNode defines *left* and *right* attributes to refer to the left and right subtrees, respectively. The **size()** method counts the nodes in a BST rooted at a given node. Because a BST is a recursive data structure, the implementation of **size()** is also recursive. Throughout this lesson, you see how to apply recursion to implement the required BST operations. The **size(n)** 'll helper method allows **size()** to be written in its simplest form. So, the size of a BST rooted at a given node **n** is 1 plus the respective sizes of the left and right subtrees or **n**.

Because the BST is composed of BinaryNode objects representing keys in the BST, we need to write an **add(E)** method that inserts a key into the BST rooted at a given node. Add these methods to the end of the **BinaryNode** class:

CODE TO TYPE: Modifications to BinaryNode

```
void add (E k) {
    int rc = k.compareTo(key);
    if (rc <= 0) {
        left = add(left, k);
    } else {
        right = add(right, k);
    }
}

BinaryNode<E> add(BinaryNode<E> parent, E k) {
    if (parent == null) {
        return new BinaryNode<E>(k);
    }


    parent.add(k);
    return parent;
}
```

The *binary search tree property* states that all keys in the left subtree of a node are less than or equal to the node's key, and all keys in the right subtree of a node are greater than the node's key. For this discussion, assume that **add(k)** is invoked on node **n** where **k** is less than or equal to **n**'s key. There are two cases to consider:

- The node **n** has no left subtree.
- The node **n** has a left subtree.

If there is no left subtree then a new node containing this key is created to be the left subtree of **n**. This is the case in **add(parent,k)** when **parent** is **null**. If, however, the left subtree does exist, then **add(parent,k)** requests to add **k** recursively to that subtree. **add(parent,k)** either returns the new node which was created or the existing node that now has a descendant node representing the newly added key. These two functions present another example of double recursion, where each function calls the other repeatedly until the computation terminates. Note that the structure of these two methods is similar to the **size()** method presented earlier.

Now you can create a **BinaryTree** class to take advantage of this **add** capability.

 In the **binary** package, create a **BinaryTree** class as shown:

CODE TO TYPE: BinaryTree class

```
package binary;

public class BinaryTree<E extends Comparable<E>> {

    BinaryNode<E> root = null;

    public int size() {
        if (root == null) { return 0; }

        return root.size();
    }

    public void add (E k) {
        if (root == null) {
            root = new BinaryNode<E>(k);
            return;
        }

        root = root.add(root,k);
    }
}
```

Let's take a closer look at a couple of things.

OBSERVE:

```
public int size() {
    if (root == null) { return 0; }

    return root.size();
}

public void add (E k) {
    if (root == null) {
        root = new BinaryNode<E>(k);
        return;
    }

    root = root.add(root,k);
}
```

The BinaryNode attribute **root** represents the top of the BST; all nodes in the BST are descendants of **root**. **When root == null**, the BST is considered to be empty.

The **add(E k)** code demonstrates an implementation style necessary for BinaryTree. **If root is null**, all methods have to handle the special case where the BST is empty. In this case, if **root is null**, the **root of the BST is set to a new BinaryNode with the given key**. Otherwise, **this key is added to the subtree rooted at root**, using the **add(parent,k)** method described earlier. Doing so allows the **root** to be updated as needed.

Until you add a **contains(E k)** method, there will be no easy way to validate that the above methods work. Add these methods to the end of **BinaryTree**:

CODE TO TYPE: Modifications to BinaryTree

```
public boolean contains (E k) {
    return contains(root, k);
}

boolean contains (BinaryNode<E> parent, E k) {
    if (parent == null) { return false; }

    int rc = k.compareTo(parent.key);
    if (rc == 0) {
        return true;
    } else if (rc < 0) {
        return contains(parent.left, k);
    } else {
        return contains(parent.right, k);
    }
}

public int height () {
    if (root == null) { return 0; }


    return height(root);
}

int height (BinaryNode<E> n) {
    if (n == null) { return 0; }


    return 1 + Math.max( height(n.left), height(n.right));
}
```

This method is placed in the **BinaryTree** class, rather than **BinaryNode**, because the notion of "containment" is a property of the BST, not an individual node. Also, doing so allows you to avoid constantly checking to find out whether the **left** or **right** subtree is empty. Specifically, **contains(parent,k)** returns **false** when **parent** node is **null**. This method recursively calls itself on either the left subtree or the right subtree if the node's key doesn't match the target key. Naturally, once a match is found, **true** is returned.

You can demonstrate proper functioning of **BinaryTree** with this JUnit test case:

 Create a **test** source folder if it doesn't already exist.

 Create a **binary** package.

 in the **/test** source folder **binary** package, create a **JUnit Test Case** named **TestBinaryTree** as shown:

Note

You may be prompted to choose JUnit 3 or JUnit 4. If you're don't know which to choose, go with JUnit 3. Either is okay, but the resulting file may look slightly different from the one shown here.

CODE TO TYPE: TestBinaryTree

```
package binary;

import java.util.*;

import junit.framework.TestCase;

public class TestBinaryTree extends TestCase {

    public void testAdditions() {
        int numToAdd = 100;
        ArrayList<Integer> vals = new ArrayList<Integer>();
        for (int i = 1; i < numToAdd; i += 2) {
            vals.add(i);
        }
        Collections.shuffle(vals);
        Integer[] add = vals.toArray(new Integer[]{});

        BinaryTree<Integer> bst = new BinaryTree<Integer>();

        for (int i : add) {
            bst.add(i);
        }

        assertEquals (numToAdd/2, bst.size());

        for (int i = 1; i < numToAdd; i++) {
            if (i % 2 == 1) {
                assertTrue (bst.contains(i));
            } else {
                assertFalse (bst.contains(i));
            }
        }
    }
}
```


Launch this JUnit test case; it validates that the BST only contains the odd numbers from 1 to 100.


Evaluating Binary Tree Implementation

To determine the efficiency of **BinaryTree**, you need to identify the worst case and average case execution of its methods. You might be able to identify the worst case behavior, that is, when keys are inserted into a BST in increasing sorted order. For example, consider adding the numbers from 1 to 10 into a BST. Each newly inserted key becomes the right-most node in the BST. In fact, the structure more closely resembles a linked list than a tree because none of the nodes in the BST have a left subtree.

To demonstrate the performance of the **add(k)** method in **BinaryTree**, write this class:

 In the **BinaryTree** project, create a **/performance** source folder.

 In the **/performance** source folder, create a **binary** package.

 In the **binary** package, create an **Evaluate** class as shown:

CODE TO TYPE: Evaluate class

```
package binary;

import java.util.*;

public class Evaluate {
    static int numTrials = 100;

    public static void main(String[] args) {

        System.out.println("N\tShuffled Stats & Time\tOrdered Stats & Time");
        System.out.println("----\t-----\t-----");
        for (int n = 128; n <= 65536; n *= 2) {
            int totalShuffledHeight = 0;
            int totalOrderedHeight = 0;
            long totalShuffled = 0;
            long totalOrdered = 0;
            int min = n;
            int max = 0;
            for (int t = 0; t < numTrials; t++) {
                ArrayList<Integer> vals = new ArrayList<Integer>();
                for (int i = 1; i < n; i++) {
                    vals.add(i);
                }
                Integer[] ordered = vals.toArray(new Integer[]{});
                Collections.shuffle(vals);
                Integer[] shuffled = vals.toArray(new Integer[]{});

                BinaryTree<Integer> bst = new BinaryTree<Integer>();
                long now = System.nanoTime();
                for (int i : shuffled) {
                    bst.add(i);
                }
                totalShuffled += (System.nanoTime() - now);

                int h = bst.height();
                if (h < min) { min = h; }
                if (h > max) { max = h; }
                totalShuffledHeight += h;

                bst = new BinaryTree<Integer>();
                now = System.nanoTime();
                for (int i : ordered) {
                    bst.add(i);
                }
                totalOrdered += (System.nanoTime() - now);
                totalOrderedHeight += bst.height();
            }

            System.out.println(n + "\t[" + min + "-" + max + ", avg:" +
                totalShuffledHeight/numTrials + "] " +
                totalShuffled/numTrials + "\t[avg:" + totalOrderedHeight/numTrials + "
] " +
                totalOrdered/numTrials);
        }
    }
}
```

Evaluate conducts 100 random trials of creating BSTs with n nodes, ranging from $n=128$ to $n=65536$; n keys ($1 \dots n$) are inserted. Let's take a closer look at this code:

OBSERVE: Creating a BST from keys inserted in random order

```
ArrayList<Integer> vals = new ArrayList<Integer>();
for (int i = 1; i < n; i++) {
    vals.add(i);
}
Integer[] ordered = vals.toArray(new Integer[]{});
Collections.shuffle(vals);
Integer[] shuffled = vals.toArray(new Integer[]{});

BinaryTree<Integer> bst = new BinaryTree<Integer>();
long now = System.nanoTime();
for (int i : shuffled) {
    bst.add(i);
}
totalShuffled += (System.nanoTime() - now);

int h = bst.height();
if (h < min) { min = h; }
if (h > max) { max = h; }
totalShuffledHeight += h;
```

Two arrays of N keys are created; **ordered** contains the keys in order while **shuffled** is created by using the **Collections.shuffle** method to distribute the keys randomly. The code conducts **100** trials and records the total time (in nanoseconds) required to add all keys to the BST. The code estimates the average cost of adding a random key to the BST by averaging the total time.

In addition, the code maintains statistics on the height of the BSTs generated during this process. For the random BSTs, it records the **min** and **max** heights of the BSTs and computes the total height so it can report on the average height for n keys. Similar code records statistics for the BST generated from the ordered insertion of keys. You want to know the average height of the BST because that will determine the performance of the **contains** method.

Here is sample output of **Evaluate**; your mileage may vary:

OBSERVE: evaluating BinaryTree Implementation

```
N Shuffled Stats & Time Ordered Stats & Time
-----
128 [11-18, avg:14] 26110 [avg:127] 90122
256 [14-23, avg:16] 35776 [avg:255] 326004
512 [16-25, avg:19] 83298 [avg:511] 1485010
1024 [19-26, avg:22] 168682 [avg:1023] 6234518
2048 [21-31, avg:25] 392221 [avg:2047] 26031424
Exception in thread "main" java.lang.StackOverflowError
  at binary.BinaryNode.add(BinaryNode.java:24)
  at binary.BinaryNode.add(BinaryNode.java:37)
  at binary.BinaryNode.add(BinaryNode.java:28)
```

The minimum height of a BST with n nodes is $\log(n)$. The average height of the BST created from the shuffled values is about twice the minimum. Also, the average time (in nanoseconds) to search for all n items grows proportionally with n . For example, when n grows from 128 to 512 (a four-fold increase) the time to search for all n numbers takes 3.19 times as long. This is much different in the BST constructed from the ordered keys. The height of the BST is $n-1$ (which means that it's really a linked list). Also, when n grows from 128 to 512, it takes more than 16 times as long to complete all n searches.

Typically you do not have advance warning of the order of the elements being added into the BST, so you need some way to avoid poor performance due to the elements being close to sorted when they were added into the BST. In addition, when the BST degenerates to a Linked List (because items are inserted in sorted order) the recursive **add** method can cause a **StackOverflowError**, as shown above.


Rebalancing Binary Trees

The smallest height for a tree with n elements is $O(\log n)$, which results in a perfectly *balanced* tree with the left subtree of the root containing roughly the same number of values as the right subtree of the root. This *balanced* property should apply recursively to all nodes in the tree, not just the root. You could try to rebuild the entire tree after each insertion to make sure that each node is balanced, but that would require way too

much work. Instead, find some incremental strategy that adjusts the structure of the tree only when it becomes *unbalanced*.

An AVL tree (named after its inventors, Adelson-Velskii, and Landis) is a *self-balancing* BST first described in 1962. In the Collections Framework, the **TreeMap** class is implemented using *Red-Black* trees, which are another form of self-balancing binary tree. After completing this lesson, you'll be able to compare these two approaches to determine which provides the best performance.

Let's define the concept of *height* with AVL nodes. The height of a leaf node is 0 since it has no children. Recursively define the height of an AVL node to be 1 greater than the maximum of the height values of its 2 children nodes (if at least 1 exists). To complete this definition, consider the height of a non-existent child node to be -1. The *height difference* for a node is defined as $height(left) - height(right)$, that is, the height of the left subtree minus the height of the right subtree. An AVL must enforce the *AVL Property* in every node, namely, that the height difference for any node is either -1, 0 or 1.

 In the **/src** source folder, create an **avl** package.

 In the **avl** package, create an **AVLBinaryNode** class as shown:

CODE TO TYPE: AVLBinaryNode class

```
package avl;

public class AVLBinaryNode<E extends Comparable<E>> {
    E key;
    int height;

    AVLBinaryNode<E> left;
    AVLBinaryNode<E> right;

    public AVLBinaryNode(E k) {
        height = 0;
        key = k;
    }

    void computeHeight (AVLBinaryNode<E> n) {
        int height = -1;
        if (n.left != null) {
            height = Math.max(height, n.left.height);
        }
        if (n.right != null) {
            height = Math.max(height, n.right.height);
        }

        n.height = height + 1;
    }

    int heightDifference(AVLBinaryNode<E> n) {
        if (n == null) { return 0; }

        int leftTarget = 0;
        if (n.left != null) {
            leftTarget = 1 + n.left.height;
        }
        int rightTarget = 0;
        if (n.right != null) {
            rightTarget = 1 + n.right.height;
        }
        return leftTarget - rightTarget;
    }
}
```

To be as efficient as possible, each **AVLBinaryNode** stores its computed *height* in addition to the expected attributes of *key*, *left*, and *right*. That is, rather than dynamically computing the height of a node when requested, you only perform this computation when a node is added to the AVL tree. Finally, the **heightDifference** method computes the height difference for a given node, *n*. As with a regular binary tree, you need to define an **AVLBinaryTree** class.

 In the **avl** package, create an **AVLBinaryTree** class as shown:

CODE TO TYPE: AVLBinaryTree

```
package avl;

public class AVLBinaryTree<E extends Comparable<E>> {

    AVLBinaryNode<E> root = null;

    public void add (E k) {
        if (root == null) {
            root = new AVLBinaryNode<E>(k);
            return;
        }

        root = root.add(root, k);
    }

    public boolean contains (E k) {
        return contains(root, k);
    }

    boolean contains (AVLBinaryNode<E> parent, E k) {
        if (parent == null) { return false; }

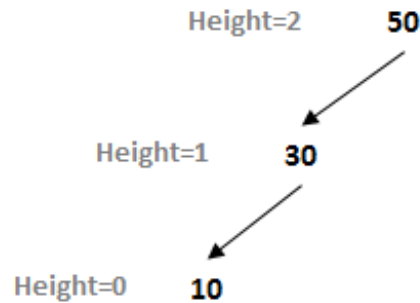
        int rc = k.compareTo(parent.key);
        if (rc == 0) {
            return true;
        } else if (rc < 0) {
            return contains(parent.left, k);
        } else {
            return contains(parent.right, k);
        }
    }
}
```

This code is nearly identical to its **BinaryTree** counterpart. The code won't compile until you complete the **add** method in **AVLBinaryNode**—be careful with this method. It is possible after just three additions to have the root node of a binary tree violate the AVL Property. Consider an AVL tree with just two nodes, constructed by adding 50 and then 30:



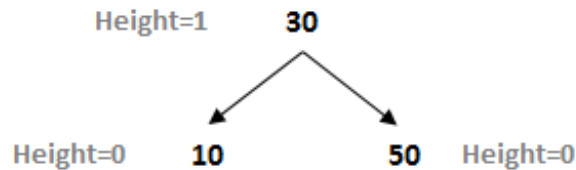
To demonstrate that this tree supports the AVL property, you must compare the heights of the children of the root node (which stores the value 50). However, there is no right subtree for this node. In this situation, the height of an empty child subtree is **-1**. The height difference for the root node is $0 - (-1)$ or $+1$, which satisfies the AVL property.

Now insert the value 10 into the tree, which results in this structure:



First confirm that this binary tree is a BST by making sure that the value for each node is greater than or equal to the value of its left child, and smaller than the value of its right child. The AVL property is maintained by all nodes *except for the root*. The height difference for the root is +2 because the left height is +1 while the (missing) right child's height is -1. The difference violates the AVL Property.

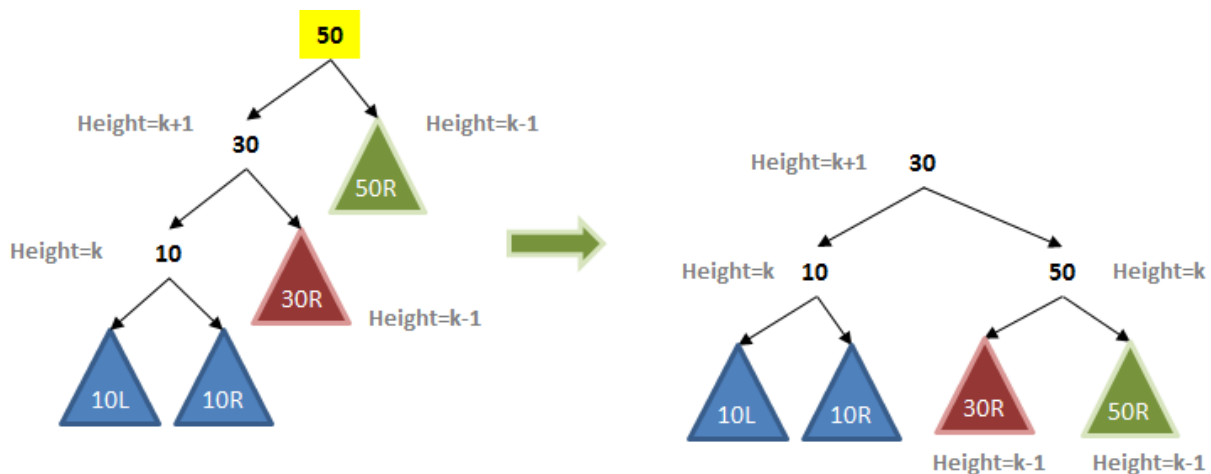
Now, consider this similar tree, which supports the AVL Property:



When the tree is rooted by the node for 30, each of its subtrees is balanced.

After adding the value 10 to the original AVL tree, it is possible to detect that one of its ancestor nodes (the one representing 50) is unbalanced. You can recreate the balanced tree above by performing a *rotate* operation. Imagine "grabbing" the 30 node in the original tree and rotating the tree to the right (or clockwise), pivoting around the 30 node to make 30 the root, thereby creating the balanced tree above. In doing so, only the height of the 50 node has changed (dropping from 2 to 0) and the AVL Property is restored.

This only works because the node 30 in the original tree had no right child. So, what if this tree had lots of other nodes, each of which was perfectly balanced and satisfied the AVL Property? In the image below, each of the shaded triangles represents a potential subtree of the original tree; each is labeled by its position, so **30R** is the subtree representing the right subtree of node 30. The situation on the left occurs immediately after the 10 value is inserted into the tree. The root is the only node that doesn't support the AVL Property. The various heights in the tree are computed assuming that the new node 10 has some height k .



Now when you *Rotate Right*, you can re-attach the entire subtree **30R** so it becomes the left subtree for node **50**. This is possible because all of these values are clearly smaller than 50 since the original tree was a Binary Search Tree. The resulting tree is balanced and all nodes satisfy the AVL Property.

Note

It is possible that the subtree **30R** had a height of k in the tree on the left. In this case, the new node **50** would have a computed height of $k+1$ and the root node **30** would have a computed height of $k+2$. However, that the AVL Property would be properly maintained even in this case.

Add this code to the end of the **AVLBinaryNode** class:

CODE TO TYPE: Modifications to AVLBinaryNode

```
AVLBinaryNode<E> rotateRight () {  
    AVLBinaryNode<E> newRoot = left;  
    AVLBinaryNode<E> grandson = newRoot.right;  
    left = grandson;  
    newRoot.right = this;  
  
    computeHeight(this);  
    return newRoot;  
}
```

This code is best described in the context of the specific example presented above. You invoke **rotateRight** on the unbalanced node, **50**, which is the *this* reference in the above code. *newRoot* is set to the **30** node while *grandson* is the subtree labeled **30R**. The code **left = grandson** sets the left child of **50** to be the subtree **30R**. The code **newRoot.right = this** makes **50** the right child of **30**. Once this manipulation is complete, the height for the **50** nodes is recomputed, but the original height of the **10** node is unaffected. Finally, the new root node of this subtree, **30**, is returned. Observe that its height has *not* yet been recomputed; that will be the responsibility of the method that calls **rotateRight**.

You've seen how to *Rotate Right* to rebalance an AVL tree in the *left-left* case, so named because the new value being added (**10** in this case) was added to left-child of the left-child of the (now-unbalanced) node **50**. Yes, that word repetition is necessary! As you can imagine, there is also a *Rotate Left* operation which can be used to rebalance a tree that is unbalanced in the *right-right* case shown below, so named because the new value being added (**60**) was added to the right-child of the right-child of the (now-unbalanced) node **20**:



In similar fashion, you can perform this *Rotate Left* even when these nodes have subtrees attached to them. Add this code to the end of the **AVLBinaryNode** class:

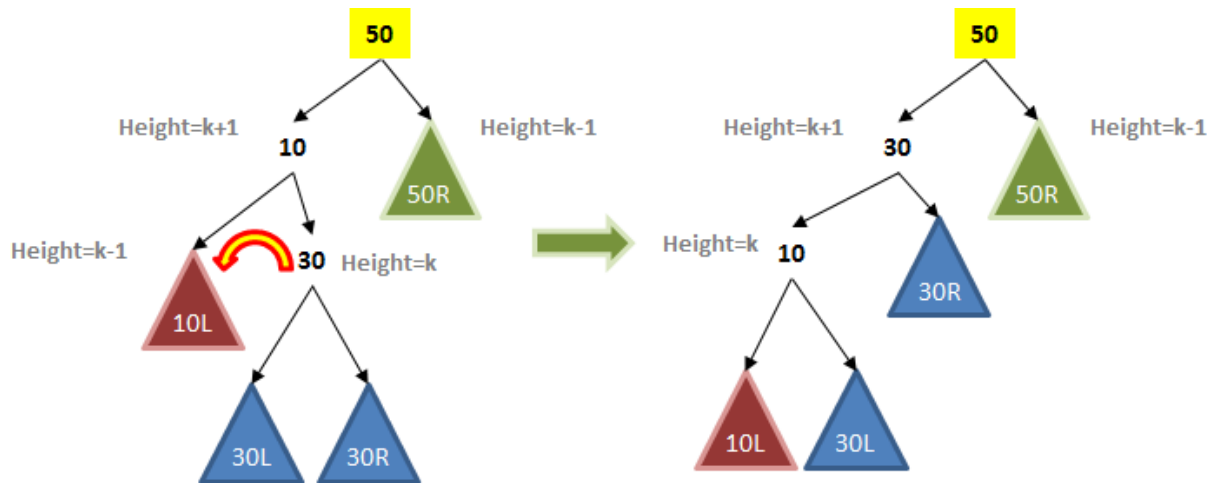
CODE TO TYPE: Modifications to AVLBinaryNode

```
AVLBinaryNode<E> rotateLeft () {  
    AVLBinaryNode<E> newRoot = this.right;  
    AVLBinaryNode<E> grandson = newRoot.left;  
    this.right = grandson;  
    newRoot.left = this;  
  
    computeHeight(this);  
    return newRoot;  
}
```

There are two additional cases that have to be handled. Let's consider the *left-right* case, which suggests that the newly added node is added to the right child of the left child of the unbalanced node. To create this AVL tree, add **50** then **10** to an empty tree. Finally, add **30**:



Once again, the root node is unbalanced, however this time you can't just *Rotate Right* to remedy the situation because the "middle" node, **10** cannot become the root of the tree because its value is smaller than both of the other two values. Fortunately, you can resolve the issue by first completing a *Rotate Left* on the child node **10**; then you'll be able to perform the *Rotate Right* step as described earlier. The image below demonstrates this situation on a larger tree. The After the *Rotate Left* operation, the tree is identical to the earlier tree on which the *Rotate Right* operation was described.



This code handles this *left-right* case. Add the following changes to the end of the **AVLBinaryNode** class:

CODE TO TYPE: Modifications to AVLBinaryNode

```

AVLBinaryNode<E> leftRightRotation () {
    AVLBinaryNode<E> child = left;
    AVLBinaryNode<E> newRoot = child.right;
    AVLBinaryNode<E> grand1 = newRoot.left;
    AVLBinaryNode<E> grand2 = newRoot.right;
    child.right = grand1;
    left = grand2;

    newRoot.left = child;
    newRoot.right = this;

    computeHeight(child);
    computeHeight(this);
    return newRoot;
}
  
```

In reference to the earlier *left-right* diagram, *child* is the **10** node, *newRoot* is the **30** node, *grand1* is the **30L** subtree and *grand2* is the **30R** subtree. The four sequential operations must take place in exactly the order as shown in order to complete the *Rotate Left* then *Rotate Right* operations efficiently. Once that's done, the heights of *child* and *this* are recomputed before *newRoot* is returned as the new root of this subtree. Once again, it is the responsibility of the calling method to recompute the height for *newRoot*.

In exactly the same way, the *right-left* case (not shown here) would first *Rotate Right* before completing the restructuring with a *Rotate Left*. The code for this case is shown below; add this method to the end of the **AVLBinaryNode** class:

CODE TO TYPE: Modifications to AVLBinaryNode

```
AVLBinaryNode<E> rightLeftRotation () {
    AVLBinaryNode<E> child = right;
    AVLBinaryNode<E> newRoot = child.left;
    AVLBinaryNode<E> grand1 = newRoot.left;
    AVLBinaryNode<E> grand2 = newRoot.right;
    child.left = grand2;
    right = grand1;

    newRoot.left = this;
    newRoot.right = child;

    computeHeight(child);
    computeHeight(this);
    return newRoot;
}
```

All that remains now is to write the appropriate **add** method in **AVLBinaryNode**. Add this code to the end of the **AVLBinaryNode** class:

CODE TO TYPE: Modifications to AVLBinaryNode

```
void add (E k) {
    int rc = k.compareTo(key);
    if (rc <= 0) {
        left = add(left, k);
    } else {
        right = add(right, k);
    }
}

AVLBinaryNode<E> add(AVLBinaryNode<E> parent, E k) {
    if (parent == null) {
        return new AVLBinaryNode<E>(k);
    }

    parent = parent.add(k);
    return parent;
}
```

To satisfy the binary search tree property of an AVL tree, the **add** method inserts the new value into either the *left* or *right* subtree. The above code is identical to the **BinaryNode** we wrote earlier in this lesson. However, now you must modify this code to maintain the AVL Property. In each of the rotations described earlier, observe how it was possible for the root of the tree to change during a rotation. For this reason, the **add** method must change to return a potentially new node which becomes the new root of a subtree. Modify **add(E k)** as shown:

CODE TO TYPE: Modifications to add(E k)

```
AVLBinaryNode<E> void add (E k) {
    int rc = k.compareTo(key);
    AVLBinaryNode<E> newRoot = this;
    if (rc <= 0) {
        left = add(left, k);
    } else {
        right = add(right, k);
    }

    computeHeight(newRoot);
    return (newRoot);
}
```

By default, the new root of the subtree to which *k* is added will be *this*, which is the existing root of the subtree. The above code prepares for the rotations by allowing a new root to be returned when a key is added to a subtree. The height of the new root is computed prior to the end of this method; doing so completes each of

the four rotation methods where it was made clear that the invoking method of the rotation would be responsible for computing the height of *newRoot*.

After each invocation of **add(parent, key)** it is possible that *this* has become unbalanced. Insert this new code which handles all four cases:

CODE TO TYPE: Modifications to AVLBinaryNode

```
AVLBinaryNode<E> add (E k) {
    int rc = k.compareTo(key);
    AVLBinaryNode<E> newRoot = this;
    if (rc <= 0) {
        left = add(left, k);
        if (heightDifference(this) == 2) {
            if (k.compareTo(left.key) <= 0) {
                newRoot = rotateRight();
            } else {
                newRoot = leftRightRotation();
            }
        }
    } else {
        right = add(right, k);
        if (heightDifference(this) == -2) {
            if (k.compareTo(right.key) > 0) {
                newRoot = rotateLeft();
            } else {
                newRoot = rightLeftRotation();
            }
        }
    }

    computeHeight(newRoot);
    return newRoot;
}
```

When you add a key to the left subtree for a node, it's possible that the *height difference* for the parent node (that is, *this*) no longer honors the AVL Property, but this only happens once the difference is 2 (because it is acceptable for this value to be -1, 0, or 1). When the *height difference* for the parent *this* node is 2, a rotation must occur to bring this node back into balance. You need to determine whether a single *Rotate Right* is needed (the *left-left* case) or a *Rotate Left* and *Rotate Right* (the *left-right* case). Fortunately a simple condition can determine which is appropriate by comparing the newly added key *k* with the key of the left child. The *left-left* is appropriate if **k <= left.key**, otherwise use the *left-right* case. In both cases, the rotation invocation returns the *newRoot* of the subtree. Similar code is used to handle unbalanced nodes when inserting *k* to the right subtree.


Similarly, when the height of the right subtree for a node exceeds the height of the left subtree, the computed *height difference* is negative; when the difference is -2, the **add** method determines whether the unbalanced node is a *right-right* or *right-left* case by comparing the newly added key with *right.key*.

You are now ready to try some head-to-head comparisons with the existing **TreeSet** implementations in the Java Collections Framework.

Using Collections TreeSet

It is commonly accepted that AVL trees are easier to implement than the *red-black* self-balancing binary trees implemented by **TreeSet**, although *red-black* offers better performance. Let's investigate and find out if this is true. The performance code below evaluates all three types of binary trees—**BinaryTree**, **AVLBinaryTree**, and **TreeSet**—against a single benchmark.

 In the **/performance** source folder, create an **avl** package.

 In the **avl** package, create an **Evaluate** class as shown:

CODE TO TYPE: Evaluate class

```
package avl;

import java.text.*;
import java.util.*;
import binary.*;

public class Evaluate {
    static int numTrials = 100;
    static double m = 1000000;
    static NumberFormat nf;
    public static void main(String[] args) {
        nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(3);

        System.out.println("N    \tB_Time\tB_Find\tA_Time\tA_Find\tT_Time\tT_Find");
        System.out.println("----\t-----\t-----\t-----\t-----\t-----\t-----");
        for (int n = 128; n <= 65536; n *= 2) {
            long totalBSTCreate = 0;
            long totalBSTFind = 0;
            long totalAVLCreate = 0;
            long totalAVLFind = 0;
            long totalTreeSetCreate = 0;
            long totalTreeSetFind = 0;
            for (int t = 0; t < numTrials; t++) {
                ArrayList<Integer> vals = new ArrayList<Integer>();
                for (int i = 0; i < 2*n; i+=2) {
                    vals.add(i);
                }
                Collections.shuffle(vals);
                Integer[] shuffled = vals.toArray(new Integer[]{});

                System.gc();
                AVLBinaryTree<Integer> avlTree = new AVLBinaryTree<Integer>();
                long now = System.nanoTime();
                for (int i : shuffled) {
                    avlTree.add(i);
                }
                totalAVLCreate += (System.nanoTime() - now);

                System.gc();
                now = System.nanoTime();
                for (int i = 0; i < 2*n; i++) {
                    if (avlTree.contains(i) != (i%2 == 0)) {
                        System.err.println("Search fails for BST");
                    }
                }
                totalAVLFind += (System.nanoTime() - now);

                System.gc();
                BinaryTree<Integer> btree = new BinaryTree<Integer>();
                now = System.nanoTime();
                for (int i : shuffled) {
                    btree.add(i);
                }
                totalBSTCreate += (System.nanoTime() - now);

                System.gc();
                now = System.nanoTime();
                for (int i = 0; i < 2*n; i++) {
                    if (btree.contains(i) != (i%2 == 0)) {
                        System.err.println("Search fails for BST");
                    }
                }
                totalBSTFind += (System.nanoTime() - now);

                System.gc();
            }
        }
    }
}
```


```

    TreeSet<Integer> tree = new TreeSet<Integer>();
    now = System.nanoTime();
    for (int i : shuffled) {
        tree.add(i);
    }
    totalTreeSetCreate += (System.nanoTime() - now);

    System.gc();
    now = System.nanoTime();
    for (int i = 0; i < 2*n; i++) {
        if (tree.contains(i) != (i%2 == 0)) {
            System.err.println("Search fails for BST");
        }
    }
    totalTreeSetFind += (System.nanoTime() - now);
}

System.out.println(n + "\t" +
    nf.format(totalBSTCreate/numTrials/m) + "\t" +
    nf.format(totalBSTFind/numTrials/m) + "\t" +
    nf.format(totalAVLCreate/numTrials/m) + "\t" +
    nf.format(totalAVLFind/numTrials/m) + "\t" +
    nf.format(totalTreeSetCreate/numTrials/m) + "\t" +
    nf.format(totalTreeSetFind/numTrials/m));
}
}

```

 Save and run it.

OBSERVE: Output from Evaluate

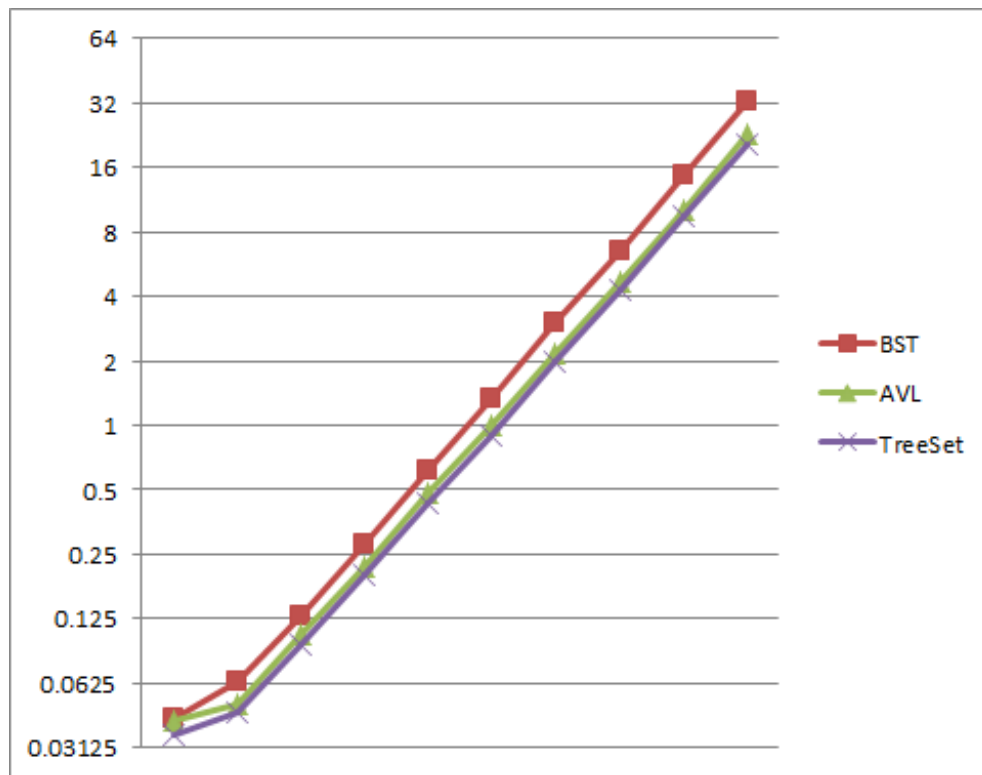
N	B_Time	B_Find	A_Time	A_Find	T_Time	T_Find
128	0.031	0.043	0.054	0.042	0.045	0.036
256	0.044	0.063	0.051	0.050	0.037	0.046
512	0.091	0.129	0.105	0.106	0.076	0.095
1024	0.196	0.276	0.234	0.220	0.166	0.198
2048	0.440	0.616	0.514	0.480	0.359	0.430
4096	0.977	1.332	1.096	1.001	0.773	0.903
8192	2.173	2.994	2.417	2.159	1.682	1.979
16384	4.994	6.526	5.534	4.630	3.837	4.262
32768	11.769	14.662	12.753	10.062	8.844	9.381
65536	26.768	32.584	29.143	22.852	20.299	20.618

To reaffirm the need for rebalancing, comment out the **Collections.shuffle(vals)** line of code in **Evaluate** and reexecute. All too soon, the **BinaryTree** implementation causes a stack overflow error, while the **AVLBinaryTree** and **TreeSet** both continue to function just fine.

OBSERVE: Reexecute comparison when inserting elements in order

N	B_Time	B_Find	A_Time	A_Find	T_Time	T_Find
128	0.105	0.155	0.040	0.039	0.037	0.034
256	0.408	0.580	0.041	0.051	0.034	0.046
512	1.732	2.326	0.085	0.098	0.069	0.122
1024	6.858	9.623	0.191	0.206	0.160	0.198
2048	26.588	36.689	0.380	0.429	0.310	0.400
Exception in thread "main" java.lang.StackOverflowError						
at binary.BinaryNode.add(BinaryNode.java:24)						
...						

If you plot the searching results performance, be sure to do so using a logarithmic scale on the y-axis.



As the above graph shows, the performance graph for all three binary tree structures is a straight line, with naive BST performing the worst. **TreeSet** performs best, but AVL trees are not that far behind. This graph provides further evidence that the searching behavior is $O(n \log n)$.

Lessons Learned

So now you know:

- Constructing an AVL Binary Tree consumes the most time of the three constructions; it's up to 50% slower than the **TreeSet** implementation. This happens because AVL trees must continually rebalance to maintain the AVL Property which is a strong constraint on the structure of the tree. By contrast, the **TreeSet** self-balancing strategy only ensures that the path from the root to the farthest leaf is no more than *twice as long* as the path from the root to the nearest leaf. This relaxed, self-balancing strategy turns out to be more efficient.
- The **TreeSet** code provides the fastest average search times, although AVL trees are not that much slower.
- The naive Binary Tree implementation performs well on randomized data, which might mistakenly lead you to use these BSTs as is for your projects. Be warned that, when the data exhibits any regularity, the construction and search times will rapidly degenerate into $O(n)$ behavior.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Multidimensional Algorithms

Lesson Objectives

When you finish this lesson, you will be able to:

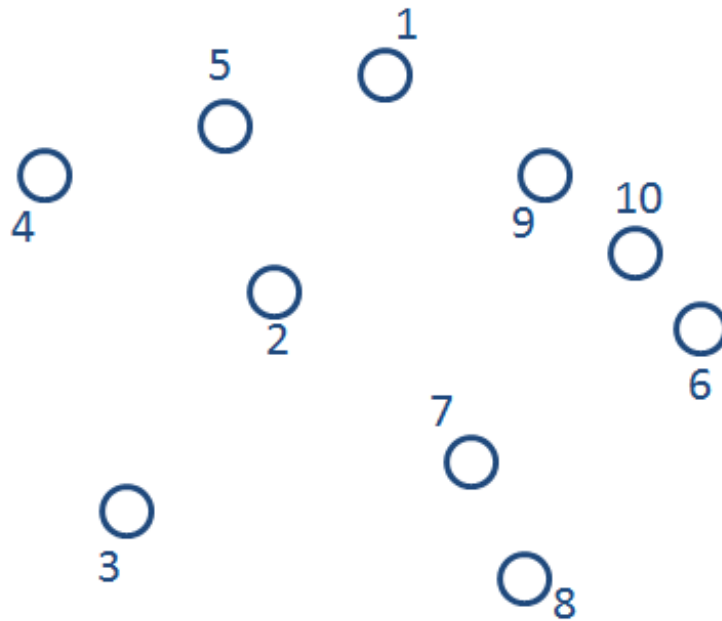
- describe the structure of a k -dimensional tree.
- implement a preorder traversal on any recursive search tree.
- construct a k -d-tree manually after the insertion of a number of points.

A Data Structure For Multidimensional Algorithms

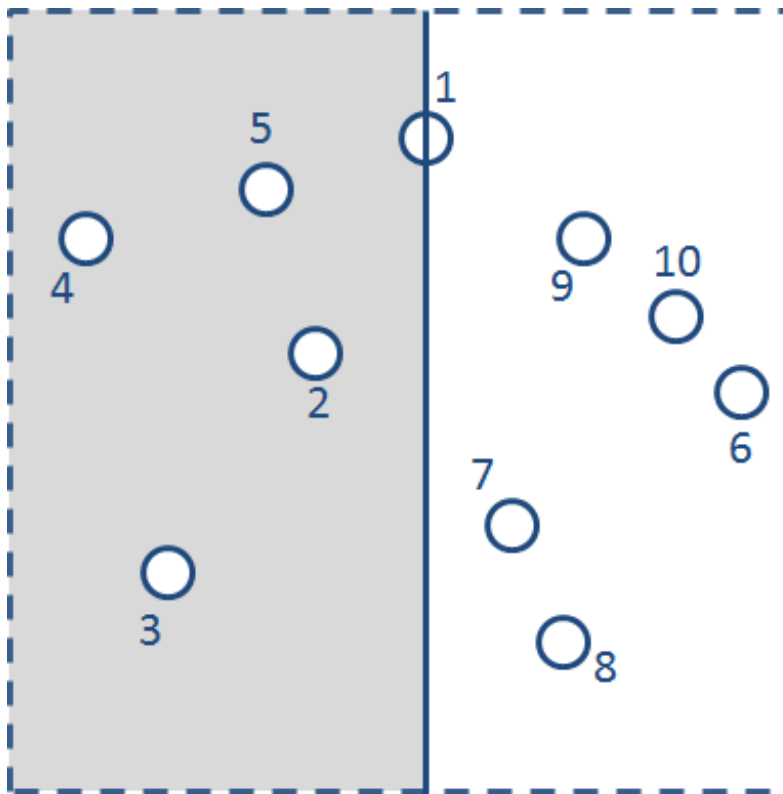
In an earlier lesson, you saw how to use *Binary Array Search* to determine efficiently whether a sorted array contains a given item in $O(\log n)$ time. However, if you have a collection of n Cartesian points (x, y) , there is no comparator function that *completely orders* the points within a one-dimensional array to enable *Binary Array Search* to locate a given point in $O(\log n)$ time. Arrays are simply not powerful enough to support efficient algorithms when data has multiple attributes or *dimensions*.

Real-world data is often represented in tabular form, which makes it well-suited to being stored in an Excel spreadsheet or a database table. Given such a table with n columns, each column can be viewed as a dimension and each row represents an n -dimensional point. Unfortunately, efficiently processing multidimensional data is challenging because there is no way to order all of the rows in a table completely, using all dimensions simultaneously. You've already seen how Binary Trees (when balanced) can store n elements effectively to guarantee $O(\log n)$ performance for searching. In this lesson, we'll apply this concept to storing n elements, each of which has k -dimensions of information. In this lesson we'll use $k=2$ so we can draw two-dimensional images more easily; this approach can also be used for arbitrary dimensions higher than 2.

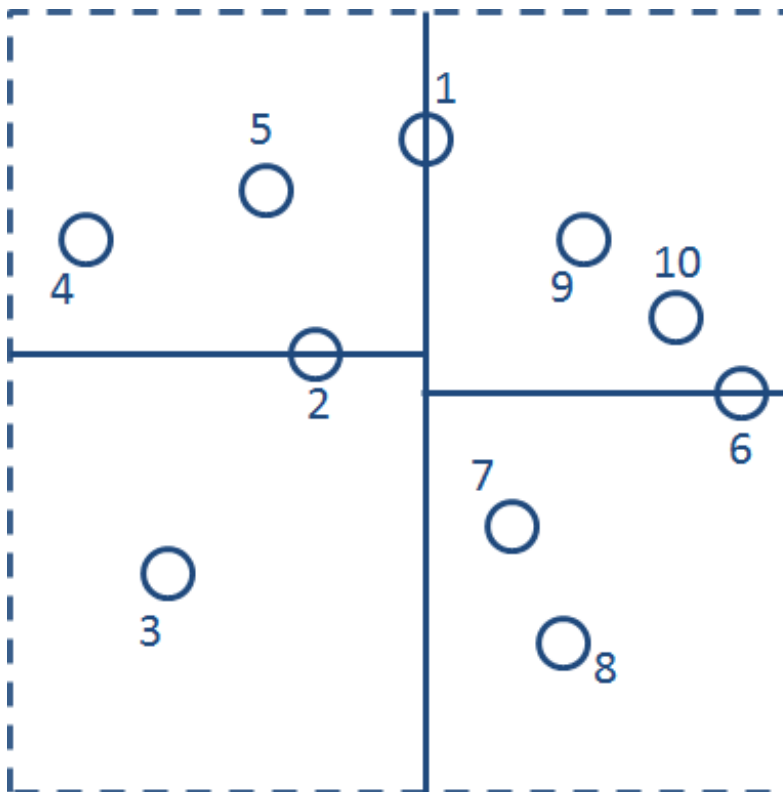
Assume you have a collection of n 2-dimensional points in the Cartesian plane. Here's an example where $n=10$:



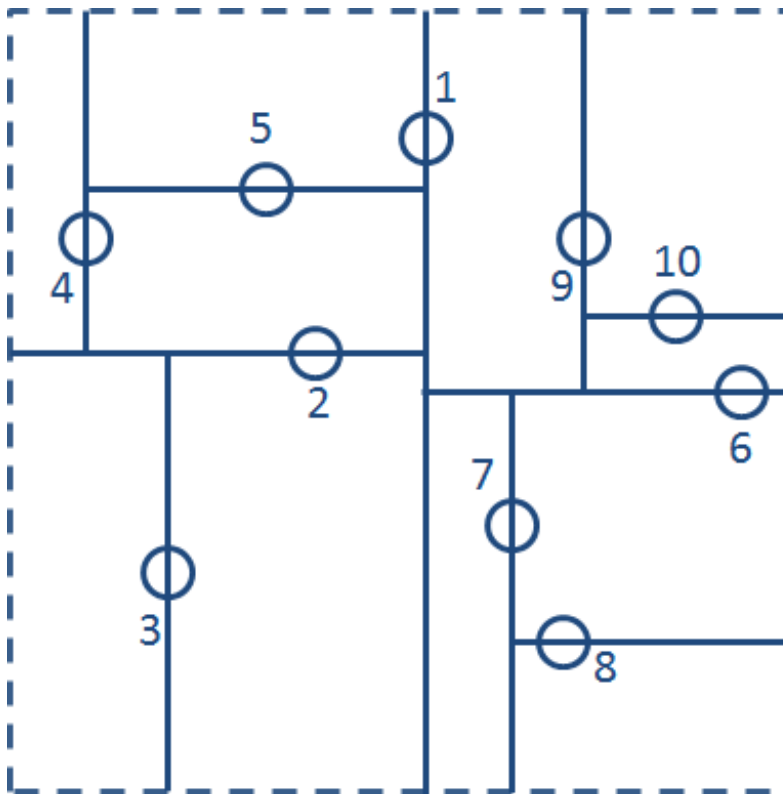
In past lessons, we demonstrated the *Divide and Conquer* approach to sort an array by dividing it into left and right sub-arrays, which were then sorted. But can you split a two-dimensional set of points into a *left* set and a *right* set? If you draw a vertical line through Point 1, you have four points on the left and five points on the right—this looks promising:



The dashed rectangle enclosing all points represents the infinite Cartesian plane [$x_{low} = -\infty$, $y_{low} = -\infty$, $x_{high} = +\infty$, $y_{high} = +\infty$]. Let's associate this region with Point 1 and consider its left sub-region to be the shaded vertical rectangle on the left and the right sub-region to be the vertical rectangle on the right. It doesn't seem possible to continue this division process by adding vertical lines. However, consider dividing these left and right sub-regions by adding two *horizontal* lines, one through Point 2 and the other through Point 6. These horizontal lines do not extend across the whole plane, but rather divide the respective sub-regions into quadrants:.



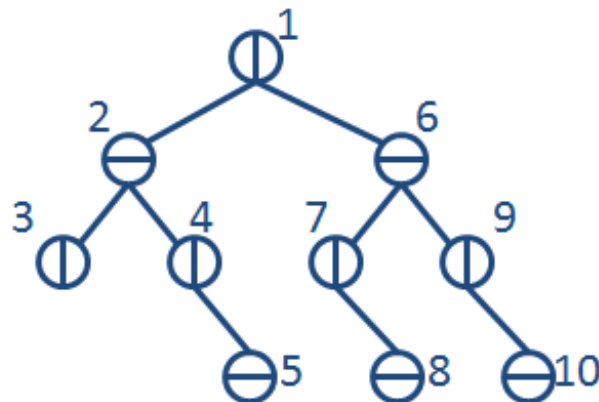
Each of these four quadrants contain just 1 or 2 points. This process of using alternating vertical and horizontal lines to subdivide the set of points can be repeated recursively within each quadrant. Here you can see the final subdivision of these ten points:



Let's design a data structure to represent the partitioned information. A *kd-tree* (short for *k-dimensional tree*) is a recursive binary tree structure with n nodes, each of which contains:

- a 2-dimensional point in the collection.
- a partition direction (either *vertical* or *horizontal*).
- an associated rectangular subregion of the two-dimensional plane.
- two child node links (named *below* and *above*).

Given the above 10 points, here is its corresponding binary kd-tree:



Point 1 partitions the maximum region into two halves. The child sub-tree rooted at the node for Point 2 contains all points to the left of the vertical line partitioning the region through Point 1. Similarly, the child sub-tree rooted at the node for Point 6 contains all points to the right of the vertical line partitioning the region through Point 1. Instead of using the terms "left" and "right" to refer to child nodes (which would only apply for vertical partitioning), kd-trees use the concept of "below" and "above." The nodes in the child sub-tree rooted at the node for Point 2 all represent points with an x-coordinate that is smaller than ("below") the x-coordinate of Point 1. Similarly, the nodes in the child sub-tree rooted at the node for Point 6 all represent points with an x-coordinate that is larger than or equal to ("above") the x-coordinate of Point 1.

Structurally, the tree is a classic binary search tree, but each level in the tree alternates the partition directions of the nodes in the tree.

Let's get started by defining a class to represent the regions partitioned by the kd-tree. We can't just use the `java.awt.Rectangle` class because that defines rectangles using widths and heights.



Create a new Java Project named **Multidimension** and assign it to the **Java6_Lessons** working set.



In your **Multidimension** project `/src` source folder, create a **kd** package.



In the **kd** package, create a **Region** class as shown:

CODE TO TYPE: Region class

```
package kd;

public class Region {
    int x_min;
    int x_max;
    int y_min;
    int y_max;

    public Region (int x1, int y1, int x2, int y2) {
        x_min = x1;
        y_min = y1;
        x_max = x2;
        y_max = y2;
    }

    public Region (Region r) {
        this(r.x_min, r.y_min, r.x_max, r.y_max);
    }

    static final int    minValue = Integer.MIN_VALUE;
    static final int    maxValue = Integer.MAX_VALUE;
    static final Region max = new Region(minValue, minValue, maxValue, maxValue);
}
```

We assume all coordinate points are integer values and all attributes are accessible within the **kd** package to simplify the programming of the algorithm. The **max** region represents the maximal region possible. Now that we have a definition for the regions, we can design the class to represent the nodes in the kd-tree.



In the **kd** package, create a **KDNode** class as shown:

CODE TO TYPE: KDNode class

```
package kd;

import java.awt.Point;

public class KDNode {
    final Point point;
    final int direction;
    Region region;
    KDNode above;
    KDNode below;

    public static final int HORIZONTAL = 0;
    public static final int VERTICAL = 1;

    public KDNode(Point p, int dir, Region r) {
        this.point = new Point (p);
        this.direction = dir;

        this.region = new Region(r);
    }

    public KDNode(Point p, int dir) {
        this (p, dir, Region.max);
    }
}
```

Each **KDNode** object represents a node in a kd-tree and stores three pieces of information as described earlier: a point, a region, and a partition direction. **KDNode** defines two constants that differentiate between **HORIZONTAL** and **VERTICAL** partitioning. By default, each **KDNode** object is associated with the maximum region available, as defined by the **Region** class. The values for **HORIZONTAL** and **VERTICAL** are chosen such that $1-d$ gives the opposite direction of d .

For this node to define a recursive search tree, it must define children nodes. In this case, each **KDNode** records two children, one "below" the partitioning line and the other "above" the partitioning line. The notion of a child node being "above" is relative to the direction of the **KDNode**. When the partitioning for a node is **HORIZONTAL**, the child node "above" a node is found vertically above the horizontal partitioning line with a y-coordinate that divides the node's region. When the partitioning for a node is **VERTICAL**, the child node "below" a node is found to the left of the vertical partitioning line with an x-coordinate that divides the node's region.

The kd-tree rooted at a **KDNode** n defines a binary search tree because all points in the sub-tree rooted by the "below" child will be "below" the partitioning line for node n , while all points in the sub-tree rooted by the "above" child will be "above" the partitioning line for node n . To make this happen, you need to define some helper methods. Add these methods to the end of **KDNode**:

CODE TO TYPE: Helper methods to add to KDNode

```
public boolean isBelow(Point p) {
    if (direction == VERTICAL) {
        return p.x < point.x;
    } else {
        return p.y < point.y;
    }
}

public boolean isAbove(Point p) {
    if (direction == VERTICAL) {
        return p.x >= point.x;
    } else {
        return p.y >= point.y;
    }
}
```

These methods help determine, for a given **KDNode**, whether a point p is "below" or "above" its partitioning line. You'll need one more method that returns a properly configured child node for a given **KDNode**. The trick is to compute the region associated with the child node based on the region associated with its existing parent node:

CODE TO TYPE: Helper method for to add to KDNode

```
KDNode createChild (Point p, boolean below) {
    Region r = new Region (region);
    if (direction == VERTICAL) {
        if (below) {
            r.x_max = point.x;
        } else {
            r.x_min = point.x;
        }
    } else {
        if (below) {
            r.y_max = point.y;
        } else {
            r.y_min = point.y;
        }
    }
    return new KDNode(p, 1-direction, r);
}
```

The child node must have the opposite partitioning of its parent; that's why **1-direction** is used as the direction of the child node. The point to associate with the child, *p* is passed to the **KDNode** constructor. The challenge is to compute the sub-region associated with the newly created child node. There are four cases to consider as implemented in the above code—we'll just explain one. If a node is horizontal, its *point* partitions its rectangular region into a region "above" the y-coordinate of its point and a region "below" the y-coordinate. Invoking **createChild(p, false)** on a *horizontal* node *n* means that the region for the child **KDNode** must be "trimmed" to be a proper subset of the current node's region. To do this, the above code sets the **y_min** of the child's region *r*. The other three cases are similar.

Now we can implement a method to add a point to a given kd-tree rooted at a **KDNode**:

CODE TO TYPE: Create add(Point) method in KDNode


```
public void add (Point p) {
    if (p.equals(point)) { return; }

    if (isBelow(p)) {
        if (below == null) {
            below = createChild (p, true);
        } else {
            below.add(p);
        }
    } else {
        if (above == null) {
            above = createChild (p, false);
        } else {
            above.add(p);
        }
    }
}
```

The kd-tree implementation here abides by Set semantics, as described earlier in this course. This means that the same point cannot exist more than once in a given kd-tree. When the **add** method returns without throwing an Exception, the point is guaranteed to be added to the kd-tree.

If the point to be added is below its partitioning line, the **add** method either creates a new node to represent the "below" child (if that node doesn't already exist) or adds the point to the kd-tree rooted at the "below" child. The logic for the "above" case is similar.

We've completed the **KDNode** class; now it's time to design the class to represent the kd-tree.

 In the **kd** package, create a **KDTree** class as shown:

CODE TO TYPE: KDTree class

```
package kd;

import java.awt.Point;

public class KDTree {
    KNode root;

    public KDTree() {
        root = null;
    }

    public void add (Point value) {
        if (root == null) {
            root = new KNode(value, KNode.VERTICAL);
        } else {
            root.add(value);
        }
    }
}
```


A **KDTree** object is defined by a root **KNode**. This class offers an **add** method to add points to the kd-tree. If the kd-tree is empty, it creates a new root node whose partitioning by default (arbitrarily) is **VERTICAL**.

Traversing a kd-tree

You have enough code written to construct a kd-tree from a set of points. The hard part is figuring out whether the code is working because the binary tree structure is stored in memory and it can't simply be printed out to the console. You need to write a *traversal* routine that walks through the kd-tree in a specific order; if the kd-tree is constructed properly, the output will be correct. There are many ways to traverse a recursive tree. Using the example presented at the beginning of this lesson, a *pre-order* traversal would:

1. process Point 1.
2. recursively process all points to the left of the partitioning line through Point 1.
3. recursively process all points to the right of the partitioning line through Point 1.

The applet class below interactively draws a kd-tree whenever a point is added because the mouse was pressed. At last, you have something to run for your efforts!

 In the **kd** package, create a **KDApplet** class as shown:

CODE TO TYPE: KDApplet class

```
package kd;

import java.awt.*;
import java.awt.event.*;

public class KDApplet extends java.applet.Applet {
    KDTree tree = new KDTree();

    int toAWT(int y) {
        if (y == Region.maxValue) { return 0; }
        int awty = getHeight();
        if (y != Region.minValue) { awty -= y; }
        return awty;
    }

    int toCartesian(int awty) { return getHeight() - awty; }

    public void init() {
        setSize(400,400);
        addMouseListener (new MouseAdapter() {
            public void mouseClicked(MouseEvent me) {
                Point pt = new Point (me.getX(), toCartesian(me.getY()));
                tree.add(pt);
                repaint();
            }
        });
    }

    public void paint(Graphics g) {
        if (tree.root == null) {
            g.drawString("Click to add points", 150, 200);
        } else {
            visit(g, tree.root);
        }
    }

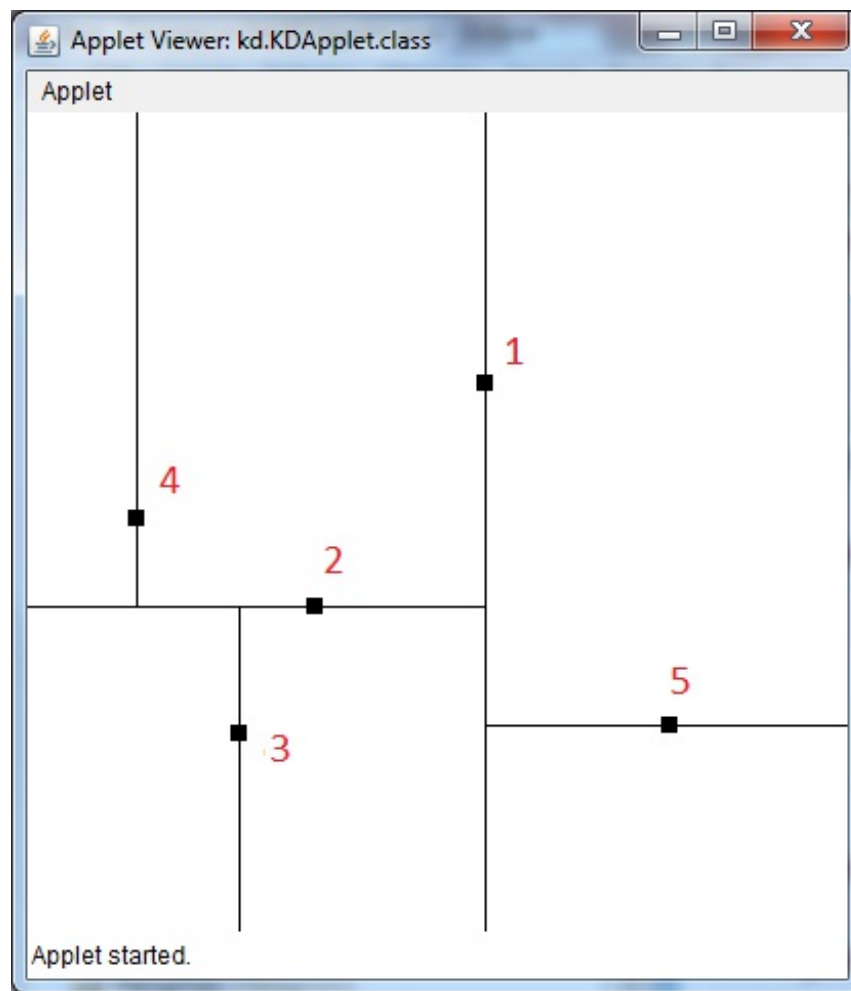
    void drawPartition (Graphics g, Region r, Point p, int type) {
        if (type == KDNode.VERTICAL) {
            g.drawLine(p.x, toAWT(r.y_min), p.x, toAWT(r.y_max));
        } else {
            int xlow = r.x_min;
            if (r.x_min == Region.minValue) { xlow = 0; }
            int xhigh = r.x_max;
            if (r.x_max == Region.maxValue) { xhigh = getWidth(); }
            g.drawLine(xlow, toAWT(p.y), xhigh, toAWT(p.y));
        }
        g.fillRect(p.x - 4, toAWT(p.y) - 4, 8, 8);
    }

    void visit (Graphics g, KDNode n) {
        if (n == null) { return; }
        drawPartition(g, n.region, n.point, n.direction);

        visit (g, n.below);
        visit (g, n.above);
    }
}
```



Save and run it. Add points to the kd-tree by clicking the mouse at different places in the Applet window. The partitioning direction alternates as each point is added. We've annotated the screenshot below using red numbers to identify the order the points were added (these don't appear in the running program):



Let's take a closer look at this code:

OBSERVE: Setting Up The Applet

```
package kd;

import java.awt.*;
import java.awt.event.*;

public class KDApplet extends java.applet.Applet {
    KDTree tree = new KDTree();

    ...

    int toCartesian(int awty) { return getHeight() - awty; }

    public void init() {
        setSize(400,400);
        addMouseListener (new MouseAdapter() {
            public void mouseClicked(MouseEvent me) {
                Point pt = new Point (me.getX(), toCartesian(me.getY()));
                tree.add(pt);
                repaint();
            }
        });
    }

    public void paint(Graphics g) {
        if (tree.root == null) {
            g.drawString("Click to add points", 150, 200);
        } else {
            visit(g, tree.root);
        }
    }

    ...
}
```

The Java graphics coordinate system is different from the Cartesian plane. Specifically, the upper-left corner of the window is coordinate (0,0). From left to right, the x-coordinate increases, as it does with Cartesian coordinates. However, when moving from top to bottom, the y-coordinate increases, which is opposite of Cartesian coordinates. The **toCartesian** helper method converts the y-coordinate of an Abstract Windowing Toolkit (AWT) point into Cartesian coordinates. This method is used within the **MouseAdapter** that responds to mouse-click events by **adding a point to the kd-tree**. After each point is added, the **applet is repainted**. The applet repaints itself by traversing the kd-tree using the **visit** method.

OBSERVE: Pre-order traversal of the kd-tree

```
void visit (Graphics g, KDNode n) {
    if (n == null) { return; }
    drawPartition(g, n.region, n.point, n.direction);

    visit (g, n.below);
    visit (g, n.above);
}
```

All nodes in the tree will be visited by the above method. This declares a pre-order traversal because it first "visits" the given node by **drawing its partitioning line on the screen**. Then it **visits its child nodes**, first the ones below it and then the ones above it. The base case of the recursion **stops when asked to visit a null node**.

The real drawing work is done in **drawPartition**:

OBSERVE: Draw Partitioning Line For KNode

```
int toAWT(int y) {
    if (y == Region.maxValue) { return 0; }
    int awty = getHeight();
    if (y != Region.minValue) { awty -= y; }
    return awty;
}

void drawPartition (Graphics g, Region r, Point p, int type) {
    if (type == KNode.VERTICAL) {
        g.drawLine(p.x, toAWT(r.y_min), p.x, toAWT(r.y_max));
    } else {
        int xlow = r.x_min;
        if (r.x_min == Region.minValue) { xlow = 0; }
        int xhigh = r.x_max;
        if (r.x_max == Region.maxValue) { xhigh = getWidth(); }
        g.drawLine(xlow, toAWT(p.y), xhigh, toAWT(p.y));
    }
    g.fillRect(p.x - 4, toAWT(p.y) - 4, 8, 8);
}
```

To draw the partitioning line for a vertical node, one only needs to **draw a vertical line through the x-coordinate $p.x$ using the y-coordinates from the associated region for the node**. However, since these regions may reflect partially infinite regions in the plane, you need the **toAWT** helper method that converts a (potentially infinite) Cartesian y-coordinate into its AWT counterpart. **If the coordinate is the maximum allowed value for a Region, the y-coordinate is 0**, because that's the topmost coordinate in the AWT coordinate system. If the Cartesian y-coordinate is the minimum allowed value for a Region, the counterpart y-coordinate is simply **the height of the Applet window**. Otherwise, the **toAWT** method **converts the y-coordinate based on its distance from the bottom of the window** (the **height of the applet window**).

Using kd-trees to Search for Points

Now that we've demonstrated the proper construction of a kd-tree, there are two kind of queries we'd like to support:

- Contains—does the kd-tree contain a given point P.
- Selection—select the points in the kd-tree that are contained within a query rectangle.

To find whether a given point exists in the tree, you can use the partitioning lines associated with each node to direct the search, either to the child node below the line or the child node above the line. Add this code to the end of **KDTree**:

CODE TO TYPE: Add find methods to KDTree

```
public KNode find(Point p) {
    return find(root, p);
}

KNode find (KNode node, Point p) {
    if (node == null) { return null; }
    if (node.point.distance(p) < 5) { return node; }

    if (node.isBelow(p)) {
        return find(node.below, p);
    } else {
        return find(node.above, p);
    }
}
```

The **find(node,p)** method must choose whether to investigate the child below or above, based on the partitioning line. The recursive method will eventually terminate at a leaf node or when the node's Euclidian distance to p is smaller than 5 pixels.

To highlight the point over which the cursor moves, modify **KDApplet** as shown:

CODE TO TYPE: Modifications to KDApplet

```
package kd;

import java.awt.*;
import java.awt.event.*;

public class KDApplet extends java.applet.Applet {
    KDTree tree = new KDTree();
    KNode match = null;
    boolean redraw = false;

    int toAWT(int y) {
        if (y == Region.maxValue) { return 0; }
        int awty = getHeight();
        if (y != Region.minValue) { awty -= y; }
        return awty;
    }

    int toCartesian(int awty) { return getHeight() - awty; }

    public void init() {
        setSize(400,400);
        addMouseListener (new MouseAdapter() {
            public void mouseClicked(MouseEvent me) {
                Point pt = new Point (me.getX(), toCartesian(me.getY()));
                tree.add(pt);
                repaint();
            }
        });

        addMouseMotionListener (new MouseAdapter() {
            public void mouseMoved(MouseEvent me) {
                Point pt = new Point (me.getX(), toCartesian(me.getY()));
                match = tree.find(pt);
                if (match != null) {
                    redraw = true;
                    Graphics g = getGraphics();
                    g.setColor(Color.RED);
                    g.fillRect(match.point.x - 4, toAWT(match.point.y) - 4, 8, 8);
                    g.dispose();
                } else {
                    if (redraw) {
                        repaint();
                        redraw = false;
                    }
                }
            }
        });
    }

    public void paint(Graphics g) {
        if (tree.root == null) {
            g.drawString("Click to add points", 150, 200);
        } else {
            visit(g, tree.root);
        }
    }

    void drawPartition (Graphics g, Region r, Point p, int type) {
        if (type == KNode.VERTICAL) {
            g.drawLine(p.x, toAWT(r.y_min), p.x, toAWT(r.y_max));
        } else {
            int xlow = r.x_min;
            if (r.x_min == Region.minValue) { xlow = 0; }
            int xhigh = r.x_max;
            if (r.x_max == Region.maxValue) { xhigh = getWidth(); }
            g.drawLine(xlow, toAWT(p.y), xhigh, toAWT(p.y));
        }
    }
}
```



```

    }
    g.fillRect(p.x - 4, toAWT(p.y) - 4, 8, 8);
}

void visit (Graphics g, KDNode n) {
    if (n == null) { return; }
    drawPartition(g, n.region, n.point, n.direction);

    visit (g, n.below);
    visit (g, n.above);
}
}

```

Let's take a closer look at the changes:

OBSERVE:


```

KDNode match = null;
boolean redraw = false;
...
addMouseListener (new MouseAdapter() {
    public void mouseMoved(MouseEvent me) {
        Point pt = new Point (me.getX(), toCartesian(me.getY()));
        match = tree.find(pt);
        if (match != null) {
            redraw = true;
            Graphics g = getGraphics();
            g.setColor(Color.RED);
            g.fillRect(match.point.x - 4, toAWT(match.point.y) - 4, 8, 8);
            g.dispose();
        } else {
            if (redraw) {
                repaint();
                redraw = false;
            }
        }
    }
});

```

Two new fields are added. **match** records the last point in the kd-tree matched by the cursor; **redraw** determines when to redraw the image upon matching the cursor.

The real logic occurs in the **MouseListener** implementation, which activates with each move of the mouse. It **converts the mouse point into Cartesian coordinates** and then **tries to find the point within the kd-tree**. **If a point is found**, it is **filled in red**. **If the mouse moves and there is no matching point**, the **entire kd-tree must be refreshed**.

 Save and run it; there is a flicker effect. We can use a technique called "double buffering" to eliminate most of the flickering. Make these changes:

CODE TO TYPE: Updates to KDApplet

```
package kd;

import java.awt.*;
import java.awt.event.*;

public class KDApplet extends java.applet.Applet {
    KDTree tree = new KDTree();
    KDNode match = null;
boolean redraw = false;
    Image bufferImage;
    Graphics bufferGraphics;

    int toAWT(int y) {
        if (y == Region.maxValue) { return 0; }
        int awty = getHeight();
        if (y != Region.minValue) { awty -= y; }
        return awty;
    }

    int toCartesian(int awty) { return getHeight() - awty; }

    public void init() {
        setSize(400,400);
        addMouseListener (new MouseAdapter() {
            public void mouseClicked(MouseEvent me) {
                Point pt = new Point (me.getX(), toCartesian(me.getY()));
                tree.add(pt);
                redraw();
                repaint();
            }
        });

        addMouseMotionListener (new MouseAdapter() {
            public void mouseMoved(MouseEvent me) {
                Point pt = new Point (me.getX(), toCartesian(me.getY()));
                KDNode newMatch = tree.find(pt);
                if (match != newMatch) {
                    match = newMatch;
                    redraw();
                    if (match != null) {
                        bufferGraphics.setColor(Color.RED);
                        bufferGraphics.fillRect(match.point.x - 4, toAWT(match.point.y) - 4,
8, 8);
                        bufferGraphics.setColor(Color.BLACK);
                    }
                    repaint();
                }
match = tree.find(pt);
if (match != null) {
    redraw = true;
    Graphics g = getGraphics();
    g.setColor(Color.RED);
    g.fillRect(match.point.x - 4, toAWT(match.point.y) - 4, 8, 8);
    g.dispose();
} else {
    if (redraw) {
        repaint();
        redraw = false;
    }
}
            }
        });
    }

    public void paint(Graphics g) {
        if (bufferImage == null) {
```

```

        bufferImage = createImage(getWidth(), getHeight());
        bufferGraphics = bufferImage.getGraphics();
    }

    if (tree.root == null) {
        g.drawString("Click to add points", 150, 200);
    } else {
        g.drawImage(bufferImage, 0, 0, this);
visit(g, tree.root);
    }
}

void redraw() {
    bufferGraphics.clearRect(0, 0, getWidth(), getHeight());
    visit(bufferGraphics, tree.root);
}

void drawPartition (Graphics g, Region r, Point p, int type) {
    if (type == KDNode.VERTICAL) {
        g.drawLine(p.x, toAWT(r.y_min), p.x, toAWT(r.y_max));
    } else {
        int xlow = r.x_min;
        if (r.x_min == Region.minValue) { xlow = 0; }
        int xhigh = r.x_max;
        if (r.x_max == Region.maxValue) { xhigh = getWidth(); }
        g.drawLine(xlow, toAWT(p.y), xhigh, toAWT(p.y));
    }
    g.fillRect(p.x - 4, toAWT(p.y) - 4, 8, 8);
}

void visit (Graphics g, KDNode n) {
    if (n == null) { return; }
    drawPartition(g, n.region, n.point, n.direction);

    visit (g, n.below);
    visit (g, n.above);
}
}

```

The key to flicker-free graphics is to have all drawing performed in an off-screen image and then have the **paint** method draw that image to the screen. The offscreen image *bufferImage* is created the first time **paint** is invoked. A newly added **redraw** method performs all drawing within the *bufferGraphics* object associated with this offscreen image:

OBSERVE:

```

public void mouseMoved(MouseEvent me) {
    Point pt = new Point (me.getX(), toCartesian(me.getY()));
    KDNode newMatch = tree.find(pt);
    if (match != newMatch) {
        match = newMatch;
        redraw();
        if (match != null) {
            bufferGraphics.setColor(Color.RED);
            bufferGraphics.fillRect(match.point.x - 4, toAWT(match.point.y) - 4,
8, 8);
            bufferGraphics.setColor(Color.BLACK);
        }
        repaint();
    }
}

```

The real logic occurs within the modified **mouseMoved** method. **If there is a new match found that differs from the last (non-null) match, the point is redrawn in red in the offscreen buffer;** otherwise the **repaint** never occurs.



Save and run this code; there's a noticeable difference in performance. You can now move the mouse around rapidly over the points in the kd-tree and see the highlighted red points appear and disappear.

A kd-tree can support *Rectangle Queries* that efficiently return the set of points contained within a two-dimensional rectangle. A kd-tree can also support *Nearest Neighbor Queries* that look for the closest point in the kd-tree to a query point (x,y) . For more details on kd-trees, you may want to refer to the [Algorithms in a Nutshell](#) book.

Lessons Learned

- **Use a recursive structure to partition an n-dimensional set of points:** In all examples so far, you have seen recursion that separates an aggregate into a "left" and a "right" side. By alternating dimensions, the kd-tree concept can accommodate n-dimensional data. This becomes really exciting with high-dimensional data because the rectangular and nearest neighbor queries can perform efficiently.
- **Use recursive traversal to visit every element in a recursive tree:** The *pre-order* traversal is introduced in this lesson. The concept applies to any recursive data structure. In general, there are three primary traversal orderings: pre-order, post-order, and in-order. Each fully traverses all elements in the tree, but does so in a different order.

Project

Modify the existing **KDApplet** class to display the *pre-order* number associated with each point. The *pre-order* number is determined in a pre-order traversal of a binary tree. If you refer to the sample screenshot image shown earlier in the lesson, the points are drawn with numbers in red signifying the pre-order numbering. As new points are added, the numbers will change.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Mathematical Algorithms and Floating Point Computations

Lesson Objectives

When you finish this lesson, you will be able to:

- explain the structure of floating-point numbers according to the IEEE standard.
 - demonstrate code techniques to mitigate rounding errors in floating point computations.
-

Mathematical Algorithms and Floating Point Computations

Computers are finite machines that are designed to perform basic computations on values stored in registers by a Central Processing Unit (CPU). The size of these registers has evolved as computer architectures have grown from the popular 8-bit Intel processors in the 1970s to today's widespread acceptance of 64-bit architectures. Computations over integer-based values (such as Booleans, 8-bit shorts, and 16- and 32-bit integers) have traditionally been the most efficient computations performed by the processor. Most CPUs today are fully integrated with a Floating Point Unit (FPU) that supports the IEEE Standard for Binary Floating-Point Arithmetic ([IEEE 754](#)). This means that the performance of floating-point computations is often more efficient than their integer counterparts.

A floating-point number is a finite representation designed to approximate a real number with a binary representation that may be infinite. As you begin to experiment with floating-point numbers, you may be surprised at some of the results.



Create a new Java Project named **Mathematical** and assign it to the **Java6_Lessons** working set.



In your **Mathematical** project **/src** source folder, create a **numeric** package.



In the **numeric** package, create a **Floating** class:

CODE TO TYPE: Floating class

```
package numeric;

public class Floating {
    public static void main(String[] args) {
        float total = 3.9f;
        while (total > 3.7) {
            System.out.println(total);
            total = total - 0.01f;
        }

        float f = 3.88f - 0.01f;
        if (f == 3.87f) {
            System.out.println ("Same");
        }

        int bits = Float.floatToIntBits(3.88f);
        int signBit = 0;
        if ((bits & 0x80000000) != 0) { signBit = 1; }
        System.out.println ("3.88f is " + Integer.toHexString(bits));
        System.out.println ("3.88f is " + Integer.toBinaryString(bits));
        System.out.println ("[s][eeeeeeee][xxxxxxxxxxxxxxxxxxxxxxxxxxxx]");
        System.out.print ("[" + signBit + "]");
        System.out.print ("[" + pad((bits & 0x7f800000) >> 23, 8) + "]");
        System.out.print ("[" + pad((bits & 0x007fffff), 23) + "]");
    }

    static String pad (int value, int len) {
        StringBuilder sb = new
            StringBuilder(Integer.toBinaryString(value));
        while (sb.length() < len) {
            sb.insert(0, '0');
        }
        return sb.toString();
    }
}
```



Save and run it. This code prints a table of values decreasing by 0.01 each time:

OBSERVE: Floating output

```
3.9
3.89
3.88
3.8700001
3.8600001
3.8500001
3.8400002
3.8300002
3.8200002
3.8100002
3.8000002
3.7900002
3.7800002
3.7700002
3.7600002
3.7500002
3.7400002
3.7300003
3.7200003
3.7100003
3.7000003
3.88f is 407851ec
3.88f is 1000000011110000101000111101100
[s] [eeeeeeee] [xxxxxxxxxxxxxxxxxxxxxxxxxxxx]
[0] [10000000] [11110000101000111101100]
```

Our program starts as expected, but shortly thereafter some of the the computations introduce an error. What's so special about subtracting **0.01** from **3.88**? Let's take a closer look at how Java represents **3.88**, using the IEEE Floating-Point standard. The 32 bits used for this value are represented in hexadecimal notation as **0x407851ec**. These bits are numbered from 31 (on the left) to 0 (farthest to the right) and encode this information.

3.88f is represented in 32 bits as **407851ec**, as explained by the floating point standard. Any number represented in floating-point is equal to $m * 2^{exp}$. Bit 31 (the bit that is selected by the mask **0x80000000**) indicates whether the value is positive or negative. Bits 30-23 (the eight bits that are selected by the mask **0x7f800000**) represent the exponent, *exp*. Bits 22-0 (the twenty-three bits that are selected by the mask **0x007fffff**) represent the mantissa, *m*, of the floating-point number. In the output above, **Integer.toBinaryString** prints only 31 binary characters and doesn't print the sign bit. The 32 bits are arranged from left to right as shown above, which produces the representation **407851ec** as a 32-bit Java floating point number.

You can determine which power of two to use by interpreting the exponent bits as a positive number and then subtracting a *bias* from the positive number. For a float, the bias is 126. Given the encoding of **0x80 = 128**, the exponent is **128 - 126 = 2**.

To maintain the most precision in the representation, the mantissa is always normalized so its leftmost digit is a 1; this means that digit can actually be omitted from the representation to increase the precision of the final number by one bit.

To interpret the mantissa, remember that it is a binary fraction computed as the sum of fractional powers of two. Here mantissa = **.[1]11110000101000111101100**, which shows that the first *implied* digit is a one. Expanding this computation results in this:

mantissa = $[1/2] + 1/4 + 1/8 + 1/16 + 1/32 + 1/1024 + 1/4096 + 1/65536 + 1/131072 + 1/262144 + 1/524288 + 1/2097152 + 1/4194304$.

If you compute the above sum using a calculator, it is exactly **0.97000000286102294921875**.

The final value = **0.97000000286102294921875 * 2²** which equals the exact number **3.880000011444091796875**. So, the actual error of this representation of **3.88f** is on the order of **0.0000001** or 1 in 10,000,000.

Note

If the first bit in the mantissa is implied, how is the value 1/2 represented in floating point? Well, the mantissa must be zero (since the 1st bit is implied), the sign is 0 and the exponent must be 0, so you need to add the bias of 126 to see that the encoding is **3F000000**.

Working with floating point computations can introduce small *rounding errors* into your solutions. In this lesson, you'll

solve a common mathematical problem and learn how to manage rounding errors in your implementation.

Note

Java has two floating-point number formats: **float** uses 32 bits, while **double** uses 64 bits. For the rest of this lesson, all computation will be done using **double** values.

Gauss Jordan Elimination

Given a set of m linear equations of m variables each, is there a unique solution for these variables? For example, let's say you are given these three equations over the three variables: x, y, z :

$x + 3y + 5z = 9$	E1
$2x + 7y + 2z = 2$	E2
$x + y + 4z = 2$	E3

You could use a *trial and error* approach, guessing values for these variables to see if they satisfy all equations simultaneously. Instead, consider an approach that systematically determines the values. For example, you could transform the equations by adding two or more of the equations together, trying to eliminate a variable.

If you subtract equation **E1** twice from **E2** and once from **E3**, the equations become simpler because you are able to eliminate the x variable from the second and third equations:

$x + 3y + 5z = 9$	E1
$y - 8z = -16$	E2
$-2y - z = -7$	E3

Now just add equation **E2** twice to equation **E3**:

$x + 3y + 5z = 9$	E1
$y - 8z = -16$	E2
$-17z = -39$	E3

Given these three equations, you can solve for $z = 39/17$. Insert this value back into the second equation and you can compute $y = -16 + 312/17 = 40/17$. Finally, insert these values of y and z back into the first equation to yield $x = 9 - 120/17 - 195/17 = -162/17$. So, the final solution is $(x = -162/17, y = 40/17, z = 39/17)$. Instead of substituting values, we could have repeated the elimination steps such that each of the three equations above has a single variable. In fact, that's the *Gauss Jordan Elimination* algorithm attempts to do that.


A problem instance is represented by an $m \times (m+1)$ matrix where there are m variables and m equations. In the above example, $m=3$. Note that each equation has $m+1$ values because the last value is the constant value equal to the sum of the variable terms. There is no singular solution when there are fewer than m equations with m variables. You have enough information to write the pseudocode for Gauss Jordan Elimination of a set of linear equations represented by matrix A , where $A[r][c]$ is the c^{th} coefficient in the r^{th} row. Row r is defined in the range $0 \dots m-1$ and column c is in the range $0 \dots m$ because of the constant value in each row.

This pseudocode captures the approach used on the problem instance shown earlier:

OBSERVE: pseudocode for Gauss Jordan Elimination

```
gaussJordan (A)
  foreach base=0 to m-1 do
    baseCoeff = A[base][base]
    foreach row=0 to m-1 do
      if (row != base) then
        innerCoeff = A[row][base]
        foreach column c=base to m do
          A[row][column] -= (innerCoeff/baseCoeff) * A[base][column]
```

It's hard to understand triply-nested loops just by looking at them. The best way to follow this logic is to write the code and follow the logic within the debugger.

 In the **numeric** package, create a **GaussJordan** class as shown:

CODE TO TYPE: GaussJordan class

```
package numeric;

public class GaussJordan {


    public static void gaussJordan(double[][] A) {
        int m = A.length;

        for (int base = 0; base < m; base++) {
            double baseCoeff = A[base][base];
            for (int row = 0; row < m; row++) {
                if (row != base) {
                    double innerCoeff = A[row][base];
                    for (int c = base; c <= m; c++) {
                        A[row][c] -= (innerCoeff/baseCoeff)*A[base][c];
                    }
                }
            }
        }
    }

    public static void main(String[] args) {
        double [][]mat = {{1,3,5,9}, {2,7,2,2}, {1,1,4,2}};

        gaussJordan(mat);

        for (int i = 0; i < mat.length; i++) {
            for (int j = 0; j < mat[0].length; j++) {
                System.out.print(mat[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

 Save and run it; it computes a solution for the earlier problem instance:

OBSERVE: Sample execution of GaussJordan

```
1.0 0.0 0.0 -9.529411764705884
0.0 1.0 0.0 2.352941176470587
0.0 0.0 -17.0 -39.0
```

You can confirm that these values correspond to the computed solution earlier. That is, $x = -162/17$, $y = 40/17$, and $z = 39/17$. Let's take a closer look at the code:

OBSERVE: Computing Gauss Jordan on a matrix

```
public static void gaussJordan(double[][] A) {
    int m = A.length;

    for (int base = 0; base < m; base++) {
        double baseCoeff = A[base][base];
        for (int row = 0; row < m; row++) {
            if (row != base) {
                double innerCoeff = A[row][base];
                for (int c = base; c <= m; c++) {
                    A[row][c] -= (innerCoeff/baseCoeff)*A[base][c];
                }
            }
        }
    }
}
```

base iterates over each row in the matrix, and **baseCoeff** stores the coefficient of the variable being eliminated in each pass. For every other row in the matrix (**other than base**), **the innermost loop reduces the coefficients of these rows proportional to baseCoeff**. **innerCoeff** is used to normalize the adjustment which should eliminate the coefficient of the row **base** in all equations.

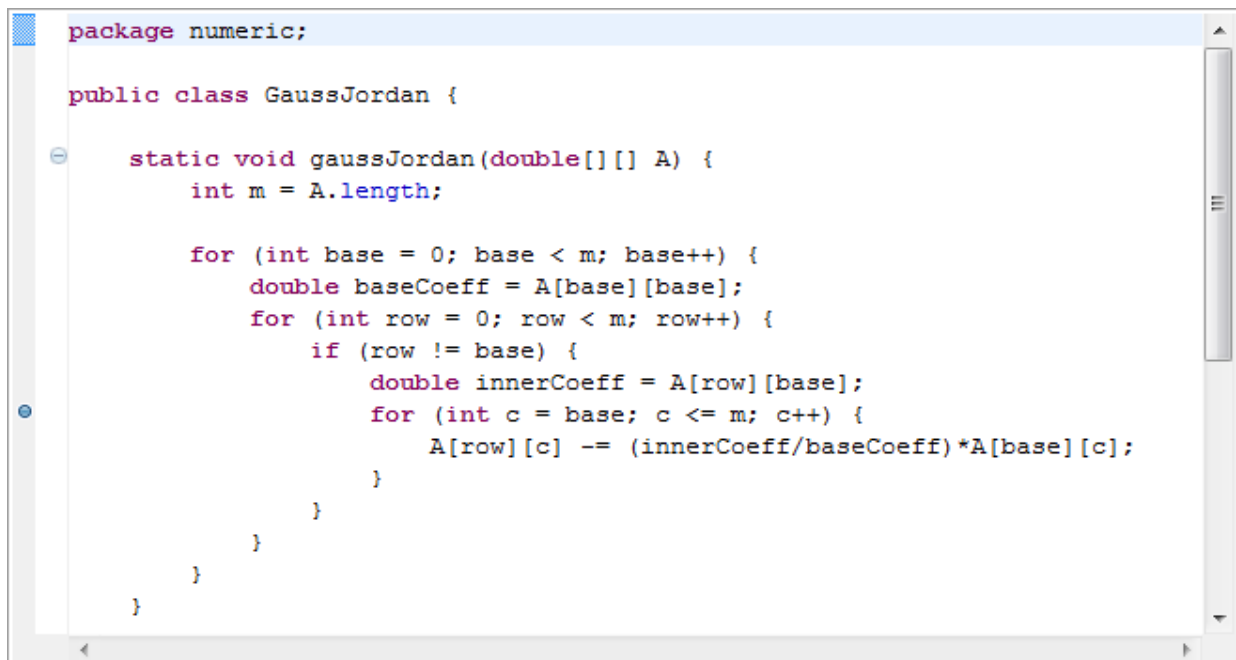
OBSERVE:

```
public static void main(String[] args) {
    double [][]mat = {{1,3,5,9}, {2,7,2,2}, {1,1,4,2}};

    gaussJordan(mat);

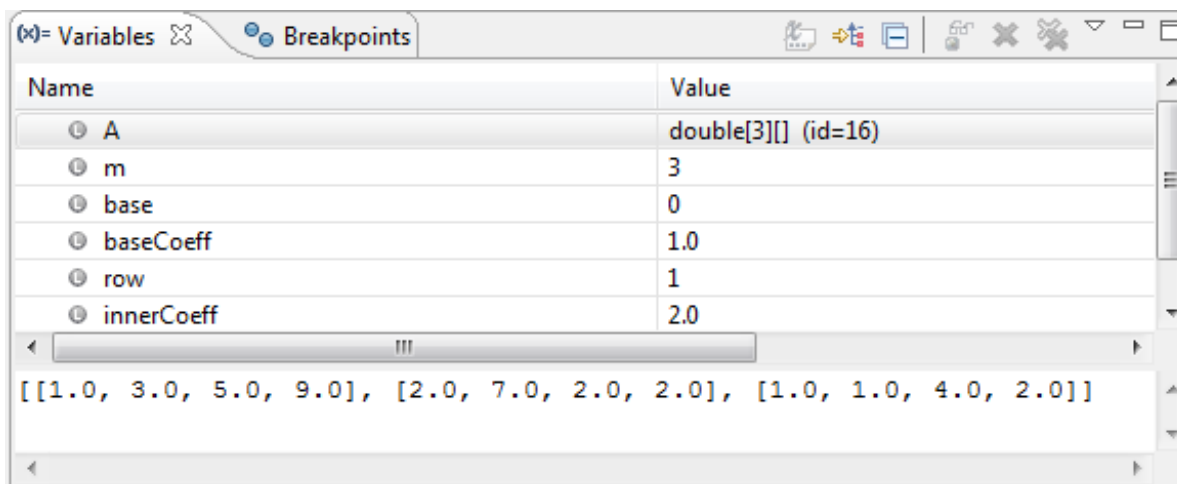
    for (int i = 0; i < mat.length; i++) {
        for (int j = 0; j < mat[0].length; j++) {
            System.out.print(mat[i][j] + " ");
        }
        System.out.println();
    }
}
```

The final code in **GaussJordan** prints out the contents of the matrix **mat**, which was modified in place by the **gaussJordan** method. To better understand this algorithm's behavior, run it within the debugger. This image shows the **GaussJordan** as it appears in Eclipse.



Set a breakpoint at the innermost for loop (do this by double-clicking within the blue vertical border on the line

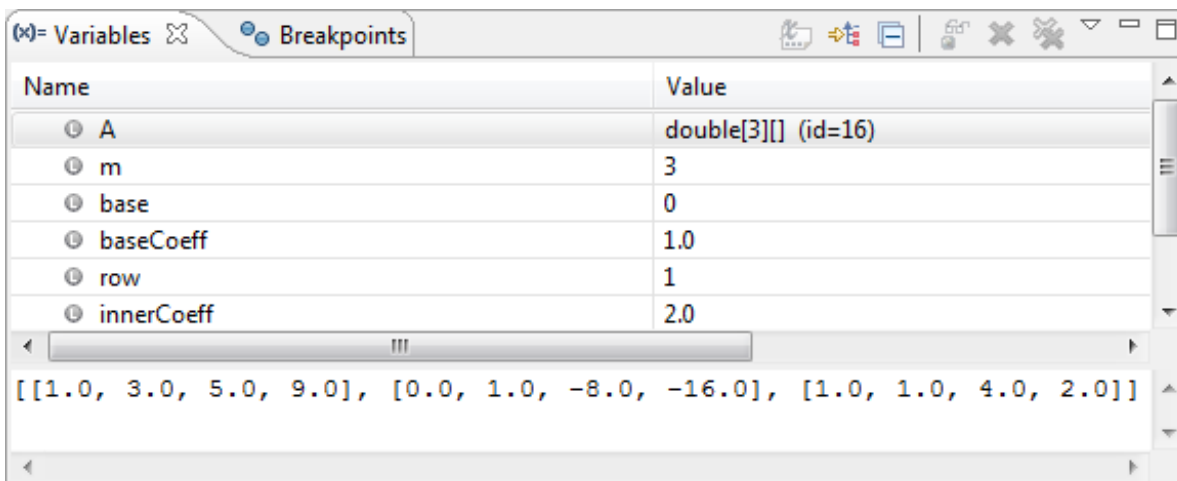
corresponding to this loop) and run **GaussJordan** using the Eclipse debugger (**Debug As | Java Application**). When the debugger stops the first time, select **Window | Show View | Other | Debug | Variables** to show the variables as follows (depending on the current Eclipse version, your screen may be slightly different):



Name	Value
A	double[3][] (id=16)
m	3
base	0
baseCoeff	1.0
row	1
innerCoeff	2.0

[[1.0, 3.0, 5.0, 9.0], [2.0, 7.0, 2.0, 2.0], [1.0, 1.0, 4.0, 2.0]]

The matrix *A* appears in the debugger as three rows of four values each. This matches the equation set described earlier exactly. Now use the debugger tools to continue the execution four times. With each continuation, the values of *A* change and ultimately become this:



Name	Value
A	double[3][] (id=16)
m	3
base	0
baseCoeff	1.0
row	1
innerCoeff	2.0

[[1.0, 3.0, 5.0, 9.0], [0.0, 1.0, -8.0, -16.0], [1.0, 1.0, 4.0, 2.0]]

The *x* variable has been eliminated from the second row in *A* because the first value of the second group of numbers is 0. Do this five more times; the values of the third row in *A* also change to eliminate its coefficient for *x*.

Rounding Errors

Is this implementation complete and correct? Go back to the matrix definition in **GaussJordan** and change it as shown:

CODE TO TYPE:

```
...
public static void main(String[] args) {
    double [][]mat = {{1,3,5,9}, {2,7,2,2}, {1,1,4,210000,10000,40000,20000}};.

    gaussJordan(mat);
...

```

Nothing has changed mathematically because you have only multiplied the coefficients in the last row by 10000, but check the output:

OBSERVE: Output of revised matrix

```
1.0 0.0 -3.552713678800501E-15 -9.529411764705884
0.0 1.0 0.0 2.3529411764705905
0.0 0.0 -170000.0 -390000.0
```

Because of this small change, the code was unable to eliminate the coefficient for *z* in the first equation. You might be tempted to use the **java.math.BigDecimal** class to represent all values because it is designed to store arbitrary-precision signed decimal numbers. However, the comment below that appears in the documentation for the **divide** method of **BigDecimal** states that, "if the exact quotient cannot be represented (because it has a non-terminating decimal expansion) an **ArithmeticException** is thrown." So you can't use **BigDecimal**.

To learn about other issues that pertain to floating-point computations, read the technical document, [What Every Computer Scientist Should Know About Floating-Point Arithmetic](#). This paper is *the standard* for explaining challenges the you'll encounter when working with floating-point. For now, let's focus on some essential points of floating-point numbers.

Computations performed on floating numbers can produce infinitesimal differences in results, such as the value above, which is on the order of 10^{-15} .

The usual strategy we use to deal with these very small numbers is to recognize that they typically occur only through subtraction and addition, rather than multiplication and division. That's because in order to achieve such low exponents, you would need to divide two numbers that are 15 orders of magnitude apart; this may happen in random data, but it is unlikely to occur with most real-world data.

Make these changes to **GaussJordan** with the revised *mat* matrix definition in the **main()** method:

OBSERVE: Modifications to GaussJordan

```
package numeric;

public class GaussJordan {
    static final double epsilon = 1e-9;

    static void gaussJordan(double[][] A) {
        int m = A.length;

        for (int base = 0; base < m; base++) {
            double baseCoeff = A[base][base];
            for (int row = 0; row < m; row++) {
                if (row != base) {
                    double innerCoeff = A[row][base];
                    for (int c = base; c <= m; c++) {
                        A[row][c] -= (innerCoeff/baseCoeff)*A[base][c];
                        if (A[row][c] < epsilon && A[row][c] > -epsilon) {
                            A[row][c] = 0;
                        }
                    }
                }
            }
        }
    }

    public static void main(String[] args) {
        double [][]mat = {{1,3,5,9}, {2,7,2,2}, {10000,10000,40000,20000}};

        gaussJordan(mat);

        for (int i = 0; i < mat.length; i++) {
            for (int j = 0; j < mat[0].length; j++) {
                System.out.print(mat[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```



Save and run it. The problem has been masked:

OBSERVE: Output with larger coefficients, though same solution

```
1.0 0.0 0.0 -9.529411764705884
0.0 1.0 0.0 2.3529411764705905
0.0 0.0 -170000.0 -390000.0
```

The above solution reaches a nearly identical solution (only the middle line is off by a few fractional digits). You can use this approach when your algorithm depends on detecting zero values in your computations.

Partial Input Data

There are other situations that could cause the existing implementation to fail. Many of the equations are not mathematically *independent*, which means that one of the equations is equivalent to a sequence of additions (and subtractions) of the other equations. For example, change the **mat** input matrix in the **main()** method as shown:

CODE TO TYPE:

```
...
public static void main(String[] args) {
    double [][]mat = {{1,3,5,9}, {2,7,2,2}, {10000,10000,40000,200002,6,10,18}};
    .

    gaussJordan(mat);
    ...
```

The third row is simply twice the first row. Rerun the class:

OBSERVE: Output when equations are only partially solvable


```
1.0 0.0 NaN NaN
0.0 1.0 NaN NaN
0.0 0.0 0.0 0.0
```

NaN stands for "Not a Number," which is defined in the Floating-Point standard. This occurs typically when dividing by zero. Java only throws an `ArithmeticException` when an integer division causes a divide by zero; floating-point computations give no indication that anything has gone wrong. To understand why the problem happens, go back to the pseudocode; you can see that there is no protection when **baseCoeff** is zero. Fix it now by treating any **baseCoeff** value sufficiently close to zero as zero; in other words, skip that column:

CODE TO TYPE: Modifications to gaussJordan method

```
static void gaussJordan(double[][] A) {
    int m = A.length;

    for (int base = 0; base < m; base++) {
        double baseCoeff = A[base][base];
        if (baseCoeff < epsilon && baseCoeff > -epsilon) { continue; }
        for (int row = 0; row < m; row++) {
            if (row != base) {
                double innerCoeff = A[row][base];
                for (int c = base; c <= m; c++) {
                    A[row][c] -= (innerCoeff/baseCoeff)*A[base][c];
                    if (A[row][c] < epsilon && A[row][c] > -epsilon) {
                        A[row][c] = 0;
                    }
                }
            }
        }
    }
}
```

 Save and run it:

OBSERVE: Proper output when equations are only partially solvable

```
1.0 0.0 29.0 57.0
0.0 1.0 -8.0 -16.0
0.0 0.0 0.0 0.0
```

While the matrix has not been fully reduced to one variable per row, the output clearly demonstrates that there is no unique solution for all variables, but rather a family of solutions.


Since you have just protected against zero coefficients, consider the set of equations below, in which no equation uses more than two variables; note that the coefficient of x in the first equation is zero.

CODE TO TYPE:

```
...
public static void main(String[] args) {
    double [][]mat = {{1,3,5,9}, {2,7,2,2}, {2,6,10,18}{0,2,1,7}, {1,2,0,3}, {2,
0,5,3}};.

    gaussJordan(mat);
...

```

 Save and run it:

OBSERVE: Invalid output when first coefficient is zero

```
0.0 0.0 0.0 3.4
1.0 2.0 0.0 3.0
2.0 0.0 5.0 3.0
```


However, if you reorder the rows—which ultimately should have no effect on the solution—a solution can be found. Change it again as shown:

CODE TO TYPE:

```
...
public static void main(String[] args) {
    double [][]mat = {{0,2,1,7}, {1,2,0,3}, {2,0,5,3}{1,2,0,3}, {0,2,1,7}, {2,0,
5,3}};.

    gaussJordan(mat);
...

```

 This results in a valid solution:

OBSERVE: Valid output when reorganizing rows

```
1.0 0.0 0.0 -2.428571428571429
0.0 2.0 0.0 5.428571428571429
0.0 0.0 7.0 11.0
```

You must modify the algorithm slightly such that for each of the first m columns, it finds a non-zero *pivot* value to use, rather than simply choosing the assigned coefficient in that column. It turns out that there are mathematical advantages if the pivot is the coefficient in that column of greatest magnitude. The modified pseudocode looks like this:

OBSERVE: pseudocode for Gauss Jordan Elimination

```
gaussJordan (A)
  foreach base=0 to m-1 do
    determine row r whose A[r][base] is highest magnitude; if zero skip and continue
    if r is different from base, swap rows r and base
    baseCoeff = A[base][base]
    foreach row=0 to m-1 do
      if (row != base) then
        innerCoeff = A[row][base]
        foreach column c=base to m do
          A[row][c] -= (innerCoeff/baseCoeff)*A[base][c]
```

The actual code modifications to the **gaussJordan** method look like this:

CODE TO TYPE: Modifications to GaussJordan class

```
static void gaussJordan(double[][] A) {
    int m = A.length;

    for (int base = 0; base < m; base++) {
        double pivot = 0;
        int r = -1;
        for (int k = base; k < m; k++) {
            if (Math.abs(A[k][base]) > pivot) {
                pivot = Math.abs(A[k][base]);
                r = k;
            }
        }
        if (pivot < epsilon && pivot > -epsilon) { continue; }
        if (r != base) {
            for (int c = base; c < m+1; c++) {
                double tmp = A[base][c];
                A[base][c] = A[r][c];
                A[r][c] = tmp;
            }
        }
        double baseCoeff = A[base][base];
        if (baseCoeff < epsilon && baseCoeff > -epsilon) { continue; }
        for (int row = 0; row < m; row++) {
            if (row != base) {
                double innerCoeff = A[row][base];
                for (int c = base; c <= m; c++) {
                    A[row][c] -= (innerCoeff/baseCoeff)*A[base][c];
                    if (A[row][c] < epsilon && A[row][c] > -epsilon) {
                        A[row][c] = 0;
                    }
                }
            }
        }
    }
}
```



Save and run it; the output is correct again:

OBSERVE: Valid output when reorganizing rows

```
2.0 0.0 0.0 -4.857142857142858
0.0 2.0 0.0 5.428571428571429
0.0 0.0 -3.5 -5.5
```

Matrix Determinant

There are a number of useful algorithms over square matrices that demonstrate techniques you can use to

solve worthwhile problems. In linear algebra, the *determinant* is a value associated with a square matrix. When the square matrix represents a system of linear equations, there will be a unique solution for the equations if the determinant is non-zero. You can use the determinant to solve for the linear system of equations, much like the *Gauss Jordan* elimination.

$$A = \begin{pmatrix} 3 & 5 \\ 2 & 4 \end{pmatrix}$$

Given the above two-dimensional matrix A, its determinant is:

$$\det(A) = \begin{vmatrix} 3 & 5 \\ 2 & 4 \end{vmatrix} = 3 * 4 - 2 * 5 = 2$$

You can visualize this computation by subtracting the product of the Northeast diagonal (2*5) from the product of the Southwest diagonal (3*4). To show the utility of the determinant, let's solve this system of two linear equations:

$3x + 5y = 7$	E1
$2x + 4y = 5$	E2

If you look at the 2x2 matrix formed by just the coefficients of the x and y variables, you'll recognize the earlier 2x2 matrix. Because this determinant has a non-zero value, you know that there is a valid unique solution. To determine the solution for x and y, we need to compute four determinant values:

$$x = \frac{\begin{vmatrix} 7 & 5 \\ 5 & 4 \end{vmatrix}}{\begin{vmatrix} 3 & 5 \\ 2 & 4 \end{vmatrix}} \quad y = \frac{\begin{vmatrix} 3 & 7 \\ 2 & 5 \end{vmatrix}}{\begin{vmatrix} 3 & 5 \\ 2 & 4 \end{vmatrix}}$$

The denominators of these two fractions are the determinant when using the coefficients of the x and y variables. The numerator of the solution for x is the determinant of a 2x2 matrix that is formed by replacing the column containing the x coefficients with the column of the constant values. Similarly, the numerator of the solution for y is the determinant of a 2x2 matrix formed by replacing the column containing the y coefficients with the column of the constant values. The value of the x numerator is $7*4 - 5*5 = 3$ while the value of the y numerator is $3*5 - 2*7 = 1$. These results show that the solution to these equations is ($x = 3/2$ and $y = 1/2$). You can verify these values by plugging them back into either of the original equations.

Does this scale to equations with more than 2 variables? Yes! Let's revisit the earlier set of three equations used in the Gauss Jordan section and use determinants to compute the solution. Here are those equations again:

$x + 3y + 5z = 9$	E1
$2x + 7y + 2z = 2$	E2
$x + y + 4z = 2$	E3

Using the same determinant logic as the 2x2 case, the solution to this equation is:


$$x = \frac{\begin{vmatrix} 9 & 3 & 5 \\ 2 & 7 & 2 \\ 1 & 1 & 4 \end{vmatrix}}{\begin{vmatrix} 1 & 3 & 5 \\ 2 & 7 & 2 \\ 1 & 1 & 4 \end{vmatrix}} \quad y = \frac{\begin{vmatrix} 1 & 9 & 5 \\ 2 & 2 & 2 \\ 1 & 2 & 4 \end{vmatrix}}{\begin{vmatrix} 1 & 3 & 5 \\ 2 & 7 & 2 \\ 1 & 1 & 4 \end{vmatrix}} \quad z = \frac{\begin{vmatrix} 1 & 3 & 9 \\ 2 & 7 & 2 \\ 1 & 1 & 2 \end{vmatrix}}{\begin{vmatrix} 1 & 3 & 5 \\ 2 & 7 & 2 \\ 1 & 1 & 4 \end{vmatrix}}$$

In the numerators, you replace the column corresponding to the variable with the column of constant values.

To solve this, you need to be able to compute the determinant for a 3x3 matrix. Fortunately, you can compute the determinant of a 3x3 matrix by a computation involving the determinants of 2x2 vertices. This image shows how this is done:

$$\begin{vmatrix} 1 & 9 & 5 \\ 2 & 2 & 2 \\ 1 & 2 & 4 \end{vmatrix} = 1 * \begin{vmatrix} 2 & 2 \\ 2 & 4 \end{vmatrix} - 9 * \begin{vmatrix} 2 & 2 \\ 1 & 4 \end{vmatrix} + 5 * \begin{vmatrix} 2 & 2 \\ 1 & 2 \end{vmatrix}$$

To visualize this, take the top row of the 3x3 matrix which contains the values (1,9,5). For each of these values, compute the determinants of the three 2x2 matrices that remain when you remove the top row and the respective column of each of these values. Then, sum the computation above, *alternating signs of the constituent sub-parts*. Instead of doing this operation by hand, you need to write a program (especially when computing determinants for higher-order matrices).

 In the **numeric** package, create a **Matrix** class as shown:

CODE TO TYPE: Matrix class

```
package numeric;

public class Matrix {

    public static double det(double[][] m) {
        switch (m.length) {
            case 1:
                return m[0][0];
            case 2:
                return m[0][0]*m[1][1] - m[0][1]*m[1][0];
            case 3:
                return m[0][0]*( m[1][1]*m[2][2] - m[2][1]*m[1][2]) -
                    m[0][1]*( m[1][0]*m[2][2] - m[2][0]*m[1][2]) +
                    m[0][2]*( m[1][0]*m[2][1] - m[2][0]*m[1][1]);
        }

        double result = 0;
        for (int i = 0; i < m[0].length; i++) {
            double temp[][] = new double[m.length - 1][m[0].length - 1];
            for (int j = 1; j < m.length; j++) {
                System.arraycopy(m[j], 0, temp[j-1], 0, i);
                System.arraycopy(m[j], i+1, temp[j-1], i, m[0].length-i-1);
            }

            result += m[0][i] * Math.pow(-1, i) * det(temp);
        }

        return result;
    }
}
```

Let's look at this code more closely. As a recursive implementation, there are three base cases to consider—matrices of size 1x1, 2x2, and 3x3:

OBSERVE: Determinant recursion base cases

```
switch (m.length) {
    case 1:
        return m[0][0];
    case 2:
        return m[0][0]*m[1][1] - m[0][1]*m[1][0];
    case 3:
        return m[0][0]*( m[1][1]*m[2][2] - m[2][1]*m[1][2]) -
            m[0][1]*( m[1][0]*m[2][2] - m[2][0]*m[1][2]) +
            m[0][2]*( m[1][0]*m[2][1] - m[2][0]*m[1][1]);
}
```

The above codes the computation as discussed earlier. For matrices of size $n \geq 4$, the code must recreate n sub-matrices of size $n-1$ and recursively call **det** with these sub-matrices, similar to the 3x3 example described earlier.

OBSERVE: Recursive invocations

```
double result = 0;
for (int i = 0; i < m[0].length; i++) {
    double temp[][] = new double[m.length - 1][m[0].length - 1];
    for (int j = 1; j < m.length; j++) {
        System.arraycopy(m[j], 0, temp[j-1], 0, i);
        System.arraycopy(m[j], i+1, temp[j-1], i, m[0].length-i-1);
    }

    result += m[0][i] * Math.pow(-1, i) * det(temp);
}

return result;
```

This code processes an $n \times n$ matrix by creating n smaller $(n-1) \times (n-1)$ sub-matrices in **temp**. The two **arraycopy** invocations copy the left and right side of the smaller matrices, essentially skipping the i^{th} column with each pass. Using the **Math.pow(-1,i)** statement, the **result** alternates between adding and subtracting the partial computations.

Add this method to the end of the **Matrix** class to solve a linear system of equations represented by the $m \times m+1$ matrix used earlier:

CODE TO TYPE: Modifications to Matrix class

```
public static double[] solve(double[][] mat) {
    double[][] base = new double[mat.length][mat[0].length-1];
    for (int i = 0; i < mat.length; i++) {
        System.arraycopy(mat[i], 0, base[i], 0, mat[0].length-1);
    }
    double denom = det(base);
    if (denom == 0) {
        return null;
    }

    double[] solution = new double[mat.length];
    for (int k = 0; k < mat.length; k++) {
        for (int i=0, j=0; i < mat.length; i++, j++) {
            System.arraycopy(mat[i], 0, base[i], 0, mat[0].length-1);
            base[i][k] = mat[j][mat[0].length-1];
        }
        solution[k] = det(base)/denom;
    }

    return solution;
}
```

When you review implementations of mathematical algorithms, you must become familiar with array indices and nested loops. One of the qualities of efficient mathematical code is dense nested looping logic. Let's take

a closer look at this code:

OBSERVE: Compute denominator determinant

```
double[][] base = new double[mat.length][mat[0].length-1];
for (int i = 0; i < mat.length; i++) {
    System.arraycopy(mat[i], 0, base[i], 0, mat[0].length-1);
}
double denom = det(base);
if (denom == 0) {
    return null;
}
```

The above code **constructs a base array of size $m \times m$** , where m is the number of rows in the input **mat** matrix. This matrix contains the variable coefficients only. **If the determinant of this matrix is zero**, there is no unique solution:

OBSERVE: Computing partial sums

```
double[] solution = new double[mat.length];
for (int k = 0; k < mat.length; k++) {
    for (int i=0, j=0; i < mat.length; i++, j++) {
        System.arraycopy(mat[i], 0, base[i], 0, mat[0].length-1);
        base[i][k] = mat[j][mat[0].length-1];
    }
    solution[k] = det(base)/denom;
}

return solution;
```

The resulting solution of m variables is determined by computing the fractions of determinants identified earlier. The **inner for loop over the i variable** creates an $m \times m$ matrix where **the k^{th} column in base is replaced by the coefficients of the constant column**, the rightmost column in the original matrix, **mat**.

To validate that this code works, add the **main** method to the end of the **Matrix** class as shown:

CODE TO TYPE: Main method to demonstrate working solve

```
public static void main(String[] args) {
    double[][] mat = {{1,3,5,9}, {2,7,2,2}, {1,1,4,2}};
    double[] vals = solve(mat);
    System.out.println("x=" + vals[0] + ", y=" + vals[1] + ", z=" + vals[2]);
}
```



Save and run it:

OBSERVE: Solution to linear equations using determinants

```
x=-9.529411764705882, y=2.3529411764705883, z=2.2941176470588234
```

Compare these values with earlier values computed in the [Gauss Jordan section](#); they are nearly identical. They only begin to differ in the last few digits. You'll see this phenomenon whenever you work with floating point numbers.

Lessons Learned

Working with floating-point numbers can be surprising and challenging. You need to understand the ways that rounding errors can be introduced into computations.

- **Use epsilon-based conditionals when comparing to zero.** When trying to compare a floating-point number with zero, you must be careful to take into account the minute rounding error that often happens in computations. Instead of comparing with equality, use two conditionals that check if **($x < \text{epsilon} \ \&\& \ x > -\text{epsilon}$)** or call **Math.abs()**.

- **Rounding Errors Can Accumulate:** Even though rounding errors by themselves are minute values, they can rapidly accumulate through computations, decreasing the accuracy of your computations if you don't take time to review the computations in your code. Since each computation involves some rounding errors, try to minimize the number of operations you perform. For example, even though $a*(b-c)$ is equal mathematically to $a*b-a*c$, the latter computation requires three floating point operations while the former computation only requires two (and this would be preferred in your code).
- **Floating point can store impossible numbers:** In floating point, a computation could actually divide by zero without throwing an Exception. When both the numerator and denominator are of type `int`, Java throws an `java.lang.ArithmeticException`.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Brute Force Algorithms

Lesson Objectives

After completing this lesson, you will be able to implement brute-force solutions to permutation-style problems.

Using Brute Force To Solve Permutation Problems

These problems have something in common:

- What are the 5-letter words that you can make using just the letters found in PALINDROME (without repetition)?
- Generate all 4x4 magic squares using the numbers 1 through 16 where all rows, columns, and two long diagonals sum to the same value.
- How many ways can you place N queens on an NxN chessboard such that no two queens attack each other?

Each of these problems relies on some combinatoric permutation of input elements. There is a standard *brute force* approach that can be used for those types of problems, as long as the size of the problem instance isn't too big. In mathematics, a *Permutation* is defined for a set of elements by imposing some particular order of the elements. For example, given the set {"A", "B", "C"}, there are six permutations: {"ABC", "ACB", "BAC", "BCA", "CAB", "CBA"}. To solve each of the above problems, you must somehow compute the valid permutations. In this lesson, you'll see an implementation that you can use as the structure for solving such permutation problems.

Consider how to generate a 3x3 magic square using the digits 1 through 9 where all rows and columns and the two long diagonals sum to the same value. Start by writing in the numbers from 1 to 9, three to a row from top to bottom. Is this a magic square? Nope. Here's an image of six consecutive attempts.

1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	...
4	5	6	4	5	6	4	5	6	4	5	6	4	5	6	4	5	6	
7	8	9	7	9	8	8	7	9	8	9	7	9	7	8	9	8	7	

These attempts were generated by backtracking from the initial attempt. Erase the 9 and 8 digit in the first attempt, and instead try placing a 9 in the middle of the bottom row. This second attempt is not a magic square either. Okay, so backtrack and erase the 8, 9 and 7 digits and instead place an 8 in the left corner of the matrix. At this point, you can generate the third attempt with "8, 7, 9" as the bottom row, which is still not a solution. Now erase the 9 and 7 digits and write "8, 9, 7" as the bottom row of the fourth attempt—still not a solution, so backtrack by erasing the 7, 9, and 8 digits and write a 9 in the left corner of the matrix. You can generate the fifth attempt "9, 7, 8" as the bottom row, which is not a solution. Now erase the 8 and 7 digits and write "9, 8, 7" as the sixth and final attempt—no solution. This certainly appears to be a tedious exercise and you've only tried six of the possible solutions! Fortunately, a computer program can use this approach to try all such configurations.

Given how important backtracking is to that approach, expect to see recursion play an important role in the algorithm. If you can divide your problem into a finite number of steps, the pseudocode below describes a method **solve(int step)** which attempts to solve the "next step" in a progression of steps:

OBSERVE: pseudocode for brute force solver

```
boolean solve(int step) {  
    if (step is past the final step) {  
        return isValid();  
    }  
  
    foreach possible value in step do  
        update state with value  
        if (solve(step+1)) then  
            return true  
  
        undo update of state  
  
    return false;  
}
```

This recursive function first checks to see **if the given step number is one more than the final step**. If so, it's finished, and **returns whether the computed attempt is a valid solution**. If there are more steps left to be executed, then **for each possible value allowed in a given step, solve() updates the state** and recursively tries to **solve the next step**. If **solve(step+1)** ever returns true, then it has worked, and the **algorithm stops immediately**. If, however, this recursive execution returns false, the **most recent state update must be undone** so the next possible value at the given step can be applied. If the **foreach** loop completes without finding a solution, then the entire step fails, so the **last statement in the pseudocode returns false**.

Each problem using this approach has its own **isValid()** method to determine whether the computed attempt solves the stated problem. For the magic square problem, the final code only needs to check that the sums of each row, column, and long diagonal equal the same target value.

You may recall the definition of the Factorial function **n!** in mathematics, which computes **$n * (n-1) * (n-2) * \dots * 2 * 1$** . This function grows incredibly fast! While 9! is a manageable **362,880**, 16! is **20,922,789,888,000**. If you review the six earlier attempts, you can see that this brute-force approach is inefficient because it blindly tries all permutations. It does have the benefit, however, of being relatively easy to write and it takes advantage of the incredible power of computers to try hundreds of thousands of possibilities per second.



Create a new Java Project named **BruteForce** and assign it to the **Java6_Lessons** working set.



In your **BruteForce** project **/src** source folder, create a package **permute**.



In the **permute** package, create a **MagicSquare** class as shown:

CODE TO TYPE: MagicSquare

```
package permute;

public class MagicSquare {
    final int square[][];
    final boolean used[];
    final int n;

    public MagicSquare(int n) {
        square = new int[n][n];
        this.n = n;
        used = new boolean[n*n+1];
    }

    boolean solve(int step) {
        if (step == n*n) {
            return isValid();
        }

        for (int val = 1; val <= n*n; val++) {
            if (used[val]) { continue; }

            used[val] = true;
            square[step/n][step%n] = val;
            if (solve(step+1)) {
                return true;
            }
            square[step/n][step%n] = 0;
            used[val] = false;
        }

        return false;
    }

    boolean validUpTo(int step) {
        for (int r = 0; r < n; r++) {
            if (step == (r+1)*n-1) {
                int sum = 0;
                for (int c = 0; c < n; c++) { sum += square[r][c]; }
                return (sum == magicSum);
            }
        }
        for (int c = 0; c < n; c++) {
            if (step == n*(n-1)+c) {
                int sum = 0;
                for (int r = 0; r < n; r++) { sum += square[r][c]; }
                return (sum == magicSum);
            }
        }
        return true;
    }
}
```

Let's look closer:

OBSERVE:

```
public class MagicSquare {
    final int square[][];
    final boolean used[];
    final int n;

    public MagicSquare(int n) {
        square = new int[n][n];
        this.n = n;
        used = new boolean[n*n+1];
    }
}
```

The state of the algorithm is contained in two arrays: **used[n]** records whether the number **n** already appears somewhere in the magic square. **square[r][c]** stores the number at the given row and column index location. The **MagicSquare** constructor determines the desired problem size, **n**. The size of **used[]** is one larger than necessary because the numbers in the magic square are from 1 to **n*n**.

OBSERVE:

```
boolean solve(int step) {
    if (step == n*n) {
        return isValid();
    }

    for (int val = 1; val <= n*n; val++) {
        if (used[val]) { continue; }

        used[val] = true;
        square[step/n][step%n] = val;
        if (solve(step+1)) {
            return true;
        }
        square[step/n][step%n] = 0;
        used[val] = false;
    }

    return false;
}
```

The real logic is in the **solve(int step)** method. The first invocation of this method must be **solve(0)**. Once all numbers have been placed (when **step is n*n** or one more than the number of steps when counting from zero), the method determines whether the state of the magic square is a valid solution using the **isValid()** method (which we'll write shortly).

To determine "foreach possible value in step," this code **relies on the used[n] array so it doesn't place the same number multiple times in the magic square**. To "update state with value," the code records **used[val]=true** and **places the value in the magic square at the appropriate row and column**. This code uses a common idiom to convert a simple number into a two-dimensional row and column placement. The integer computation **step/n** properly truncates the step number to compute the row value, while **step%n** uses modulo arithmetic to determine the proper column. Once the state is updated, it recursively calls **solve(step+1)**; if this method returns **true**, a solution has been found. Otherwise, it must **undo the state change (both in used and square)** before continuing the **for** loop. If this loop completes without having found a solution, the method **returns false** and backtracks to the previous step.

To complete the implementation of **MagicSquare**, make the changes below. They'll take advantage of the mathematical fact that the target sum value for an **n x n** magic square is **n*(n*n+1)/2**. So, for a **3x3** magic square, the sum of each row, column and diagonal is **3*(3*3+1)/2 = 3*10/2 = 15**.

CODE TO TYPE: Modify MagicSquare

```
package permute;

public class MagicSquare {
    final int square[][];
    final boolean used[];
    final int n;
    final int magicSum;

    public MagicSquare(int n) {
        square = new int[n][n];
        this.n = n;
        used = new boolean[n*n+1];
        magicSum = n*(n*n+1)/2;
    }

    boolean isValid() {
        int sumD1 = 0;
        int sumD2 = 0;
        for (int i = 0; i < n; i++) {
            int sumR = 0;
            int sumC = 0;
            sumD1 += square[i][i];
            sumD2 += square[i][n-i-1];
            for (int j = 0; j < n; j++) {
                sumR += square[i][j];
                sumC += square[j][i];
            }
            if (sumR != magicSum || sumC != magicSum) { return false; }
        }

        // diagonals
        return (sumD1 == magicSum && sumD2 == magicSum);
    }

    boolean solve(int step) {
        if (step == n*n) {
            return isValid();
        }

        for (int val = 1; val <= n*n; val++) {
            if (used[val]) { continue; }

            used[val] = true;
            square[step/n][step%n] = val;
            if (solve(step+1)) {
                return true;
            }
            square[step/n][step%n] = 0;
            used[val] = false;
        }

        return false;
    }

    boolean validUpTo(int step) {
        for (int r = 0; r < n; r++) {
            if (step == (r+1)*n-1) {
                int sum = 0;
                for (int c = 0; c < n; c++) { sum += square[r][c]; }
                return (sum == magicSum);
            }
        }
        for (int c = 0; c < n; c++) {
            if (step == n*(n-1)+c) {
                int sum = 0;
                for (int r = 0; r < n; r++) { sum += square[r][c]; }
            }
        }
    }
}
```

```


        return (sum == magicSum);
    }
}
return true;
}


public void outputSolution () {
    for (int r = 0; r < n; r++) {
        for (int c = 0; c < n; c++) {
            System.out.print(square[r][c]);
            System.out.print(' ');
        }
        System.out.println();
    }
    System.out.println();
}
}


```

The **outputSolution()** method prints out the two-dimensional square using the values in **square**. The **isValid()** method does the real work, using a nested **for** loop to compute the sum of each row, column, and long diagonal. If any sum fails to match **magicSum**, the **isValid()** method returns **false**, which forces the **solve()** method to backtrack and try to find another solution.

To validate this solution, write this performance code:

 In your **BruteForce** project, create a **/performance** source folder.

 In the **/performance** source folder, create a **permute** package.

 In the **permute** package, create a **Main** class as shown:

CODE TO TYPE: Main class

```

package permute;

public class Main {
    public static void main(String[] args) {
        MagicSquare m = new MagicSquare(3);
        m.solve(0);
        m.outputSolution();
    }
}

```

 Save and run **Main**.

OBSERVE: Output from Main for 3x3 magic square

```

2 7 6
9 5 1
4 3 8

```

This is a valid 3x3 magic square because all rows, columns, and long diagonals sum to 15.

Before going further, stop and think about how many 3x3 magic square solutions might exist. This is a natural extension to the problem. You can determine how many solutions there are by adding a **count()** method that makes a small modification to the basic algorithm. Modify **MagicSquare** as shown:

CODE TO TYPE: Updates to MagicSquare class

```
int total = 0;

...

void count(int step) {
    if (step == n*n) {
        if (isValid()) {
            total++;
            outputSolution();
        }
        return;
    }

    for (int val = 1; val <= n*n; val++) {
        if (used[val]) { continue; }

        used[val] = true;
        square[step/n][step%n] = val;
        if (validUpTo(step)) {
            count(step+1);
        }
        square[step/n][step%n] = 0;
        used[val] = false;
    }
}
```

Let's look closer:

OBSERVE:

```
int total = 0;

void count(int step) {
    if (step == n*n) {
        if (isValid()) {
            total++;
            outputSolution();
        }
        return;
    }

    for (int val = 1; val <= n*n; val++) {
        if (used[val]) { continue; }

        used[val] = true;
        square[step/n][step%n] = val;
        count(step+1);
        square[step/n][step%n] = 0;
        used[val] = false;
    }
}
```

The count(step) method stores the total number of such magic squares found in an attribute, **total**. The structure of this method is nearly identical to **solve(step)**, except that it doesn't stop looking for solutions when the first one is found. Accordingly, this method is now defined as **void count(int step)**. So, when the final step is reached and **isValid()** validates a solution, the **count()** method **increments the total count** and **outputs the solution to the screen**. The recursive call always executes and the code backtracks after every recursive invocation. Together, these changes ensure that all solutions are inspected.


Modify **Main** as shown:

CODE TO TYPE: Main class

```
package permute;

public class Main {
    public static void main(String[] args) {
        MagicSquare m = new MagicSquare(3);
        m.solve(0);
        m.outputSolution();

        m = new MagicSquare(3);
        m.count(0);
        System.out.println("There are " + m.total + " possible squares.");
    }
}
```

 Save and run it.

INTERACTIVE SESSION: Output from revised Main

```
2 7 6
9 5 1
4 3 8
```

```
2 7 6
9 5 1
4 3 8
```

```
2 9 4
7 5 3
6 1 8
```

```
4 3 8
9 5 1
2 7 6
```

```
4 9 2
3 5 7
8 1 6
```

```
6 1 8
7 5 3
2 9 4
```

```
6 7 2
1 5 9
8 3 4
```

```
8 1 6
3 5 7
4 9 2
```

```
8 3 4
1 5 9
6 7 2
```

```
There are 8 possible squares.
```

These magic squares are all really the same solution rotated (and flipped) to produce eight different versions of the same magic square.

It's amazing how quickly this program computed these solutions. However, if you change this to solve a 4x4 magic square, you'll have to wait a while longer for solutions. How long? Well, a 4x4 magic square requires $16!$ permutations, or **20,922,789,888,000**; as a rough estimate, it will take 5,765,760 times as long to generate all 4x4 solutions as it did for the 3x3 solution. Clearly, you have to optimize this approach, or you'll never be able to compute even slightly larger

problems.

Fortunately, you can modify the **solve()** method to search through the solution set more intelligently. Basically, instead of blindly pursuing each recursive step, you can validate partial results of the search before going forward. Revise **MagicSquare** as shown:

CODE TO TYPE: Revised MagicSquare

```

package permute;

public class MagicSquare {
    final int square[][];
    final boolean used[];
    final int n;
    final int magicSum;
    int total = 0;

    public MagicSquare(int n) {
        square = new int[n][n];
        this.n = n;
        used = new boolean[n*n+1];
        magicSum = n*(n*n+1)/2;
    }

    // handles only rows and columns
    boolean validUpTo(int step) {
        for (int r = 0; r < n; r++) {
            if (step == (r+1)*n-1) {
                int sum = 0;
                for (int c = 0; c < n; c++) { sum += square[r][c]; }
                return (sum == magicSum);
            }
        }

        for (int c = 0; c < n; c++) {
            if (step == n*(n-1)+c) {
                int sum = 0;
                for (int r = 0; r < n; r++) { sum += square[r][c]; }
                return (sum == magicSum);
            }
        }

        return true;
    }

    boolean isValid() {
        int sumD1 = 0;
        int sumD2 = 0;
        for (int i = 0; i < n; i++) {
int sumR = 0;
int sumC = 0;
            sumD1 += square[i][i];
            sumD2 += square[i][n-i-1];
for (int j = 0; j < n; j++) {
sumR += square[i][j];
sumC += square[j][i];
}
if (sumR != magicSum || sumC != magicSum) { return false; }
        }

        // diagonals
        return (sumD1 == magicSum && sumD2 == magicSum );
    }

    boolean solve(int step) {
        if (step == n*n) {
            return isValid();
        }

        for (int val = 1; val <= n*n; val++) {
            if (used[val]) { continue; }

            used[val] = true;
            square[step/n][step%n] = val;

```

```

        if (validUpTo(step) && solve(step+1)) {
            return true;
        }
        square[step/n][step%n] = 0;
        used[val] = false;
    }

    return false;
}

boolean validUpTo(int step) {
    for (int r = 0; r < n; r++) {
        if (step == (r+1)*n-1) {
            int sum = 0;
            for (int c = 0; c < n; c++) { sum += square[r][c]; }
            return (sum == magicSum);
        }
    }
    for (int c = 0; c < n; c++) {
        if (step == n*(n-1)+c) {
            int sum = 0;
            for (int r = 0; r < n; r++) { sum += square[r][c]; }
            return (sum == magicSum);
        }
    }
    return true;
}

public void outputSolution () {
    for (int r = 0; r < n; r++) {
        for (int c = 0; c < n; c++) {
            System.out.print(square[r][c]);
            System.out.print(' ');
        }
        System.out.println();
    }
    System.out.println();
}

void count(int step) {
    if (step == n*n) {
        if (isValid()) {
            total++;
            outputSolution();
        }
        return;
    }
}

for (int val = 1; val <= n*n; val++) {
    if (used[val]) { continue; }

    used[val] = true;
    square[step/n][step%n] = val;
    if (validUpTo(step)) {
        count (step+1);
    }
    square[step/n][step%n] = 0;
    used[val] = false;
}
}
}

```

Let's take a closer look at the **validUpTo()** method:

OBSERVE: validUpTo method

```
// handles only rows and columns
boolean validUpTo(int step) {
    for (int r = 0; r < n; r++) {
        if (step == (r+1)*n-1) {
            int sum = 0;
            for (int c = 0; c < n; c++) { sum += square[r][c]; }
            return (sum == magicSum);
        }
    }

    for (int c = 0; c < n; c++) {
        if (step == n*(n-1)+c) {
            int sum = 0;
            for (int r = 0; r < n; r++) { sum += square[r][c]; }
            return (sum == magicSum);
        }
    }

    return true;
}
```

The algorithm partially reviews its progress after filling each row and column completely. Given a **step** numbered from **0** to **n*n-1**, this means that **whenever step equals (r+1)*n-1** for some row numbered **0 .. n-1**, there is enough information to determine if the sum total of the row is **magicSum**. Similarly, **whenever step equals n*(n-1)+c** for some column numbered **0 .. n-1**, there is enough information to determine if the sum total of the column is **magicSum**.

With these changes in place, modify **Main** to count the number of 4x4 magic squares, as shown:

CODE TO TYPE: Revised Main class

```
package permute;

public class Main {
    public static void main(String[] args) {
        MagicSquare m = new MagicSquare(4);
        m.solve(0);
        m.outputSolution();

        m = new MagicSquare(4);
        m.count(0);
        System.out.println("There are " + m.total + " possible squares.");
    }
}
```



Save and run it. The first 4x4 magic square appears almost immediately:

OBSERVE: First computed magic square

```
1 2 15 16
12 14 3 5
13 7 10 4
8 11 6 9
```

If you let the program run for about three more minutes, it reports that **7,040** 4x4 magic squares were found. You know that each magic square appears 8 times in this set (rotated and flipped); this means there are 880 unique 4x4 magic squares. The revised code prints each of these solutions.

Of course, you can't use this approach for 5x5 magic squares (which have **1.5×10^{25}** possible solutions). You could try to run this example on your own computer. After 41 hours of computation on the 5x5 solution, this solution shows up:

INTERACTIVE SESSION: Solution for 5x5 found


```
Sun Sep 29 13:33:58 EDT 2013
1 2 13 24 25
3 22 19 6 15
23 16 10 11 5
21 7 9 20 8
17 18 14 4 12
```

```
Tue Oct 01 06:21:05 EDT 2013
```

Even so, this powerful technique can be used to solve many small, "human-scale" problems in which you might be interested.

Finding All Five-Letter words in PALINDROME

Now, use this same algorithm to determine the five-letter words that you can make using just the letters found in the word "PALINDROME." Perhaps you can already see how to apply the described algorithm to solve this problem. Using the same pseudocode from before, create this **WordFinder** class.

 In the **permute** package of the **/src** source folder, create a **WordFinder** class as shown:

CODE TO TYPE: WordFinder

```
package permute;

import java.util.*;

public class WordFinder {

    final char[] letters;
    final boolean[] used;
    final int n;

    char[] solution;
    Set<String> results;

    public WordFinder (String word) {
        letters = word.toCharArray();
        Arrays.sort(letters);
        n = letters.length;
        used = new boolean[n];
    }

    public void generate (int numChars) {
        solution = new char[numChars];
        results = new TreeSet<String>();
        generate(numChars, 0);
    }

    void generate(int numChars, int step) {
        if (step == numChars) {
            results.add(new String(solution));
            return;
        }

        for (int idx = 0; idx < n; idx++) {
            if (used[idx]) { continue; }

            used[idx] = true;
            solution[step] = letters[idx];
            generate(numChars, step+1);
            solution[step] = 0;
            used[idx] = false;
        }
    }
}
```

Let's break this solution up into its constituent parts:

OBSERVE: Instantiating the WordFinder problem

```
public class WordFinder {  
  
    final char[] letters;  
    final boolean[] used;  
    final int n;  
  
    char[] solution;  
    Set<String> results;  
  
    public WordFinder (String word) {  
        letters = word.toCharArray();  
        Arrays.sort(letters);  
        n = letters.length;  
        used = new boolean[n];  
    }  
  
    public void generate (int numChars) {  
        solution = new char[numChars];  
        results = new TreeSet<String>();  
        generate(numChars, 0);  
    }  
  
    ...  
}
```


The **letters** array contains the letters from the original word in sorted order. If the original word repeats a letter, that letter will appear multiple times in this array. The **used** array keeps track of which letters have been already used, and the **results** object stores the set of all computed words found.

The **generate(int numChars)** method allows you to use this object repeatedly to generate all words that use a given number of characters. It **sets the size of the solution array based on the desired number of characters**, and it instantiates **results** using a **TreeSet** object; this is done so it can produce the words in sorted order rapidly when requested. This method uses the recursive **generate(numChars,step)** method shown:

OBSERVE: Generating all words using a given number of characters

```
void generate(int numChars, int step) {  
    if (step == numChars) {  
        results.add(new String(solution));  
        return;  
    }  
  
    for (int idx = 0; idx < n; idx++) {  
        if (used[idx]) { continue; }  
  
        used[idx] = true;  
        solution[step] = letters[idx];  
        generate(numChars, step+1);  
        solution[step] = 0;  
        used[idx] = false;  
    }  
}
```

This recursive function terminates when the **step number is the same as the number of desired characters**. **numChars** is passed through as an unchanged parameter to each of the recursive invocations. The **solution[]** array stores the permutation of letters. Whenever an appropriate word is found, **it is added to the results set**. Let's try to run this code.

 Create a **MainWordFinder** class in the **/performance** source folder:

CODE TO TYPE: MainWordFinder demonstration class

```
package permute;

public class MainWordFinder {

    public static void main(String[] args) {
        WordFinder wf = new WordFinder("PALINDROME");
        wf.generate(5);

        System.out.println("There are " + wf.results.size() + " five letter words possible.
");
        System.out.println("First ten are:");
        int idx = 10;
        for (String word : wf.results) {
            System.out.println(word);
            if (--idx == 0) { break; }
        }
    }
}
```



Save and run it.

INTERACTIVE SESSION: Output from MainWordFinder

```
There are 30240 five letter words possible.
The first ten words are:
ADEIL
ADEIM
ADEIN
ADEIO
ADEIP
ADEIR
ADELI
ADELM
ADELN
ADELO
```

N Queens Problem

As a final example, consider the *N Queens problem*, which asks you to place N queens on an NxN chessboard such that no two queens attack each other. For $N \geq 4$ the problem always has a solution, but it may take you some time to determine that solution. If you could only find some way to convert this problem into a permutation problem, then you could use the existing algorithm to solve it. Start by breaking the problem into N steps, placing a non-attacking queen, one at a time, into each of the N columns on the chessboard. With this approach, you need some permutation of queen placements in each column. Let's get started.



In the `/src` source folder `permute` package, create an `NQueensProblem` class as shown:

CODE TO TYPE: NQueensProblem

```
package permute;

public class NQueensProblem {
    final int n;
    final int solution[];

    public NQueensProblem(int n) {
        this.n = n;
        solution = new int[n];
    }

    public void solve() {
        solve(0);
        outputSolution();
    }

    public int count() {
        total = 0;
        count(0);
        return total;
    }

    int total = 0;
    void count(int column) {
        // TBA
    }

    boolean solve(int column) {
        if (column == n) {
            return true;
        }

        // TBA

        return false;
    }

    public void outputSolution () {
        for (int r = 0; r < n; r++) {
            for (int c = 0; c < n; c++) {
                if (solution[c] == r) {
                    System.out.print("Q");
                } else {
                    if ((r-c) %2 == 0) {
                        System.out.print(" ");
                    } else {
                        System.out.print(".");
                    }
                }
            }
            System.out.println();
        }
        System.out.println();
    }
}
```

Using the model of placing a queen in each column, the **solution[i]** array will record the row value (**0 .. n-1**) of each queen placed in the **ith** column. How many ways can you place N queens on an NxN chess board? Well, there are N*N squares and from these you choose N squares. When choosing B elements from a larger set of unique A elements, the mathematical formula to use is **A!/(B!*(A-B)!)**. This number is actually far smaller than the totals we were dealing with earlier. For example, given an 8x8 chess board on which to place 8 queens, the above formula is **64!/(8!*56!)** or **64*63*62*61*60*59*58*57/8*7*6*5*4*3*2*1**, which equals **4,426,165,368**.

We can make this code even more efficient by placing only non-attacking queens at each step instead of placing N queens in the N columns, and *only then* checking whether any two queens attack. Doing this efficiently can be a bit

tricky. This code creates three arrays to keep track of important state information:

CODE TO TYPE:

```
...
public class NQueensProblem {
    final int n;
    final boolean usedRow[];
    final boolean usedDiagonalNE[];
    final boolean usedDiagonalNW[];
    final int solution[];

    public NQueensProblem(int n) {
        this.n = n;
        solution = new int[n];
        usedRow = new boolean[n];
        usedDiagonalNE = new boolean[2*n-1];
        usedDiagonalNW = new boolean[2*n-1];
    }
}
...
```

- **usedRow[i]** records if a queen is placed in row **i**.
- **usedDiagonalNE[i]** records if a queen is placed in one of the northeast diagonals on the board.
- **usedDiagonalNW[i]** records if a queen is placed in one of the northwest diagonals on the board.

There are **n** elements in **usedRow**, but **2*n-1** elements in each of the diagonal arrays. When placing a queen at square (**column**, **row**), the code records that **usedRow[row]** is true. Given the coordinates for **column** and **row**, observe that for all squares on the i^{th} northeast diagonal for i in the range **0 .. 2*n-1**, the sum of **row+column** equals **i**. For squares on the northwest diagonals, observe that the difference of index values **row-column** is in the range **(-n+1) .. (n-1)**, so to normalize this array, the code uses **row-column+n-1** as the index values for i in the range **0 .. 2*n-1** into **usedDiagonalNW[i]**.

Revise the **solve(int column)** method as shown below. You'll see that the recursive call is made only when it is clear that placing the queen at (**column**, **row**) does not attack any existing queen on the board. Once placed, these arrays are updated prior to the recursive invocation; they are reset after the invocation:

CODE TO TYPE: Revised solve() method

```
boolean solve(int column) {
    if (column == n) {
        return true;
    }

    ///TBA
    for (int row = 0; row < n; row++) {
        if (usedRow[row]) { continue; }
        if (usedDiagonalNW[row-column+n-1]) { continue; }
        if (usedDiagonalNE[row+column]) { continue; }

        usedRow[row] = true;
        usedDiagonalNW[row-column+n-1] = true;
        usedDiagonalNE[row+column] = true;
        solution[column] = row;

        if (solve(column+1)) {
            return true;
        }

        usedDiagonalNE[row+column] = false;
        usedDiagonalNW[row-column+n-1] = false;
        usedRow[row] = false;
    }

    return false;
}
```

The recursive method terminates when all columns have a queen. If there are more columns to process, the **for** loop runs through all possible row indices to find one that doesn't currently contain a queen (as determined by the **usedRow** array). However, two queens can attack diagonally, so two different arrays are used to record whether there is already a queen on any of the northeast diagonals or northwest diagonals.

This code completes the **count(int column)** implementation:

CODE TO TYPE: Modified count() method

```
void count(int column) {
    ///TBA
    if (column == n) {
        total++;
        return;
    }

    for (int row = 0; row < n; row++) {
        if (usedRow[row]) { continue; }
        if (usedDiagonalNW[row-column+n-1]) { continue; }
        if (usedDiagonalNE[row+column]) { continue; }

        usedRow[row] = true;
        usedDiagonalNW[row-column+n-1] = true;
        usedDiagonalNE[row+column] = true;
        solution[column] = row;

        count(column+1);

        usedDiagonalNE[row+column] = false;
        usedDiagonalNW[row-column+n-1] = false;
        usedRow[row] = false;
    }
}
```

Write some validating code to demonstrate the proper execution of this method.


 Create a **MainNQueens** class in the **permute** package of the **/performance** source folder:

CODE TO TYPE: MainNQueens class

```
package permute;

public class MainNQueens {
    public static void main(String[] args) {
        for (int i = 8; i < 10; i++) {
            NQueensProblem nqp = new NQueensProblem(i);
            nqp.solve();
            System.out.println("-----");
        }

        for (int i = 4; i < 14; i++) {
            NQueensProblem nqp = new NQueensProblem(i);
            System.out.println(i + ". " + nqp.count());
        }
    }
}
```

 Save and run it.

OBSERVE: Output from MainNQueens

```
Q. . . .
. . . Q
. . Q. .
. . . .Q
Q . . .
. .Q. .
. . Q .
. Q . .

-----

Q. . . .
. . Q . .
Q . . .
. . .Q. .
. . . .Q
. Q . . .
. . . Q
. .Q. . .
. . .Q.

-----

4. 2
5. 10
6. 4
7. 40
8. 92
9. 352
10. 724
11. 2680
12. 14200
13. 73712
```

The table at the end of the output records the total number of unique boards for different values of n . You can validate that these are correct by comparing against well-known tables of these values, like those found at the [On-Line Encyclopedia of Integer Sequences](#).

Lessons Learned

There are many permutation-style problems that can be solved in small problem instances using a brute-force approach. While the technique doesn't scale to larger problem instances, it can be a useful "cure-all" when no known algorithm exists, or you just want to conduct a quick search to see if some solution exists.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Path Finding for Single-Player Games

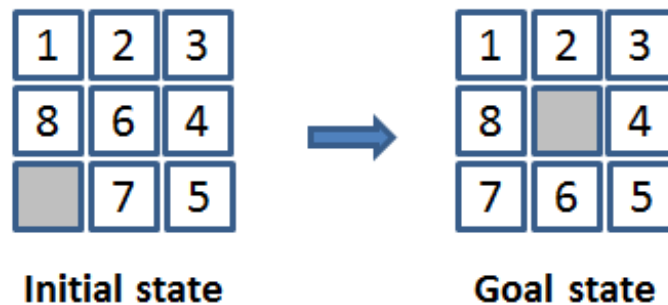
Lesson Objectives

After completing this lesson, you will be able to:

- draw a search tree (of a fixed depth) for a solitaire game.
- design classes to represent the state in a solitaire game.
- design move classes to represent allowable moves.
- explain the difference between a Breadth First and Depth First search tree.

Path Finding For Single-Player Games

8-puzzle is a solitaire game formed using a three-by-three grid containing eight square tiles numbered 1 to 8 and an empty space that contains no tile. A tile adjacent (either horizontally or vertically) to the empty space can be moved by sliding it into the empty space. The aim is to start from a shuffled initial state and move tiles to achieve a goal state (for example, with the numbers in clockwise order from the upper left corner, with the middle square being empty). There are no competing players taking alternate turns for these problems, but the behavior is quite similar to game trees.



A *search tree* represents the set of intermediate board states as the path-finding algorithm progresses. To be as efficient as possible, the path-finding algorithm must avoid visiting the same board state twice, otherwise it might get stuck in an infinite repetition of useless moves. The result of the computed search structure is a tree because the algorithm ensures that it does not visit a board state twice. The algorithm decides the order of board states to visit as it attempts to reach its goal.

In order to write a program to solve 8-puzzle, you need a class that represents the board state.



Create a new Java Project named **SinglePlayer** and assign it to the **Java6_Lessons** working set.



In the **/src** source folder, create a **puzzle** package.



In the **puzzle** package, create a **Board** class as shown:

CODE TO TYPE: Board class

```
package puzzle;

public class Board {
    int[][] tiles;

    public Board (int[][] initial) {
        tiles = new int[3][3];
        for (int r = 0; r < 3; r++) {
            for (int c = 0; c < 3; c++) {
                tiles[r][c] = initial[r][c];
            }
        }
    }

    public Board (Board b) {
        tiles = new int[3][3];
        for (int r = 0; r < 3; r++) {
            for (int c = 0; c < 3; c++) {
                tiles[r][c] = b.tiles[r][c];
            }
        }
    }

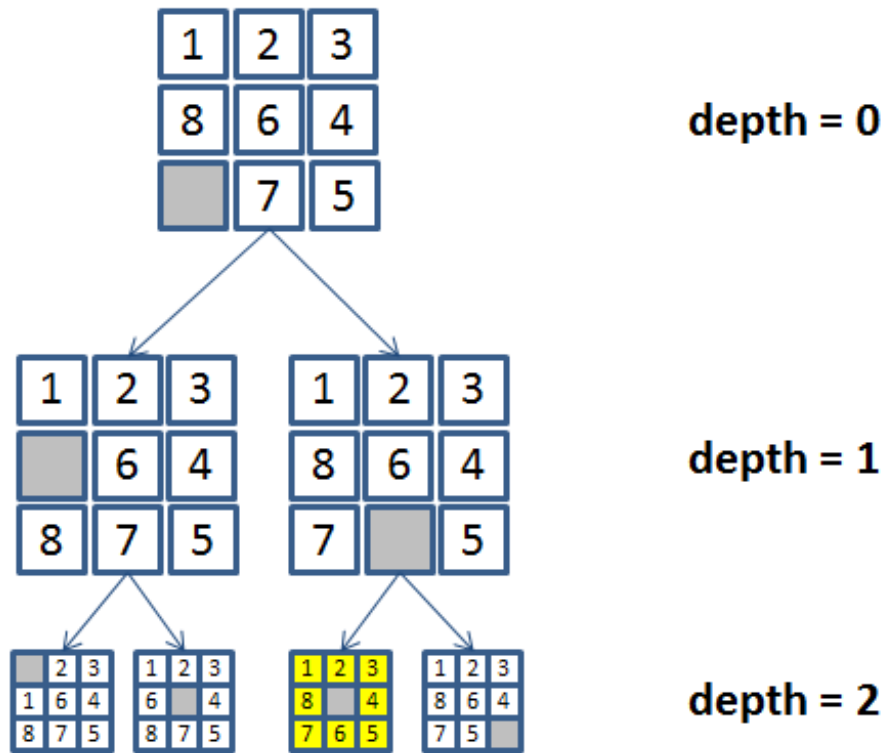
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for (int r = 0; r < 3; r++) {
            for (int c = 0; c < 3; c++) {
                if (tiles[r][c] == 0) {
                    sb.append('-');
                } else {
                    sb.append(tiles[r][c]);
                }
            }
            sb.append('\n');
        }
        return sb.toString();
    }
}
```

A two-dimensional array of **int** values, **tiles**, stores the location of each tile, using the number **0** to represent an empty tile. The first constructor passes in a sample state representation with three rows of three columns each. The second constructor makes a copy of a **Board** object. The **toString()** method prepares a human-readable string depicting the board.

The most common operation on a board state is to determine the valid moves where the player can slide a tile—horizontally or vertically—into the adjacent empty space. When the empty space is in the middle of the board, there are four possible moves, but when the empty space is in one of the corners of the board, only two moves are available.

If you have ever played a solitaire puzzle, you know how frustrating it can be to make (often random) long sequences of moves without knowing whether you are actually making real progress towards solving the problem! You could prevent repeating a series of moves if only you could remember whether you had visited a board state previously. In fact the algorithm presented here demands this functionality. The ability to detect whether a state has been visited is directly analogous to the ability to color vertices in a graph during *Depth-First Search*.

Let's try to solve a sample **8-puzzle** problem instance. The initial state reading from left to right is: **{{1,2,3}, {4,0,6}, {7,5,8}}** where **0** represents the empty square in the middle of the board. Let's make the final goal state **{{1,2,3}, {8,0,4}, {7,6,5}}** (highlighted in yellow below). The search tree below shows all six possible board states reachable in two or fewer moves from the initial state. The nodes of a search tree are the board states.



From the initial state (depth 0), there are two possible moves that result in two new board states on depth 1. In each of these two board states, there are three moves; however, depth 2 has only two children states for each. That's because the search tree *never contains any board states that have already been visited*. The goal of this lesson is to demonstrate how to automate this process to determine the sequence of moves that leads from the initial state to a desired goal state.

We've already used Depth-First Search to search through a graph. In this case, the algorithm constructs a search tree by blindly exploring ahead, choosing moves to make from the available set of moves in each board state. When using Depth-First Search on an actual graph, the algorithm backtracks when it runs out of new vertices to visit. However, the search tree continually expands as you execute moves to uncover new board states to explore. To avoid having a "runaway" *DepthFirstSearch*, let's introduce a new parameter, **maxDepth**, which determines the *maximum depth* to explore a particular branch of the search tree. If it doesn't reach the goal state by this maximum depth, it begins to backtrack to try alternate sequences of moves.

To keep track of the solution through the search tree, you need to make some modifications to the **Board** class structure:

CODE TO TYPE: Modifications to Board class

```
package puzzle;

import java.util.*;

public class Board {
    int[][] tiles;
    Board previous;
    int depth;

    ...
}
```

The **depth** attribute records the depth of a Board in the search tree, while **previous** will be a link to the previous Board in the search tree, which will always be the parent board state for each board state in the search tree.

This pseudocode presents an approach to solve solitaire puzzles like the 8-puzzle:

OBSERVE: pseudocode for solving solitaire puzzles with depth restriction

```
search(initial, goal, maxDepth)
    solution = {}
    if initial is goal then return "Solution"


    open = new Set
    closed = new Set
    insert (open, initial)
    while open is not empty do
        board = select state from open
        insert (closed, board)
        foreach valid move m at board do
            next = board state after playing m
            if closed doesn't contain next then
                if next = goal then return "Solution"
                else if not exceeded maxDepth then
                    insert (open, next)

    return "No Solution"
```

The algorithm maintains **open** and **closed** sets to guide the search. **open** represents the active horizon of the search, and contains board states that will be explored. **closed** remembers the past board states that were visited. The algorithm proceeds by **selecting a board state from open to explore**. Then it **generates potential board states to visit based on the available moves**. **If it hasn't exceeded its maximum depth** and the **newly generated board states haven't been visited**, the **board states are inserted into the open collection**.

The behavior of the search algorithm changes based on the strategy used to decide the next board state in **open** to process. As you've seen in past lessons, if you use a Stack to store the open states, the algorithm pursues a Depth-First Search approach; if you use a Queue, the algorithm implements *Breadth-First Search*.

To complete the implementation of 8-puzzle you need to create a class to represent a valid move.

 In the **puzzle** package, create a **SlideMove** class as shown:

CODE TO TYPE: SlideMove class

```
package puzzle;

public class SlideMove {
    final int fromR, fromC;
    final int toR, toC;

    public SlideMove (int fromR, int fromC, int toR, int toC) {
        this.fromR = fromR;
        this.fromC = fromC;
        this.toR = toR;
        this.toC = toC;
    }

    public boolean execute(Board b) {
        if (!isValid(b)) { return false; }
        b.swap(fromR, fromC, toR, toC);
        return true;
    }

    public boolean isValid(Board b) {
        if (fromR < 0 || fromR >= 3) { return false; }
        if (fromC < 0 || fromC >= 3) { return false; }
        if (toR < 0 || toR >= 3) { return false; }
        if (toC < 0 || toC >= 3) { return false; }

        return b.isAdjacentAndEmpty(fromR, fromC, toR, toC);
    }
}
```

Let's look at this class more closely. It won't compile just yet, but you'll soon add the required new methods to the

Board class.

OBSERVE:

```
public class SlideMove {
    final int fromR, fromC;
    final int toR, toC;

    public SlideMove (int fromR, int fromC, int toR, int toC) {
        this.fromR = fromR;
        this.fromC = fromC;
        this.toR = toR;
        this.toC = toC;
    }

    public boolean execute(Board b) {
        if (!isValid(b)) { return false; }
        b.swap(fromR, fromC, toR, toC);
        return true;
    }

    public boolean isValid(Board b) {
        if (fromR < 0 || fromR >= 3) { return false; }
        if (fromC < 0 || fromC >= 3) { return false; }
        if (toR < 0 || toR >= 3) { return false; }
        if (toC < 0 || toC >= 3) { return false; }

        return b.isAdjacentAndEmpty(fromR, fromC, toR, toC);
    }
}
```

A **SlideMove** represents the movement of a tile from a given (**fromR, fromC**) location to a destination (**toR, toC**) location. Such a move is valid **if the index values are all valid, there is an empty square in the (toR, toC) location, and the two locations are neighbors**. The move executes by **swapping the contents of the two locations**.

Add these two methods to the end of the **Board** class to enable **SlideMove** to compile:

CODE TO TYPE: Add methods to end of Board class

```
public boolean isAdjacentAndEmpty(int fromR, int fromC, int toR, int toC) {
    if (tiles[toR][toC] != 0) { return false; }

    int dC = Math.abs(fromR-toR);
    int dR = Math.abs(fromC-toC);
    if ((dC == -1 && dR == 0) || (dC == +1 && dR == 0) ||
        (dC == 0 && dR == -1) || (dC == 0 && dR == +1)) {
        return true;
    }

    return false;
}

public void swap (int fromR, int fromC, int toR, int toC) {
    int tmp = tiles[toR][toC];
    tiles[toR][toC] = tiles[fromR][fromC];
    tiles[fromR][fromC] = tmp;
}
```

OBSERVE:

```

public boolean isAdjacentAndEmpty(int fromR, int fromC, int toR, int toC) {
    if (tiles[toR][toC] != 0) { return false; }

    int dC = Math.abs(fromR-toR);
    int dR = Math.abs(fromC-toC);
    if ((dC == -1 && dR == 0) || (dC == +1 && dR == 0) ||
        (dC == 0 && dR == -1) || (dC == 0 && dR == +1)) {
        return true;
    }

    return false;
}

```

The **swap** method swaps the location of two tiles in the board. **isAdjacentAndEmpty** performs some calculations to determine if (**toR,toC**) represents an empty square neighboring another location (**fromR,fromC**) either horizontally or vertically. Variables **dC** and **dR** compute the difference of the indices and **a neighbor is just one column or row away (but not both)**.

The algorithm needs to know the valid moves at any given board state. Add the attribute below and **validMoves()** method to **Board** to return a **List** of the available sliding moves at that state:

CODE TO TYPE: Add attribute and method to the end of the Board class

```

static int deltas[][] = {{+1, 0}, {0, -1}, {-1, 0}, {0, 1}};

public List<SlideMove> validMoves() {
    int br = -1, bc = -1;

    for (int r = 0; r < 3; r++) {
        for (int c = 0; c < 3; c++) {
            if (tiles[r][c] == 0) {
                br = r;
                bc = c;
            }
        }
    }

    ArrayList<SlideMove> list = new ArrayList<SlideMove>();
    for (int i = 0; i < deltas.length; i++) {
        int dr = deltas[i][0];
        int dc = deltas[i][1];

        SlideMove sm = new SlideMove (br+dr, bc+dc, br, bc);
        if (sm.isValid(this)) { list.add(sm); }
    }

    return list;
}

```

This method determines the row and column of the empty space and then constructs an **ArrayList** object that represents the the available valid moves, using the **SlideMove** class designed earlier.

All the pieces are now ready to implement the Depth-First Search algorithm for solving the 8-puzzle problem. To implement a Depth-First Search you need to implement a search that uses a Stack to store the set of **open** states that have not yet been visited.

 In the **puzzle** package, create a **Search** class as shown:

CODE TO TYPE: Search class

```
package puzzle;

import java.util.*;

public class Search {
    public static Board depthFirst(Board initial, Board goal, int maxDepth) {
        if (initial.equals(goal)) { return initial; }

        Stack<Board> open = new Stack<Board>();
        HashSet<Board> closed = new HashSet<Board>();
        open.add(initial);
        while (!open.isEmpty()) {
            Board b = open.pop();
            closed.add(b);
            for (SlideMove sm : b.validMoves()) {
                Board next = new Board(b);
                sm.execute(next);

                next.previous = b;
                next.depth = b.depth + 1;

                if (next.equals(goal)) { return next; }

                if (!closed.contains(next)) {
                    if (next.depth < maxDepth) {
                        open.add(next);
                    }
                }
            }
        }

        return null;
    }
}
```

Let's break this code down:

OBSERVE: Initializing the Depth First Search algorithm

```
public static Board depthFirst(Board initial, Board goal, int maxDepth) {
    if (initial.equals(goal)) { return initial; }

    Stack<Board> open = new Stack<Board>();
    HashSet<Board> closed = new HashSet<Board>();
    open.add(initial);
}
```

If the initial board state is the goal state, the method exits immediately. Otherwise, it creates an **open** object, using the existing Stack class from the Java Collections Framework. As we've seen in earlier lessons, the stack is the fundamental structure to use when pursuing a depth-first algorithm, because it can save states to which the algorithm can backtrack as necessary.

The **closed** object is a HashSet that represents states that have already been visited. This object cannot simply be an **ArrayList** or **LinkedList** because the size of the **closed** set can be quite large and these classes only support $O(n)$ performance when checking whether the list contains an item. Also, you cannot use TreeSet immediately because that demands that the elements being added all implement Comparable; there is no immediate way to compare to board states to see which one is smaller (or larger). We'll use HashSet, which offers $O(1)$ constant time performance to locate an element. This object will represent a set because the algorithm will not visit the same state twice in the search tree.

For **HashSet** to work properly, the **Board** class must implement a specialized **hashCode** method. Specifically, if two **Board** state objects represent the same state, the **hashCode** method must return the same value. In an earlier lesson, we showed how the **String** class implements its **hashCode** method efficiently by caching the computed hash value. Here we take advantage of the fact that once a **Board** is added to the **closed** set, it never changes. When writing classes that are used as key values in the Java Collections Framework, you must provide both the **hashCode** method and a compatible **equals** method.

Modify the **Board** class as shown:

CODE TO TYPE: Modifications to Board

```
package puzzle;

import java.util.*;

public class Board {
    int[][] tiles;
    int hash;
    Board previous;
    int depth;

    ...

    public List<SlideMove> validMoves() {
        int br = -1, bc = -1;

        for (int r = 0; r < 3; r++) {
            for (int c = 0; c < 3; c++) {
                if (tiles[r][c] == 0) {
                    br = r;
                    bc = c;
                }
            }
        }

        ArrayList<SlideMove> list = new ArrayList<SlideMove>();
        for (int i = 0; i < deltas.length; i++) {
            int dr = deltas[i][0];
            int dc = deltas[i][1];

            SlideMove sm = new SlideMove (br+dr, bc+dc, br, bc);
            if (sm.isValid(this)) { list.add(sm); }
        }

        return list;
    }

    public boolean equals (Object o) {
        if (o == null) { return false; }
        if (!(o instanceof Board)) { return false; }
        Board other = (Board) o;
        for (int r = 0; r < 3; r++) {
            for (int c = 0; c < 3; c++) {
                if (tiles[r][c] != other.tiles[r][c]) { return false; }
            }
        }
        return true;
    }

    public int hashCode() {
        if (hash == 0) {
            for (int r = 0; r < 3; r++) {
                for (int c = 0; c < 3; c++) {
                    hash = 31*hash + tiles[r][c];
                }
            }
        }

        return hash;
    }
}
```

Let's review this code:

OBSERVE: equals method for Board

```
public boolean equals (Object o) {  
    if (o == null) { return false; }  
    if (!(o instanceof Board)) { return false; }  
  
    Board other = (Board) o;  
    for (int r = 0; r < 3; r++) {  
        for (int c = 0; c < 3; c++) {  
            if (tiles[r][c] != other.tiles[r][c]) { return false; }  
        }  
    }  
    return true;  
}
```


Two **Board** objects are equal if they contain the same arrangement of tiles. The structure is common to many **equals** methods you may have seen. First, it makes sure that the **object o is not null**, because the Java contract for **equals** states that no object is ever equal to null. Second, **any attempt to compare equality with a non-Board object must fail**. Finally, this method iterates over all tiles to determine whether any two corresponding locations contain different tiles, **returning false at the first difference between the two Board objects**:

OBSERVE: hashCode method for Board

```
public int hashCode() {  
    if (hash == 0) {  
        for (int r = 0; r < 3; r++) {  
            for (int c = 0; c < 3; c++) {  
                hash = 31*hash + tiles[r][c];  
            }  
        }  
    }  
  
    return hash;  
}
```

The **hashCode** method uses the **hash** attribute to cache the computed hash value. This method computes the hash by using the same multiplicative function found in the String class. For example, the hash value computed for the goal state is **274869244**.

Everything is in place to try solving an initial board state.

 In the **puzzle** package, create a **Main** class as shown:

CODE TO TYPE: Main class for searching

```
package puzzle;

public class Main {
    public static void printSolution(Board goal) {
        if (goal == null) {
            System.out.println("No Solution reached");
        } else {
            int count = -1;
            while (goal != null) {
                System.out.println(goal);

                goal = goal.previous;
                count++;
            }
            System.out.println(count + " total moves");
        }
    }

    public static void main(String[] args) {
        Board initial = new Board(new int[][]{{1,2,3}, {8,6,4}, {0,7,5}});
        Board goal = new Board(new int[][]{{1,2,3}, {8,0,4}, {7,6,5}});

        int maxDepth = 8;
        Board result = Search.depthFirst(initial, goal, maxDepth);
        printSolution(result);
    }
}
```



Save and run it to confirm the solution found earlier. The boards are displayed in reverse order, from the *goal* state all the way back to the initial position:

OBSERVE: Sample Execution of DepthFirstSearch algorithm on search tree

```
123
8-4
765

123
864
7-5

123
864
-75

2 total moves
```

Breadth-First Search

With only marginal changes, you can create an implementation that explores the search tree in Breadth-First fashion. Add this method to the end of the **Search** class:

CODE TO TYPE: Modifications to Search class

```
package puzzle;

import java.util.*;

public class Search {
    public static Board depthFirst(Board initial, Board goal, int maxDepth) {
        if (initial.equals(goal)) { return initial; }

        Stack<Board> open = new Stack<Board>();
        HashSet<Board> closed = new HashSet<Board>();
        open.add(initial);
        while (!open.isEmpty()) {
            Board b = open.pop();
            closed.add(b);
            for (SlideMove sm : b.validMoves()) {
                Board next = new Board(b);
                sm.execute(next);

                next.previous = b;
                next.depth = b.depth + 1;

                if (next.equals(goal)) { return next; }

                if (!closed.contains(next)) {
                    if (next.depth < maxDepth) {
                        open.add(next);
                    }
                }
            }
        }

        return null;
    }

    public static Board breadthFirst(Board initial, Board goal) {
        if (initial.equals(goal)) { return initial; }

        Queue<Board> open = new LinkedList<Board>();
        HashSet<Board> closed = new HashSet<Board>();
        open.add(initial);
        while (!open.isEmpty()) {
            Board b = open.remove();
            closed.add(b);
            for (SlideMove sm : b.validMoves()) {
                Board next = new Board(b);
                sm.execute(next);

                next.previous = b;
                next.depth = b.depth + 1;

                if (next.equals(goal)) { return next; }

                if (!closed.contains(next)) {
                    open.add(next);
                }
            }
        }

        return null;
    }
}
```

The code is identical to **depthFirst**, except that it uses a **Queue** (implemented by **LinkedList**) to store the *open* board states, and it removes the next board state from *open* using the **remove** method. The behavior is very different from **depthFirst** though; it methodically inspects all board states in increasing distance from

the initial state, based on the number of moves that are used.

Modify **Main** to report on Breadth-First Search results as well:

CODE TO TYPE: Modifications to Main class

```
package puzzle;


public class Main {
    public static void printSolution(Board goal) {
        if (goal == null) {
            System.out.println("No Solution reached");
        } else {
            int count = -1;
            while (goal != null) {
                System.out.println(goal);

                goal = goal.previous;
                count++;
            }
            System.out.println(count + " total moves");
        }
    }

    public static void main(String[] args) {
        Board initial = new Board(new int[][]{{1,2,3}, {8,6,4}, {0,7,5}});
        Board goal = new Board(new int[][]{{1,2,3}, {8,0,4}, {7,6,5}});

        int maxDepth = 8;
        Board result = Search.depthFirst(initial, goal, maxDepth);
        printSolution(result);

        Board bfsResult = Search.breadthFirst(initial, goal);
        printSolution(bfsResult);
    }
}
```

 Save and run it.

OBSERVE: Compare DFS and BFS on simple board

```
123
8-4
765

123
864
7-5

123
864
-75

2 total moves
123
8-4
765

123
864
7-5

123
864
-75

2 total moves
```

When running on a board state that is only two moves removed from the solution, both approaches locate the solution quickly, but which one is more efficient? You can judge efficiency in terms of speed of execution, but also evaluate the efficiency of a search by comparing the size of the board states visited (**closed**) as well as the size of the board states yet to be processed (**open**). You need to *instrument* the **Search** and **Main** classes to record this information.

Make these changes to **Search**:

CODE TO TYPE: Modifications to Search

```
package puzzle;

import java.util.*;

public class Search {
    static int numDFSOpen = 0;
    static int numDFSProcessed = 0;
    static int numBFSOpen = 0;
    static int numBFSProcessed = 0;

    public static Board depthFirst(Board initial, Board goal, int maxDepth) {
        if (initial.equals(goal)) { return initial; }

        Stack<Board> open = new Stack<Board>();
        HashSet<Board> closed = new HashSet<Board>();
        open.add(initial);
        numDFSOpen=1;
        numDFSProcessed=0;
        while (!open.isEmpty()) {
            Board b = open.pop();
            numDFSOpen--;
            numDFSProcessed++;
            closed.add(b);
            for (SlideMove sm : b.validMoves()) {
                Board next = new Board(b);
                sm.execute(next);

                next.previous = b;
                next.depth = b.depth + 1;

                if (next.equals(goal)) { return next; }

                if (!closed.contains(next)) {
                    if (next.depth < maxDepth) {
                        numDFSOpen++;
                        open.add(next);
                    }
                }
            }
        }

        return null;
    }

    public static Board breadthFirst(Board initial, Board goal) {
        if (initial.equals(goal)) { return initial; }

        Queue<Board> open = new LinkedList<Board>();
        HashSet<Board> closed = new HashSet<Board>();
        open.add(initial);
        numBFSOpen=1;
        numBFSProcessed=0;
        while (!open.isEmpty()) {
            Board b = open.remove();
            numBFSOpen--;
            numBFSProcessed++;
            closed.add(b);
            for (SlideMove sm : b.validMoves()) {
                Board next = new Board(b);
                sm.execute(next);

                next.previous = b;
                next.depth = b.depth + 1;

                if (next.equals(goal)) { return next; }
            }
        }
    }
}
```

```

        if (!closed.contains(next)) {
            numBFSOpen++;
            open.add(next);
        }
    }
}

return null;
}
}

```

The changes update two counts for each algorithm, recording the number of board states in the **open** state and the number of board states that were processed.

Make these changes to **Main** to display this information for the individual runs:

CODE TO TYPE: Modifications to Main class

```

package puzzle;

public class Main {
    public static void printSolution(Board goal) {
        if (goal == null) {
            System.out.println("No Solution reached");
        } else {
            int count = -1;
            while (goal != null) {
                System.out.println(goal);

                goal = goal.previous;
                count++;
            }
            System.out.println(count + " total moves");
        }
    }

    public static void main(String[] args) {
        Board initial = new Board(new int[][]{{1,2,3}, {8,6,4}, {0,7,5}});
        Board goal = new Board(new int[][]{{1,2,3}, {8,0,4}, {7,6,5}});

        int maxDepth = 8;
        Board result = Search.depthFirst(initial, goal, maxDepth);
        printSolution(result);
        System.out.println("DFSOpen=" + Search.numDFSOpen + ", DFSProcessed=" + Search.numDFSProcessed);

        Board bfsResult = Search.breadthFirst(initial, goal);
        printSolution(bfsResult);
        System.out.println("BFSOpen=" + Search.numBFSOpen + ", BFSProcessed=" + Search.numBFSProcessed);
    }
}

```



Save and run it to generate statistics for this initial test:

OBSERVE: Sample statistics for trivial board state
<pre> 123 8-4 765 123 864 7-5 123 864 -75 2 total moves DFSOpen=1, DFSProcessed=2 123 8-4 765 123 864 7-5 123 864 -75 2 total moves BFSOpen=2, BFSProcessed=3 </pre>

In both cases, a solution of 2 moves was found, but Breadth-First Search had more states in its **open** set to be considered for the future and it also processed one more board state than Depth-First Search.

Evaluating Search Tree Algorithms

To evaluate these search algorithms properly, you have to generate initial board states from which to search for the goal state. However, you can't just randomly assign the eight digits and the empty space to a **Board**, because there may be no way to slide the tiles to achieve the goal state from the randomly selected initial state. The solution is to make a random number of moves from some initial state, and then let the algorithms try to search their way back to the original board state. Let's get started on that infrastructure now.

 In the **puzzle** package, create a **Generate** class as shown:

CODE TO TYPE: Generate class

```
package puzzle;

import java.util.*;

public class Generate {
    public static Board generate(int n) {
        Board state = new Board(new int[][]{{1,2,3}, {8,0,4}, {7,6,5}});
        Set<Board> visited = new HashSet<Board>();
        visited.add(new Board(state));
        for (int i = 0; i < n; i++) {
            List<SlideMove> moves = state.validMoves();
            Collections.shuffle(moves);

            Board next = null;
            for (SlideMove sm : moves) {
                next = new Board(state);
                sm.execute(next);
                if (!visited.contains(next)) {
                    visited.add(next);
                    break;
                }
            }

            if (state.equals(next)) {
                System.err.println("Unable to generate " + n + " moves");
                return null;
            }
            state = next;
        }

        return state;
    }
}
```

Let's take a closer look at this code.

OBSERVE:

```
Board state = new Board(new int[][]{{1,2,3}, {8,0,4}, {7,6,5}});
Set<Board> visited = new HashSet<Board>();
visited.add(new Board(state));
for (int i = 0; i < n; i++) {
    List<SlideMove> moves = state.validMoves();
    Collections.shuffle(moves);
```

Starting from the goal state **state**, we instantiate a **visited** set to make sure that we properly generate a total of n steps without revisiting a previously visited board state. The **for** loop **generates a list of potential moves from the given state** and **shuffles this list** so the moves are investigated in random order.

OBSERVE: Making a random move

```
Board next = null;
for (SlideMove sm : moves) {
    next = new Board(state);
    sm.execute(next);
    if (!visited.contains(next)) {
        visited.add(next);
        break;
    }
}
```

The code above **tries each move, one at a time**, to **see if it generates a new board state that has not already been visited**. With each pass through the loop, it **creates a copy of the state board state in next** so the move is executed on the copy without affecting the original **state**. If we find an unvisited board

state, we **break** out of the **for** loop; otherwise, it repeats until all moves are exhausted. To complete the loop, review this logic:

OBSERVE: Identifying when not possible to generate board

```
if (state.equals(next)) {
    System.err.println("Unable to generate " + n + " moves");
    return null;
}
state = next;
```

When this method returns **null**, it means that it was unsuccessful in locating a board state n moves away; the only reason for failure is that all board states with fewer number of moves were visited. If the computed **next** state is not the same as **state** though, it **sets state to next** and advances to try another move. Once the appropriate number of moves has been applied, the method returns a sample board state n moves away.

Demonstrate the use of this **Generate** method by modifying the **Main** class as shown:

CODE TO TYPE: Modifications to Main class

```
package puzzle;


public class Main {
    public static void printSolution(Board goal) {
        if (goal == null) {
            System.out.println("No Solution reached");
        } else {
            int count = -1;
            while (goal != null) {
                System.out.println(goal);

                goal = goal.previous;
                count++;
            }
            System.out.println(count + " total moves");
        }
    }

    public static void main(String[] args) {
        Board initial = new Board(new int[][]{{1,2,3}, {0,6,4}, {0,7,5}});
        Board initial = Generate.generate(6);
        Board goal = new Board(new int[][]{{1,2,3}, {8,0,4}, {7,6,5}});

        int maxDepth = 9;
        Board result = Search.depthFirst(initial, goal, maxDepth);
        printSolution(result);
        System.out.println("DFSOpen=" + Search.numDFSOpen + ", DFSProcessed=" + Search.numDFSProcessed);

        Board bfsResult = Search.breadthFirst(initial, goal);
        printSolution(bfsResult);
        System.out.println("BFSOpen=" + Search.numBFSOpen + ", BFSProcessed=" + Search.numBFSProcessed);
    }
}
```

 Run the revised **Main** class; you'll get different results based on the generated board state. Here is a sample run where a Depth-First approach finds an 8-move solution (after processing just 21 board states), while a Breadth-First approach finds a minimal 6-move solution (after processing 45 board states).

OBSERVE: Sample run comparing Breadth-First and Depth-First

123
8-4
765

123
-84
765

-23
184
765

2-3
184
765

283
1-4
765

283
-14
765

-83
214
765

8-3
214
765

83-
214
765

8 total moves
DFSOpen=5, DFSProcessed=21

123
8-4
765

1-3
824
765

-13
824
765

813
-24
765

813
2-4
765

8-3
214
765

83-
214
765

```
6 total moves  
BFSOpen=32, BFSProcessed=45
```

Run it multiple times and compare the results to see low, high, and average results. In general, the Breadth-First approach will compute the shortest number of moves to achieve the goal state, but it will process far more states than the Depth-First approach. However, Depth-First is a blind algorithm and will generate solutions with perhaps hundreds or thousands of moves if you don't set **maxDepth**—but there may not be an easy way to determine the proper value to use for **maxDepth** because that implies that you have (more or less) an idea as to how many moves away you are.

Lessons Learned

In this lesson you learned:

- how to use a **Queue** structure to impose a Breadth-First approach when inserting and removing states to search from the **open** set.
- how to use a **Stack** structure to impose a Depth-First approach when inserting and removing states to search.
- that the **HashSet** class from the Java Collections Framework provides $O(1)$ constant performance for determining whether the set contains a given element.
- how to avoid the need to implement an undo method in the moves by using a constructor to copy the existing board state to which the moves execute their changes. This behavior is distinctly different from the game trees from the upcoming two-player lesson.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Path Finding for Two-Player Games

Lesson Objectives

After completing this lesson you will be able to:

- describe the structure of a game tree for two-player games.
 - explain how Minimax computes best move for player in a game tree.
 - explain how an evaluation function guides Minimax to choosing the best move.
-

Path Finding For Two-Player Games

Chopsticks is a two-player hand game. The version presented here is just one out of many possible variations. Each player uses both hands, each of which can represent the values 0 to 4 by the number of extended fingers on that hand (thumbs are not involved). To start, both players extend the index finger on each hand. The players alternate turns, and when it is his turn, a player may:

- **Tap**
- **Rebalance**

To **tap**, a player taps the hand of an opponent with one of his hands. When this happens the player's "points" in the tapping hand (represented by the number of extended fingers) are added to the opponent's tapped hand; the number of points in the player's hand does not change. Once a hand has five or more points, the player closes the hand into a fist and the hand is considered to be "dead" with zero points.

To **rebalance**, a player must have one "dead" hand, and one hand with an even number of points. On his turn, the player taps his own "dead" hand with his other hand, and the points are evenly split between both hands.

The goal of each player is to force his opponent to have two "dead" hands.

This lesson shows how to use a *path finding technique* from Artificial Intelligence to compute the best move for a player, that is, the move that has the highest likelihood of leading that player to victory. To solve this kind of problem you have to frame the problem computationally and correctly. At any given moment, the state of the game can be represented by:

- which player's turn it is (Player1 or Player2).
- Player1's left hand points (0, 1, 2, 3, or 4).
- Player1's right hand points (0, 1, 2, 3, or 4).
- Player2's left hand points (0, 1, 2, 3, or 4).
- Player2's right hand points (0, 1, 2, 3, or 4).

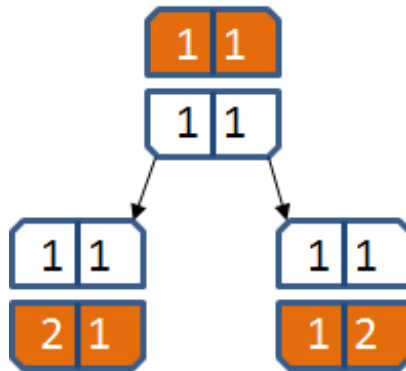
So, there are $2 \times 5 \times 5 \times 5 \times 5 = 1,250$ unique states of the game. Consider creating a *game tree* with nodes that represent valid states of the chopsticks game as the players take their turns. Given a particular game state, there are no more than five possible moves that can be made at any time:

- Rebalance.
- Use left hand to tap opponent's left hand.
- Use left hand to tap opponent's right hand.
- Use right hand to tap opponent's left hand.
- Use right hand to tap opponent's right hand.

To construct the game tree from a given starting position, create a root node that represents the initial game state of **(Player1, 1, 1, 1, 1)**. In the graphic below, Player1 (the starting player) is depicted on top and his opponent (Player2) is on the bottom. The two numbers, from left to right, represent the points on the left and right hand, respectively. Because it's Player1's turn (the top player), his state is highlighted in orange. This is a convenient way to visualize the state **(Player1, 1, 1, 1, 1)**.

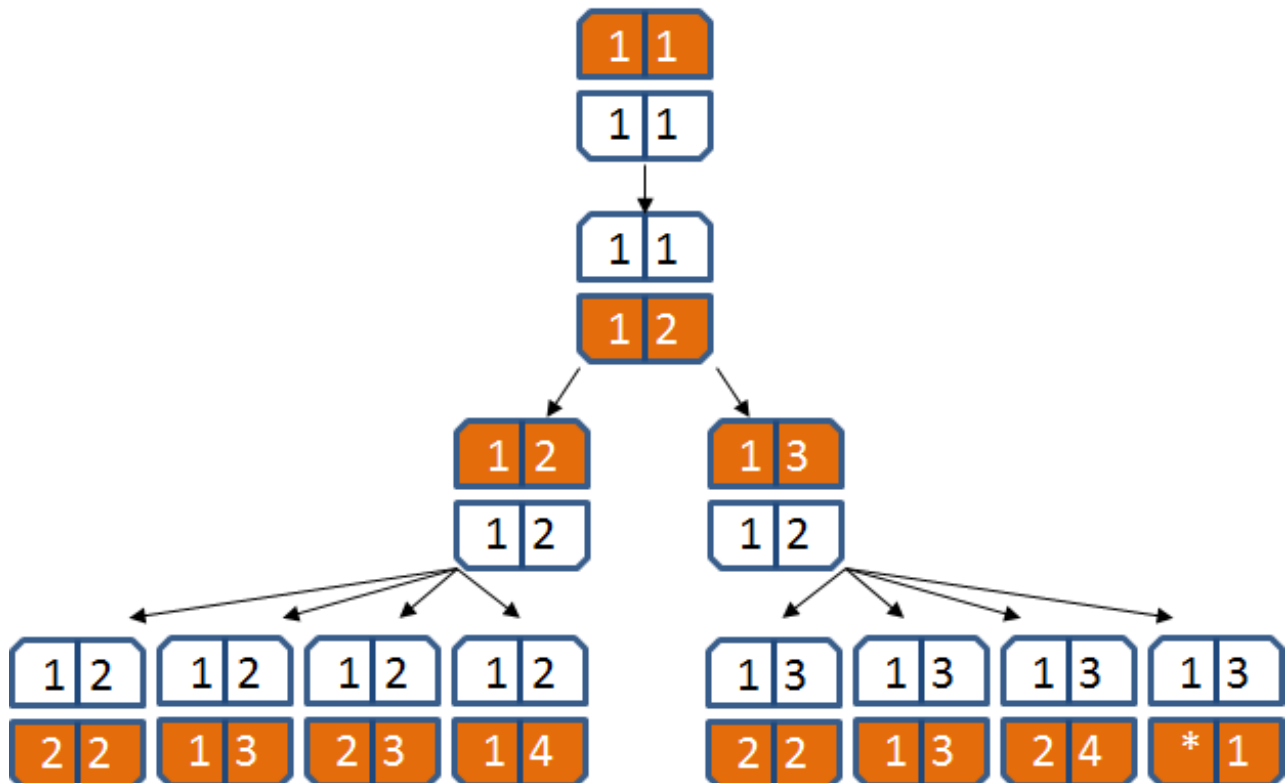


The game tree is expanded by adding child nodes to the leaf nodes in the tree (the root node of the initial game tree is a leaf node). The game tree expands based upon the allowed moves for the active player. There are four possible tapping moves here, but these result in only two distinct game states: if Player1 uses his left (or right) hand to tap the left hand of Player2, the resulting state is **(Player2, 1, 1, 2, 1)**, which would become the left child in the game tree. If Player1 uses his left (or right) hand to tap the right hand of Player2, the resulting state is **(Player2, 1, 1, 1, 2)** which would become the right child in the game tree. Note that in each of these two nodes, it will be Player2's turn.



At this point, observe that these two child nodes are really equivalent. If a player has X points on the left hand and Y points on the right hand, this is equivalent to having Y points on the left hand and X points on the right hand. So, you can combine these child nodes into one. For consistency, a player's hand can be defined by two values (**Low,High**) such that **Low <= High**. So, the game state is fully captured by **(Player#, Player1Low, Player1High, Player2Low, Player2High)**, where **Player1Low <= Player1High** and **Player2Low <= Player2High**.

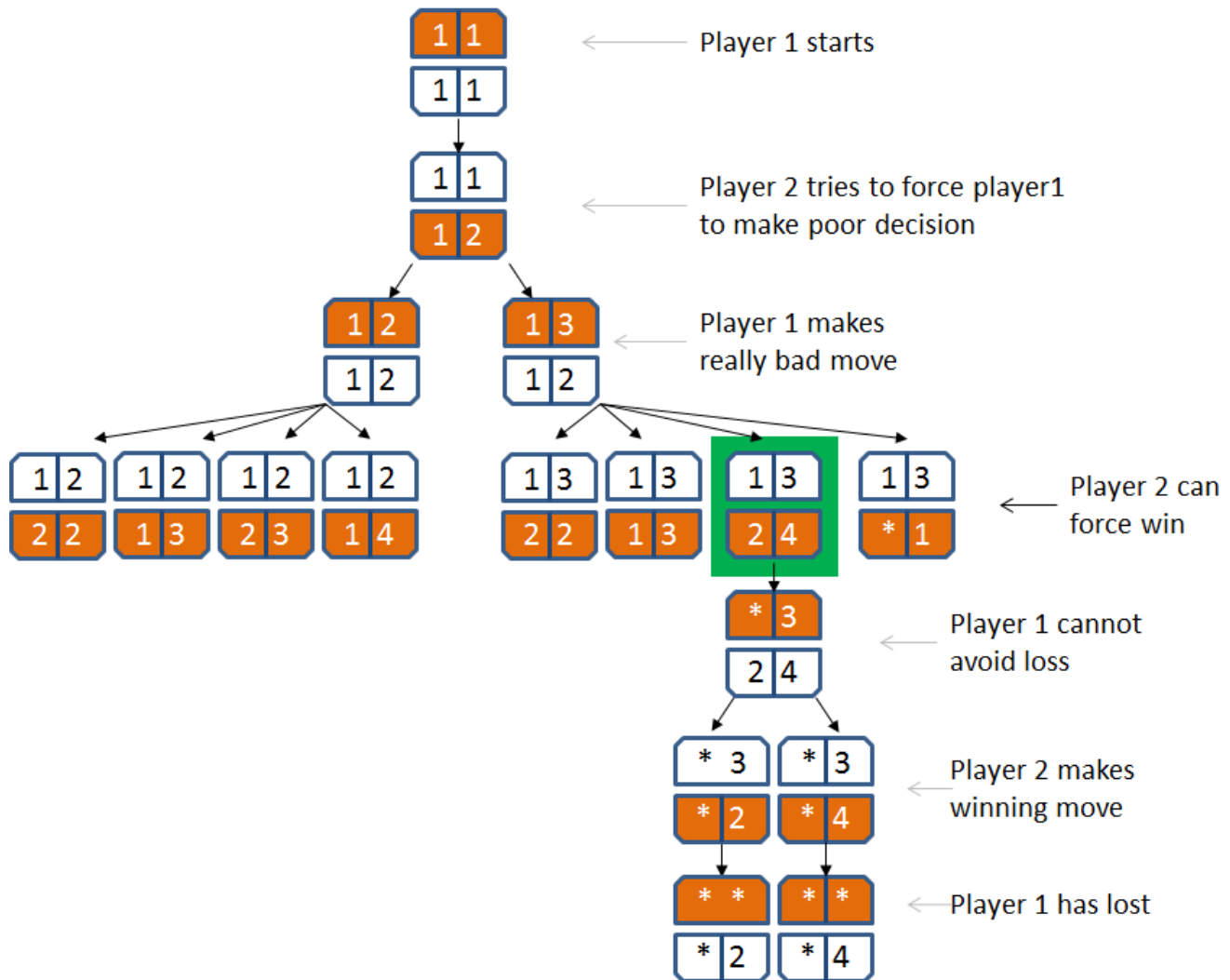
The above game tree has one level below the root node. Each successive level is known as a *ply* and represents a set of game states associated with the same active player. For example, the root node (often called ply 0) represents the active state for Player1, while the next level (ply 1) represents the active states for Player2. Let's expand the tree so there are four levels (ply 0 through ply 3):



The eight nodes in level 3 of this game tree represent all possible game states after three moves have been made.

The game tree is expanded by adding children to nodes, so the resulting structure will have no cycles. The final level in the tree represents the total number of possible states after $p=3$ moves. As you can see, it's possible for Player2 to have a "dead hand" after just three moves. You can find this state above because it uses the "*" character to indicate that one of the player's hands has no points. Each level (or ply) of the tree contains nodes with an active player that is the same. Even though there are five possible moves in each game state, not every node is expanded to have five children nodes. The size of the game tree is ultimately determined by an *expansion factor* k which is the average number of child nodes for any node in the game tree; for chopsticks, $k \leq 5$.

The above image demonstrates how large these trees can grow, even for simple games. If we looked eight moves ahead (ply=8), the game tree would contain up to 488,280 nodes if every node expanded by five children. The goal is to select the best available move for a player in a given game state. In the chopsticks game tree above, Player1 can make only one move in the initial state, so there is really no decision to make. However, Player2 can try to force Player1 to make a bad decision that would directly lead to Player2's victory. Let's expand one part of the game tree above to show this situation:



The highlighted state above is **(Player2, 1, 3, 2, 4)**. It's Player2's turn in this state, so she can guarantee a victory by using her hand with 4 fingers to tap Player1's hand with 1 finger. This results in the state **(Player1, *, 3, 2, 4)**. Player1 would then have only two possible moves: (1) tap Player2's hand with two fingers to create a dead hand; or (2) tap Player2's hand with 4 fingers to create a dead hand. However, in both of these situations, Player2 can simply tap Player1's remaining hand with three fingers to make it a dead hand, thus forcing Player1 to have two dead hands, and victory is guaranteed.

The game tree represents the full set of potential game states that result from a sequence of valid moves from the initial state; due to its size, it may never be computed fully. The goal of a path-finding algorithm is to determine from a starting game state, the player's move that maximizes (or even guarantees) his chance of winning the game. So we transform an intelligent set of player decisions into a *path-finding problem* over the game tree. This approach works for games with small game trees, but it also can be scaled to solve more complex problems.

Given a node representing the current game state, the algorithm computes the best move for a player. Instead of considering only the current game state and available moves at that state, the program must consider any countermoves that the opponent will make after each move. The program must assume that the opponent will select his best move choice and make no mistakes. To make this work computationally, we need a function that can evaluate a game state objectively and return an integer "score" value for that state. Smaller integer numbers (even negative ones) reflect weaker positions, while larger integer numbers (including positive infinity) represent stronger positions for the player.

Given a specific game state, it is easy to determine whether Player1 or Player2 has lost the game. For example, one of the final states above is (**Player1**, *, *, *, **4**) so from Player1's perspective, this is a loss and the evaluation function must rate this board as **-Infinity**. However, from Player2's point of view, this state is a win so the evaluation function would rate the same board as **+Infinity**. Clearly, it matters from whose perspective the board is evaluated.

The first steps we'll take will be to write code that represents the game state. Here are two ways we could accomplish that task:

- Represent state with four **int** attributes: **playerLow**, **playerHigh**, **opponentLow**, **opponentHigh**.
- Represent state with two **int** arrays: **playerPoints[]** and **opponentPoints[]**.

The first option above is inefficient, because you'd have to write special code constantly to compare the four different sums that result from adding together the different possible values. The second option uses arrays which is better (because you could use nested **for** loops to compute the sums), but you would spend a lot of time keeping these two values in sorted order to take advantage of the earlier observation regarding game state.

Use an array of **TreeSet** objects to maintain the **Set** of points for the players; when a player has two hands with the same value, only one value is stored for that player's hand. This allows you to write simpler code, although the code may look slightly complicated the first time that you see it.



Create a new Java Project named **TwoPlayer** and assign it to the **Java6_Lessons** working set.



In your **TwoPlayer** project **/src** source folder, create a **chopsticks** package.



In the **chopsticks** package, create a **GameState** class as shown:

CODE TO TYPE: GameState class

```
package chopsticks;

import java.util.*;

public class GameState {
    public static final int Player1 = 0;
    public static final int Player2 = 1;

    int player;
    Set<Integer> values[] = new TreeSet[2];

    public GameState (int player, int left1, int right1, int left2, int right2) {
        this.player = player;

        values[0] = new TreeSet<Integer>();
        values[0].add(left1);
        values[0].add(right1);

        values[1] = new TreeSet<Integer>();
        values[1].add(left2);
        values[1].add(right2);
    }

    public boolean hasWon(int p) {
        return values[1-p].size() == 1 && values[1-p].contains(0);
    }

    public String toString() {
        Iterator<Integer> pValues = values[0].iterator();
        Iterator<Integer> oppValues = values[1].iterator();

        StringBuilder sb = new StringBuilder("(Player").append(1+player).append(",");
        int left1 = pValues.next();
        sb.append(left1).append(",");
        if (pValues.hasNext()) { sb.append(pValues.next()); } else { sb.append(left1); }

        int left2 = oppValues.next();
        sb.append(",").append(left2).append(",");
        if (oppValues.hasNext()) { sb.append(oppValues.next()); } else { sb.append(left2); }

        return sb.append(")").toString();
    }
}
```


Each game state must store the current **player** in that situation, using **0** for Player1 and **1** for Player2. These values were chosen to make it easier to index into the **Set<Integer> values[]** array that stores the **TreeSet** objects. By using sets, the game state uses the optimization presented earlier where it only records one value when both hands have the same value. The **hasWon(p)** method determines whether a player **p** has won in a game state; this happens when that player's opponent (**1-p**) has only one value (**values[1-p].size() == 1**) and that value is **0**.

The **toString()** helper method is used only during debugging. It builds up a string representing the game state by iterating over the values in each hand. For efficiency, it uses the **StringBuilder** class.

Aside from an outright victory, how is it possible to compute a number that increases in value when a game state is more likely to lead to victory for a given player? You must develop a *heuristic* based on properties of the game state. Let's start with some observations:

- Having one dead hand is bad.
- Having hands with 4 points is more risky than having hands with 1 point.
- A player has a strong position when one of her hands can make an opponent's hand dead (and when the opponent can do this, the player's hand is weaker).

Instead of including an evaluation method inside **GameState**, define an interface to be used by any evaluation function. This lets you to experiment with different evaluation functions quickly.

 In the `/src` source folder **chopsticks** package, create an **IEvaluate** interface as shown:

CODE TO TYPE: IEvaluate interface

```
package chopsticks;

public interface IEvaluate {
    int evaluate(GameState state, int player);
}
```

 In the **chopsticks** package, create an **Evaluator** class that implements the heuristics defined earlier:

CODE TO TYPE: Evaluator class

```
package chopsticks;

public class Evaluator implements IEvaluate {
    public int evaluate(GameState s, int p) {
        if (s.hasWon(p)) { return 10000; }
        if (s.hasWon(1-p)) { return -10000; }

        int sign = 1;
        if (s.player == 1-p) { sign = -1; }

        int value = 0;

        // Having one dead hand is bad
        if (s.values[s.player].contains(0)) { value -= sign*100; }
        if (s.values[1-s.player].contains(0)) { value += sign*100; }


        // Having hands with 4 points is more risky than having hands with 1 point (scale b
        y 5 pts)
        for (int pt : s.values[s.player]) {
            value -= sign*pt*5;
        }
        for (int pt : s.values[1-s.player]) {
            value += sign*pt*5;
        }


        // Player has strong position when one of his hands can make an opponent's hand dea
        d
        int numMakeDead = 0;
        for (int pt1 : s.values[s.player]) {
            for (int pt2 : s.values[1-s.player]) {
                if (pt1 + pt2 >= 5) { numMakeDead++; }
            }
        }
        value += sign * numMakeDead * 20;

        return value;
    }
}
```

The **evaluate(state,p)** method is symmetric; that is, **evaluate(state,Player1) = -evaluate(state,Player2)**. The goal of this method is to make it possible to compare the computed evaluation of two game states, `gs1` and `gs2`, to determine which is more favorable to the current player. Accordingly, a victory is a clearly identified 10000 (used in place of infinity since no number computed by the **evaluate** method will ever be larger than this value). The code defines a **sign** variable that determines whether the resulting computation is negative (worse for the player **p**) or positive (better for the player **p**).

When developing heuristics, it is imperative that you write test cases so you can track changes to those heuristics. Many of the constants were chosen arbitrarily and you will have to experiment with minor tweaks, so test cases will prove very useful.

 In your **TwoPlayer** project, create a **/test** source folder.

 In the **/test** source folder, create a **chopsticks** package.

 In the **chopsticks** package, create a **TestGameState** JUnit test case as shown:

CODE TO TYPE: TestGameState JUnit class

```
package chopsticks;

import junit.framework.TestCase;

public class TestGameState extends TestCase {

    public void testWinning() {
        GameState gs = new GameState(GameState.Player1, 0, 0, 0, 2);
        assertTrue (gs.hasWon(GameState.Player2));
    }

    public void testNotWinning() {
        GameState gs = new GameState(GameState.Player1, 1, 1, 1, 1);
        assertFalse (gs.hasWon(GameState.Player1));
        assertFalse (gs.hasWon(GameState.Player2));
    }

    public void testFourBoards() {
        GameState[] states = {
            new GameState(GameState.Player2, 1, 3, 1, 3),
            new GameState(GameState.Player2, 1, 3, 0, 1),
            new GameState(GameState.Player2, 1, 3, 2, 2)
        };

        GameState worst = new GameState(GameState.Player2, 1, 3, 2, 4);
        IEvaluate eval = new Evaluator();
        int worstRating = eval.evaluate(worst, GameState.Player1);
        System.out.println("Worst State:" + worstRating + " " + worst);

        System.out.println("Other Moves:");
        for (GameState gs : states) {
            int gsRating = eval.evaluate(gs, GameState.Player1);
            System.out.println(gsRating + " " + gs);
            assertTrue (eval.evaluate(gs, GameState.Player1) <= gsRating);
        }
    }
}
```

This test case ensures that the **hasWon(p)** method works properly. It also compares the four game state children on the right side of the game tree depicted earlier to validate that the worst state evaluates to a number that is **smallest** of the other three sibling states. In other words, this should identify that this state is the worst possible arrangement from Player1's point of view. You must be sure that you represent the board states accurately, as well as the player for whom the evaluation is being made. In **testFourBoards**, all **GameState** objects are associated with Player2 and **evaluate()** is called with Player1 as an argument, because the originating node in the game tree is Player1.

Run the test case now and check the output:

OBSERVE: Game State evaluation scores

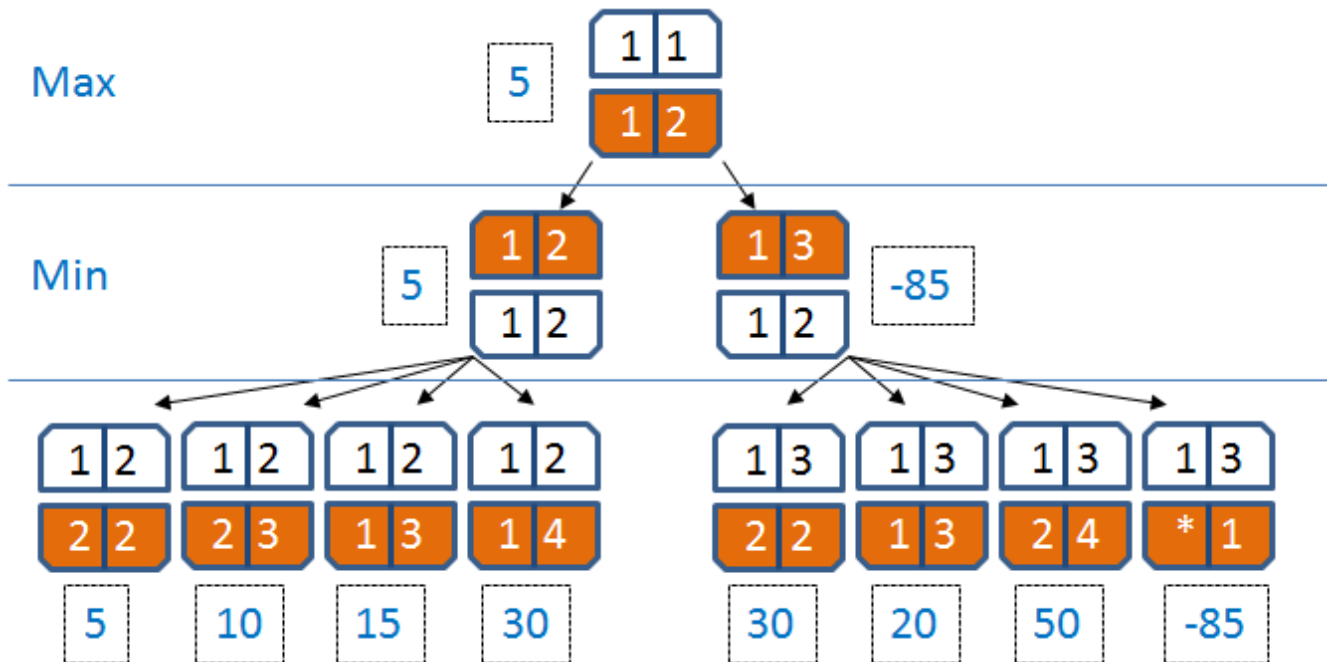
```
Worst State:-50 (Player2,1,3,2,4)
Other Moves:
-20 (Player2,1,3,1,3)
85 (Player2,1,3,0,1)
-30 (Player2,1,3,2,2)
```

When evaluating these four game states from the perspective of Player1, the **(Player2, 1, 3, 2, 4)** game state is rated the lowest.

The game tree is expanded by considering future game states after *n* moves have been made. Each level of the tree alternates between MAX levels (where the goal is to benefit the player by maximizing the evaluated score of a game

state) and MIN levels (where the goal is to benefit the opponent by minimizing the evaluated score of a game state). So, the levels alternate between MAX and MIN levels, which leads to an algorithm known as *Minimax*. In all cases, the board is evaluated *from the point of view of the player making the original move in the game tree* (that is, the active player in ply 0).

The next game tree shows, in dashed boxes, the score of the evaluation function on each leaf node in a 2-ply game tree starting from the **(Player2, 1, 1, 1, 2)** state where it is Player2's turn. Only leaf nodes are evaluated. Interior nodes on levels marked **Max** receive the maximum score of their children nodes. Similarly, interior nodes on levels marked **Min** receive the minimum score of their children nodes.



Of the eight nodes on the ply-2 level, the state **(Player2, 1, 3, 2, 4)** mentioned earlier is the highest-rated state for Player2 with a rating of **50**. However, the lowest rated state **(Player2, 1, 3, *, 1)** with a rating of **-85** is a sibling of this state. In the **Min** level in state **(Player1, 1, 3, 1, 2)**, it is Player1's turn to make a move. The algorithm must assume that an opponent plays without making mistakes; thus given the chance, Player1 would force Player2 into the lowest rated state. For this reason, the interior nodes on the ply-1 level are rated as **5** and **-85** respectively, representing the worst positions that Player2 would find himself in after Player1 moves. Finally, the root node on ply 0 selects the move that maximizes the evaluation of its children nodes, thus the algorithm would choose the **Tap** move that results in the state **(Player2, 1, 2, 1, 2)**.

Constructing the game tree above does not include writing code to automate this process. This pseudocode describes that process:

OBSERVE: pseudocode for Minimax

```
bestMove (s, player)
    original = player
    [move, score] = minimax(s, ply, MaxLevel)
    return move

minimax (s, ply, player, opponent)
    best = [null, 0]
    if (ply = 0 or no valid moves) then
        score = evaluate s for original player
        return [null, score]

    foreach valid move m for player in state s do
        execute move m on s
        [move, score] = minimax(s, ply-1, not MaxLevel)
        undo move m on s

        if (player is original) then
            if (score > best.score) then best = [m, score]
        else
            if (score < best.score) then best = [m, score]
    return best
```


Because the game tree is a recursive structure, the **Minimax** implementation also recursively identifies game states to explore. With each recursive call, the **ply** depth is decreased until **ply=0**, at which case the state **s** is evaluated from the perspective of the original player.

The **foreach loop** inside **minimax** evaluates the score for each of the children nodes from state **s** and remembers the highest score if that level is a **Max** level (that is, player at that level is the original player); alternatively, it remembers the lowest score if that level is a **Min** level (i.e., **not** a Maxlevel). Because the goal of **minimax** is to return the best move for the original player, it must return both the move and the ultimate best score that the player can hope for in the game tree when making that move.

Now let's go write code to match the Minimax pseudocode.

Minimax Implementation

The goal of Minimax is to identify a move for a player in a given game state. To represent a valid move, create this interface.

 In the **/src** source folder **chopsticks** package, create an **IMove** interface as shown:

CODE TO TYPE: IMove

```
package chopsticks;

public interface IMove {
    boolean valid(GameState state);
    boolean make(GameState state);
    boolean undo(GameState state);
}
```

Classes that claim to be a chopsticks move must be able to execute that move on a **GameState** object, changing its contents. All changes are made in place on a **GameState** object, so a move class must also be able to undo that move. Finally, the move class must be able to determine if it is even valid for a given **GameState**.

Define a class to represent the information returned by **minimax**.


 In the **/src** source folder **chopsticks** package, create a **Pair** class as shown:

CODE TO TYPE: Pair class

```
package chopsticks;

public class Pair {
    IMove move;
    int score;

    Pair (IMove move, int score) {
        this.move = move;
        this.score = score;
    }
}
```

 In the **/src** source folder **chopsticks** package, create a **Minimax** class as shown:

CODE TO TYPE: Minimax class

```
package chopsticks;

import java.util.*;

public class Minimax {
    int ply;
    int original;
    IEvaluate eval;

    public Minimax (int ply, IEvaluate ie) {
        this.ply = ply;
        this.eval = ie;
    }

    public IMove bestMove (GameState s, int player) {
        original = player;
        Pair bestMove = minimax (s, ply, true);
        return bestMove.move;
    }

    Pair minimax (GameState s, int ply, boolean maxLevel) {
        Collection<IMove> validMoves = null;
        if (ply > 0) { validMoves = computeMoves(s); }

        if (ply == 0 || validMoves.isEmpty()) {
            int score = eval.evaluate(s, original);
            return new Pair (null, score);
        }

        Pair best = null;
        for (IMove m : validMoves) {
            if (m.make(s)) {
                Pair next = minimax(s, ply-1, !maxLevel);
                next.move = m;
                m.undo(s);

                if (maxLevel) {
                    if (best == null || next.score > best.score) { best = next; }
                } else {
                    if (best == null || next.score < best.score) { best = next; }
                }
            }
        }

        return best;
    }
}
```

This code is missing the **computeMoves()** method (we'll get to that in a minute.) Let's take a closer look at this class:

OBSERVE: Constructing a Minimax instance

```
public class Minimax {
    int ply;
    int original;
    IEvaluate eval;

    public Minimax (int ply, IEvaluate ie) {
        this.ply = ply;
        this.eval = ie;
    }

    public IMove bestMove (GameState s, int player) {
        original = player;
        Pair bestMove = minimax (s, ply, true);
        return bestMove.move;
    }

    ...
}
```

Minimax **stores the IEvaluate implementation** used to evaluate game states, as well as the **ply** representing the maximum depth of the game tree to expand. To find the best move for a given state **s**, call the Minimax method **bestMove(s, p)**, which then **stores the original player to use** when evaluating game states. All of the interesting action happens in the **minimax** method:

OBSERVE: minimax recursive method

```
Pair minimax (GameState s, int ply, boolean maxLevel) {
    Collection<IMove> validMoves = null;
    if (ply > 0) { validMoves = computeMoves(s); }

    if (ply == 0 || validMoves.isEmpty()) {
        int score = eval.evaluate(s, original);
        return new Pair (null, score);
    }

    Pair best = null;
    for (IMove m : validMoves) {
        if (m.make(s)) {
            Pair next = minimax(s, ply-1, !maxLevel);
            next.move = m;
            m.undo(s);

            if (maxLevel) {
                if (best == null || next.score > best.score) { best = next; }
            } else {
                if (best == null || next.score < best.score) { best = next; }
            }
        }
    }


    return best;
}
```

Assuming that **ply** is greater than zero, minimax **iterates through all of the valid moves one by one**. After applying each move to the game state, minimax() **recursively invokes minimax at a depth of ply-1 and negates the maxLevel to alternate between Min and Max levels**. When the recursive call ends, **the move is undone**, minimax() records the maximum (or minimum) score of the children nodes of **s**, and it associates that move with the computed score. This method **returns the best computed move**.

When the recursive minimax method **reaches ply of 0**, it has reached the final depth (for completeness. There may be some game trees where a player has no more available moves earlier than that depth; that case is treated in the same way). minimax **evaluates the game state from the perspective of the**

original player and **returns the evaluated score within a Pair object** that currently has no move associated with it. The invoking method will associate the appropriate move object.

Now you just need to implement the **computeMoves(s)** method that returns a collection of move objects that represent the available moves at that state. First, you need to create a class that represents a **Tap** move.

 In the **/src** source folder **chopsticks** package, create a **TapMove** class as shown:

CODE TO TYPE: TapMove class

```
package chopsticks;

public class TapMove implements IMove {

    final int fromPoints;
    final int toPoints;
    int     newValue;

    public TapMove (int fromPoints, int toPoints) {
        this.fromPoints = fromPoints;
        this.toPoints = toPoints;
    }

    public String toString () {
        return "Tap " + toPoints + " with " + fromPoints;
    }

    public boolean valid(GameState s) {
        if (!s.values[s.player].contains(fromPoints)) { return false; }
        if (!s.values[1-s.player].contains(toPoints)) { return false; }
        return true;
    }

    public boolean make(GameState s) {
        if (!valid(s)) { return false; }

        newValue = fromPoints + toPoints;
        if (newValue >= 5) { newValue = 0; }
        if (s.values[1-s.player].size() == 2) {
            s.values[1-s.player].remove(toPoints);
        }
        s.values[1-s.player].add(newValue);

        s.player = 1-s.player;
        return true;
    }

    public boolean undo(GameState s) {
        s.player = 1-s.player;

        if (s.values[1-s.player].size() > 1) {
            s.values[1-s.player].remove(newValue);
        }
        s.values[1-s.player].add(toPoints);

        return true;
    }
}
```

Let's take a closer look.

TapMove class

```
public class TapMove implements IMove {

    final int fromPoints;
    final int toPoints;
    int      newValue;

    public TapMove (int fromPoints, int toPoints) {
        this.fromPoints = fromPoints;
        this.toPoints = toPoints;
    }

    public String toString () {
        return "Tap " + toPoints + " with " + fromPoints;
    }

    public boolean valid(GameState s) {
        if (!s.values[s.player].contains(fromPoints)) { return false; }
        if (!s.values[1-s.player].contains(toPoints)) { return false; }
        return true;
    }

    ...
}
```

A TapMove object represents the Tap move with a tapping hand that has **fromPoints** and with a tapped hand that contains **toPoints**. To determine whether a given move is valid in **GameState s**, the **valid** method only needs to determine **if the values associated with the current player (s.player) contain fromPoints**. Similarly, the move is only valid **if the hand of the opponent (1-s.player) contains toPoints**. The **newValue** value is computed within the **make** move to allow **undo** to work.

The real logic occurs within **make()**:

OBSERVE: TapMove make() method

```
public boolean make(GameState s) {
    if (!valid(s)) { return false; }

    newValue = fromPoints + toPoints;
    if (newValue >= 5) { newValue = 0; }
    if (s.values[1-s.player].size() == 2) {
        s.values[1-s.player].remove(toPoints);
    }
    s.values[1-s.player].add(newValue);

    s.player = 1-s.player;
    return true;
}
```

If the move is not valid in the state s, then it returns false; otherwise it determines the **newValue** to use for the opponent's hand. **If newValue is five or greater**, the player has a dead hand. The only tricky logic is to decide how to update the points for the opponent's hand. **If the opponent's hand already contained two distinct values (as determined by s.values[1-s.player].size())**, you must **remove the toPoints value** because it is being replaced with **newValue**. However, if the hand has two fingers with the same value (the set **s.values[1-s.player]** only has one value), you only have to **add newValue to the set**. Finally, once the move is made, **the player associated with the state flips to the other player**. To complete the **TapMove** class, there needs to be an **undo()** implementation.

OBSERVE: TapMove undo method

```
public boolean undo(GameState s) {  
    s.player = 1-s.player;  
  
    if (s.values[1-s.player].size() > 1) {  
        s.values[1-s.player].remove(newValue);  
    }  
    s.values[1-s.player].add(toPoints);  
  
    return true;  
}
```

The **undo** method is invoked only after a successful move. Its operations reverse the effect of the **make** method. It first **switches the active player of the state**, then **replaces the *newValue* value in the opponent's set with the original *toPoints* value**. If **the opponent has two distinct values**, ***newValue*** can be removed safely.

With **TapMove** available, you can now go back to the **Minimax** class and add the **computeMoves(GameState s)** method:


CODE TO TYPE: Adding computeMoves to Minimax

```
static Collection<IMove> computeMoves (GameState s) {  
    ArrayList<IMove> set = new ArrayList<IMove>();  
  
    for (int to : s.values[1-s.player]) {  
        if (to == 0) { continue; }  
        boolean alreadyOver = false;  
        for (int from : s.values[s.player]) {  
            if (from == 0) { continue; }  
            if (!alreadyOver) {  
                set.add(new TapMove (from, to));  
            }  
            if (from + to >= 5) {  
                alreadyOver = true;  
            }  
        }  
    }  
  
    return set;  
}
```

This method checks the four possible **Tap** moves by iterating over all the points in the player's hands, and trying to form **TapMove** objects with the points in each of the opponent's hands. Note that it must avoid dead hands with no points. This method also adds one more optimization that eliminates duplicate moves. For example, in the state **(Player1, 2, 3, 3, 4)**, Player1 has four **Tap** moves (2 on 3, 3 on 3, 2 on 4, 3 on 4). However, there are really only two potential states that can result from these moves: **(Player2, 2, 3, *, 3)** and **(Player2, 2, 3, *, 4)**. The **alreadyOver** variable is set to **true** to avoid computing redundant **TapMove** objects.

Now you're ready to put everything together! Write the code below to determine the best move for Player2 within the **(Player2, 1, 1, 1, 2)** state.

 In your **TwoPlayer** project, create a **/performance** source folder.

 In the **/performance** source folder, create a **chopsticks** package.

 In the **chopsticks** package, create a **PrintGameTree** class as shown:

COE TO TYPE: PrintGameTree class

```
package chopsticks;

public class PrintGameTree {
    public static void main(String[] args) {
        GameState gs = new GameState(GameState.Player2, 1, 1, 1, 2);
        IEvaluate eval = new Evaluator();
        int ply = 2;

        Minimax m = new Minimax(ply, eval);

        IMove move = m.bestMove(gs, GameState.Player2);
        System.out.println("best move: " + move);
    }
}
```



Save and run it. The output is **best move: Tap 1 with 1**. This means that the best move is going to be to the left of the game tree presented earlier. So, how can you make sure that the code is working correctly? Make these code changes to output information as the algorithm executes:

CODE TO TYPE: Changes to Minimax to expose information as it processes

```
package chopsticks;

import java.util.*;

public class Minimax {
    int ply;
    int original;
    IEvaluate eval;
    StringBuffer padding;

    public Minimax (int ply, IEvaluate ie) {
        this.ply = ply;
        this.eval = ie;
    }

    public IMove bestMove (GameState s, int player) {
        padding = new StringBuffer();
        original = player;
        Pair bestMove = minimax (s, ply, true);
        return bestMove.move;
    }

    Collection<IMove> computeMoves (GameState s) {
        ArrayList<IMove> set = new ArrayList<IMove>();

        for (int to : s.values[1-s.player]) {
            boolean alreadyOver = false;
            for (int from : s.values[s.player]) {
                if (!alreadyOver) {
                    set.add(new TapMove (from, to));
                }
                if (from + to >= 5) {
                    alreadyOver = true;
                }
            }
        }

        return set;
    }

    Pair minimax (GameState s, int ply, boolean maxLevel) {
        System.out.print(padding.toString() + s + " ");
        Collection<IMove> validMoves = null;
        if (ply > 0) { validMoves = computeMoves(s); }

        if (ply == 0 || validMoves.isEmpty()) {
            int score = eval.evaluate(s, original);
            System.out.println(" [" + score + "]");
            return new Pair (null, score);
        }

        System.out.println();
        Pair best = null;
        for (IMove m : validMoves) {
            if (m.make(s)) {
                padding.append(" ");
                Pair next = minimax(s, ply-1, !maxLevel);
                next.move = m;
                padding.setLength(padding.length()-2);
                m.undo(s);


                if (maxLevel) {
                    if (best == null || next.score > best.score) { best = next; }
                } else {
                    if (best == null || next.score < best.score) { best = next; }
                }
            }
        }
    }
}
```

```

    }
}

System.out.println(padding.toString() + " returning best: " + best.move + "
, " + best.score);
return best;
}
}

```

 Now when you run it, you see this:

OBSERVE: Trace of the Minimax algorithm

```

(Player2,1,1,1,2)
  (Player1,1,2,1,2)
    (Player2,1,2,2,2) [5]
    (Player2,1,2,2,3) [10]
    (Player2,1,2,1,3) [15]
    (Player2,1,2,1,4) [30]
    returning best: Tap 1 with 1, 5
  (Player1,1,3,1,2)
    (Player2,1,3,2,2) [30]
    (Player2,1,3,2,4) [50]
    (Player2,1,3,1,3) [20]
    (Player2,1,3,0,1) [-85]
    returning best: Tap 2 with 3, -85
  returning best: Tap 1 with 1, 5
best move: Tap 1 with 1

```

This output reflects the evaluation values presented in the earlier game tree. The indentation reflects the depth in the game tree, and the evaluation of each node (from the perspective of Player2) appears in brackets at the end of each row.

Let's see if this algorithm can find the winning move described earlier for Player2 in the state **(Player2, 1, 3, 2, 4)**. Modify **PrintGameTree** as to expand only one level:

CODE TO TYPE: Modified PrintGameTree

```


package chopsticks;

public class PrintGameTree {
    public static void main(String[] args) {
        GameState gs = new GameState(GameState.Player2, 1, 3, 2, 41112);
        IEvaluate eval = new Evaluator();
        int ply = 12;

        Minimax m = new Minimax(ply, eval);

        IMove move = m.bestMove(gs, GameState.Player2);
        System.out.println("best move:" + move);
    }
}

```

 Run it. It determines that the best move is to tap the opponent's hand with 3 fingers to force a dead hand for Player1:

OBSERVE: Computed 1-ply Minimax search on (Player2, 1, 3, 2, 4)

```

(Player2,1,3,2,4)
  (Player1,3,3,2,4) [-55]
  (Player1,0,3,2,4) [45]
  (Player1,0,1,2,4) [55]
  returning best: Tap 3 with 2, 55
best move: Tap 3 with 2

```

Minimax works best when it explores sufficient levels of the tree. Change the **ply** to 2 and observe that Minimax now finds a better move:

OBSERVE: Computed 2-ply Minimax search on (Player2, 1, 3, 2, 4)

```
(Player2,1,3,2,4)
  (Player1,3,3,2,4)
    (Player2,3,3,0,4) [-85]
    (Player2,3,3,0,2) [-75]
    returning best: Tap 2 with 3, -85
  (Player1,0,3,2,4)
    (Player2,0,3,0,4) [15]
    (Player2,0,3,0,2) [25]
    returning best: Tap 2 with 3, 15
  (Player1,0,1,2,4)
    (Player2,0,1,3,4) [90]
    (Player2,0,1,0,2) [-5]
    returning best: Tap 4 with 1, -5
  returning best: Tap 1 with 4, 15
best move: Tap 1 with 4
```

In fact, you can increase **ply** to larger values, but the extra searches are redundant now that a victory has been found. There is a more efficient path-finding algorithm called *Alpha/Beta* that can reduce the size of game trees dramatically by intelligently pruning redundant searches. You can read about this algorithm in the [Algorithms In A Nutshell](#) companion book.

Lessons Learned

Much of the success of **Minimax** is derived from its ability to model both the game state and the available moves in the game efficiently. For chopsticks, there were three potential ways to represent the game state. Design the game state with care because each potential move class must implement three methods—valid, make, and undo. If you select an overly complicated modeling structure, you will waste precious time debugging the move classes. You must choose a design that offers the greatest benefits to the most move classes.

Do not skip the step where the **IEvaluate** interface was defined. The success of Minimax ultimately depends on having accurate and relevant evaluation classes to estimate the "strength" of a board from a player's point of view. Crafting these heuristics is almost an art form and you'll want to experiment with a number of potential evaluation classes.

When implementing an algorithm, be sure to keep the logic of the core algorithm fully encapsulated within its own set of classes. Use interfaces to identify the user-specified classes cleanly for the actual game problem being solved. That way, you can use the Minimax code as an engine for multiple two-player games.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

Algorithms On Sound Data

Lesson Objectives

When you finish this lesson, you will be able to:

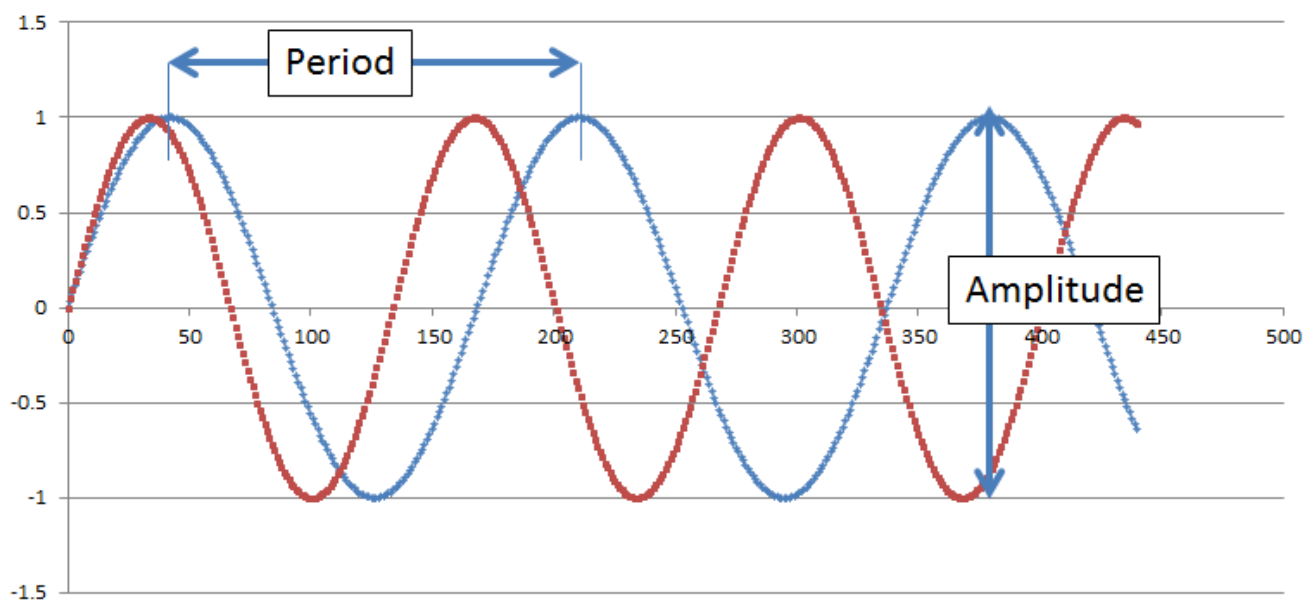
- demonstrate how to invoke Fast Fourier Transform on a third-party library.
- generate sound wave forms to play using Java's AudioFormat class.
- convert a frequency into its note equivalent on a piano keyboard.

Signal Processing Algorithms

The algorithms in this course focus mostly on human-readable real-world data, such as string values, integers, floating-point numbers, and Cartesian points. Wouldn't it be great to be able to process sound data, for example, to detect the pitch of a note—or even a chord of notes—being played? In this lesson, you'll learn how to create and process sound data using the *Fast Fourier Transform* (FFT), which is often considered one of the most important numerical algorithms of the 20th century. You will learn how to process Waveform Audio File Format (WAV) data containing uncompressed audio encoded using a linear pulse code modulation (LPCM) format.

Sound is a traveling longitudinal wave which is an oscillation of pressure. An individual wave is defined by its period (the distance in time between two high points) and amplitude (the total distance vertically from the highest point to the lowest point). The amplitude represents the energy of the wave or its "loudness." For this lesson, we will assume that all wave forms are normalized between $[-1, 1]$ because the focus is on frequency analysis.

The human ear interprets a sound wave by converting it into a musical pitch (or note). Each musical note corresponds to a specific frequency which is measured in *hertz*, or the number of complete cycles per second of a periodic phenomenon (in this case, the sound wave). Studies have demonstrated that the range of hearing for an infant child is 20 Hertz to 20,000 Hertz. The middle C on a piano keyboard is tuned to the frequency of 261.626 Hertz, which is well within this range (for additional frequency values, see the [Wikipedia entry on piano frequencies](#)). If you were to sample this sound wave 44,100 times per second, you would compute 44,100 individual values—the first 450 are shown below in the blue time series. The horizontal axis (t-axis) represents time, while the vertical y-axis represents the energy contained in the wave at time t .



To interpret the above blue sound wave, you need to know the sampling rate and the time when the blue sound wave completes a full period. The blue wave period is about 169 time units. Since there are 44,100 total samples, the computed frequency of the blue wave is $44100/169$ or 260.95—close to the middle C frequency we mentioned earlier.

The red sound wave above represents the tone when playing the E key just above middle C. Based on a period of 135 time units, its frequency is computed to be 326.66—close to its actual value of 329.628.

Pulse Code Modulation (PCM) demonstrates how to represent the continuous properties of the wave form discretely. PCM represents an audio waveform as a sequence of amplitude values recorded at a sequence of times. LPCM is

PCM with linear quantization. The standard audio file format for CDs, for example, is LPCM-encoded with two channels of 44,100 samples per second. Each sample is recorded as an unsigned 16-bit integer value.

To begin our investigation into sound data, we'll write a small program that generates an 8-bit quality sound wave form that plays a middle C note.



Create a new Java Project named **SoundFiles** and assign it to the **Java6_Lessons** working set.



In your **SoundFiles** project **/src** source folder, create an **fft** package.



In the **fft** package, create a **Generate** class as shown:

CODE TO TYPE: Generate class

```
package fft;

import javax.sound.sampled.*;

public class Generate {
    public static void main(String[] args) throws Exception {
        float sampleRate = 44100;
        double f = 261.626;
        double a = 0.5;
        double twoPiF = 2*Math.PI*f;

        double[] buffer = new double [44100];
        for (int sample = 0; sample < buffer.length; sample++) {
            double time = sample / sampleRate;
            buffer[sample] = a * Math.sin(twoPiF*time);
        }

        final byte[] byteBuffer = new byte[buffer.length];
        int idx = 0;
        for (int i = 0; i < byteBuffer.length; ) {
            int x = (int) (buffer[idx++] * 127);
            byteBuffer[i++] = (byte) x;
        }

        boolean bigEndian = false;
        boolean signed = true;
        int bits = 8;
        int channels = 1;
        AudioFormat format = new AudioFormat(sampleRate, bits, channels, signed, bigEndian);

        DataLine.Info info = new DataLine.Info(SourceDataLine.class, format);
        SourceDataLine line = (SourceDataLine) AudioSystem.getLine(info);
        line.open(format);
        line.start();
        long now = System.currentTimeMillis();
        line.write(byteBuffer, 0, byteBuffer.length);
        line.close();
        long total = System.currentTimeMillis() - now;
        System.out.println(total + " ms.");
    }
}
```



Save and run it. You hear a tone that sounds like the middle C note on the piano. To create this sound, this class generates a one-second wave form using 8-bit sound encoding. Let's take a closer look:

OBSERVE: Creating Wave Form

```
float sampleRate = 44100;
double f = 261.626;
double a = 0.5;
double twoPiF = 2*Math.PI*f;

double[] buffer = new double[44100];
for (int sample = 0; sample < buffer.length; sample++) {
    double time = sample / sampleRate;
    buffer[sample] = a * Math.sin(twoPiF*time);
}
```

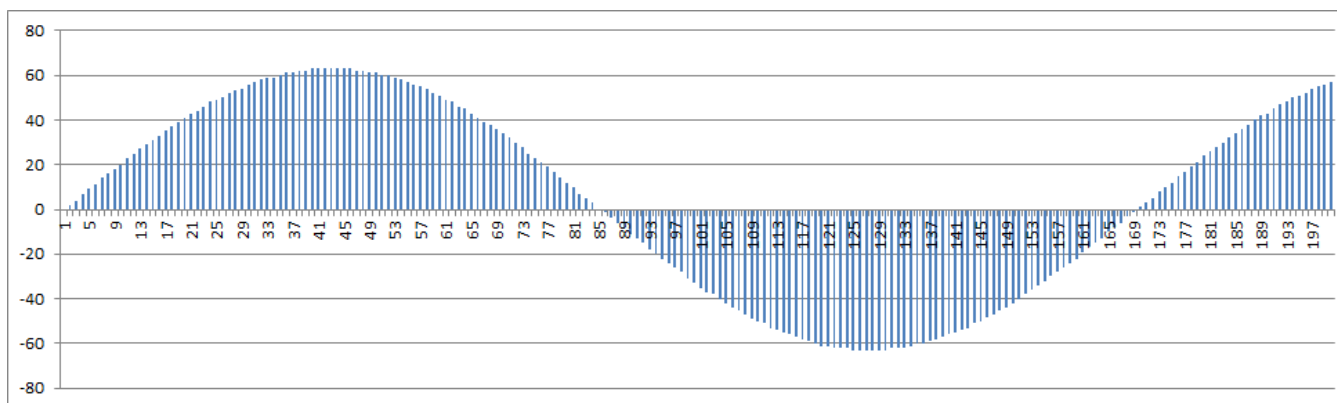
Sound data can be represented using a time series computed using the trigonometric Sine function. In a time series, the value t represents the time unit, which increments from 0 to increasing positive numbers. The variable f specifies the desired frequency (in this case, Middle C). The variable a represents a scaling factor (between 0 and 1) to apply to the amplitude of the wave. The wave will be at its "loudest" when a is 1.0, and softer with decreasing values of a . The variable $twoPiF$ pre-computes the constant value used within the **for** loop for optimization. **buffer** contains the sequence of 44,100 floating point values representing the wave form. With each pass through the **for** loop, $time$ represents the t -coordinate for which the wave form is computed using the formula $a * \sin(2\pi f t)$.

Once **buffer** is computed, it must be converted into its corresponding byte encoding. For 8-bit sound quality, there are 256 different values that can be generated. Naturally, 16-bit sound quality is able to more accurately model a sound wave form because it allows for a total of 65,536 different values:

OBSERVE: Create byte buffer from floating-point buffer

```
final byte[] byteBuffer = new byte[buffer.length];
int idx = 0;
for (int i = 0; i < byteBuffer.length; ) {
    int x = (int) (buffer[idx++] * 127);
    byteBuffer[i++] = (byte) x;
}
```

The floating point values are "scaled" to become signed byte values (-128 to 127). The code actually only computes 255 possible values (from -127 to 127) because of the conversion from **int** to **byte**, but that's acceptable when digitizing a Sine wave. The image below charts the sample byte values of this buffer. The values only range from -63 to +63 because the amplitude of the generated wave form, a , is scaled at 0.5:



The size of **byteBuffer** is the same as the original buffer. The JDK plays the 8-bit encoded byte buffer:

OBSERVE: Playing bytes as sound

```
boolean bigEndian = false;
boolean signed = true;
int bits = 8;
int channels = 1;
AudioFormat format = new AudioFormat(sampleRate,bits,channels,signed,bigEndian);


DataLine.Info info = new DataLine.Info(SourceDataLine.class, format);
SourceDataLine line = (SourceDataLine) AudioSystem.getLine(info);
line.open(format);
line.start();
long now = System.currentTimeMillis();
int written = line.write(byteBuffer, 0, byteBuffer.length);
line.close();
System.out.println(written + " bytes written.");
long total = System.currentTimeMillis() - now;
System.out.println(total + " ms.");
```

The **AudioFormat** object represents the encoding used so the underlying audio software can interpret the bytes it receives properly. As you can imagine, there are numerous encoding styles and hardware devices available to process these encodings. Here the code specifies that:

- the bytes are encoded in little-Endian order, from least significant bit to greatest. This only matters for 16-bit and higher encodings.
- the bytes are signed (where negative numbers are "below" the x-axis).
- there are 8 bits in each encoding, thus the audio hardware will read 8 bits at a time.
- there is just a single channel of output.

An **AudioFormat** object could have multiple streams of input, called *channels*. A stereo audio source, for example, would have two channels (left and right). A mono source would have only a single channel.

Once the format is declared, the code creates a **SourceDataLine** object to manage the transfer of bytes. First, the **line is opened with the declared AudioFormat object**. Then the **start method is invoked** to declare that it must be ready to receive the data, which is **written as a single block write**. Finally, the **line is closed and final statistics are output**.

 Run this code again; you might be surprised to hear that the sound seems to play for much less than one second. The output shows that the program ran in about 1/2 second:


OBSERVE: Generate output

```
44100 bytes written.
565 ms.
```

What might be going wrong? Well, normally all sound information to be played is buffered prior to being delivered to the hardware. To determine the size of the buffer, modify **Generate** as shown:

CODE TO TYPE: Detect size of sound buffer

```
...
long now = System.currentTimeMillis();
System.out.println("buffer size:" + line.available());
int written = line.write(byteBuffer, 0, byteBuffer.length);
...
```

 Run it again; you see that the buffer size is 22,050, which supports about 1/2 second of audio. The program completes before the audio hardware finishes playing the sound. The simplest way to fix this is to make sure that all bytes sent to the audio hardware are drained before it can be closed. This means calling **drain()** blocks until all data has been played.

CODE TO TYPE: Properly drain buffer to play entire sound

```
...  
    int written = line.write(byteBuffer, 0, byteBuffer.length);  
    line.drain();  
    line.close();  
...
```



Now in addition to playing the sound for a full second, the output shows something like this:

OBSERVE: Proper execution of Generate

```
buffer size:22050  
44100 bytes written.  
1035 ms.
```

You can have lots of fun with sound wave forms. The next change generates a stereo signal of middle C being played, but the amplitude changes in the left and right sides to provide the illusion of sound depth over a two-second period:

CODE TO TYPE: Modify Generate to generate stereo output

```
package fft;

import javax.sound.sampled.*;

public class Generate {
    public static void main(String[] args) throws Exception {
        float sampleRate = 44100;
        double f = 261.626;
        double a = .5;
        double twoPiF = 2*Math.PI*f;

        double[] buffer = new double[44100*4];
        for (int sample = 0; sample < buffer.length; sample++) {
            double time = (sample/2) / sampleRate;
            double a1 = a*Math.sin(Math.PI*time);
            double a2 = a*Math.cos(Math.PI*time);
            buffer[sample++] = a1 * Math.sin(twoPiF*time);    // channel 1
            buffer[sample] = a2 * Math.sin(twoPiF*time);    // channel 2
        }

        byte[] byteBuffer = new byte[buffer.length];
        int idx = 0;
        for (int i = 0; i < byteBuffer.length; ) {
            int x = (int) (buffer[idx++]*127);
            byteBuffer[i++] = (byte) x;
        }

        boolean bigEndian = false;
        boolean signed = true;
        int bits = 8;
        int channels = 2;
        AudioFormat format = new AudioFormat(sampleRate,bits,channels,signed,bigEndian);

        DataLine.Info info = new DataLine.Info(SourceDataLine.class, format);
        SourceDataLine line = (SourceDataLine) AudioSystem.getLine(info);
        line.open(format);
        line.start();
        long now = System.currentTimeMillis();
        System.out.println("buffer size:" + line.available());
        int written = line.write(byteBuffer, 0, byteBuffer.length);
        System.out.println(written + " bytes written.");
        line.drain();
        line.close();
        long total = System.currentTimeMillis() - now;
        System.out.println(total + " ms.");
    }
}
```

Experiment with the code some more. For example, change the frequency, **f**, to determine the lowest or highest pitch that you can hear.

To be able to process actual sound files containing recorded music, you need to work with 16-bit sound data. Modify **Generate** as shown to recreate a 16-bit mono encoding of just middle C:

CODE TO TYPE: Modifications to Generate class

```
package fft;

import javax.sound.sampled.*;

public class Generate {
    public static void main(String[] args) throws Exception {
        float sampleRate = 44100;
        double f = 261.626;
        double a = .5;
        double twoPiF = 2*Math.PI*f;

        double[] buffer = new double [44100*42];
        for (int sample = 0; sample < buffer.length; sample++) {
            double time = sample/2 / sampleRate;
            double a1 = a*Math.sin(Math.PI*time);
            double a2 = a*Math.cos(Math.PI*time);
            buffer[sample++] = a1 * Math.sin(twoPiF*time); // channel 1
            buffer[sample] = a2 * Math.sin(twoPiF*time); // channel 2
        }

        byte[] byteBuffer = new byte[buffer.length*2];
        int idx = 0;
        for (int i = 0; i < byteBuffer.length; ) {
            int x = (int) (buffer[idx++]*25532767);
            byteBuffer[i++] = (byte) x;
            byteBuffer[i++] = (byte) (x >>> 8);
        }

        boolean bigEndian = false;
        boolean signed = true;
        int bits = 816;
        int channels = 21;
        AudioFormat format = new AudioFormat(sampleRate,bits,channels,signed,bigEndian);

        DataLine.Info info = new DataLine.Info(SourceDataLine.class, format);
        SourceDataLine line = (SourceDataLine) AudioSystem.getLine(info);
        line.open(format);
        line.start();
        long now = System.currentTimeMillis();
        System.out.println("buffer size:" + line.available());
        int written = line.write(byteBuffer, 0, byteBuffer.length);
        System.out.println(written + " bytes written.");
        line.drain();
        line.close();
        long total = System.currentTimeMillis() - now;
        System.out.println(total + " ms.");
    }
}
```



You probably won't detect any audible difference, since it still plays a middle C tone for just about two seconds. We've only made minor changes to the earlier 8-bit mono version. Let's take a closer look at the way byteBuffer is constructed:

OBSERVE: Create byte buffer with byte pairs

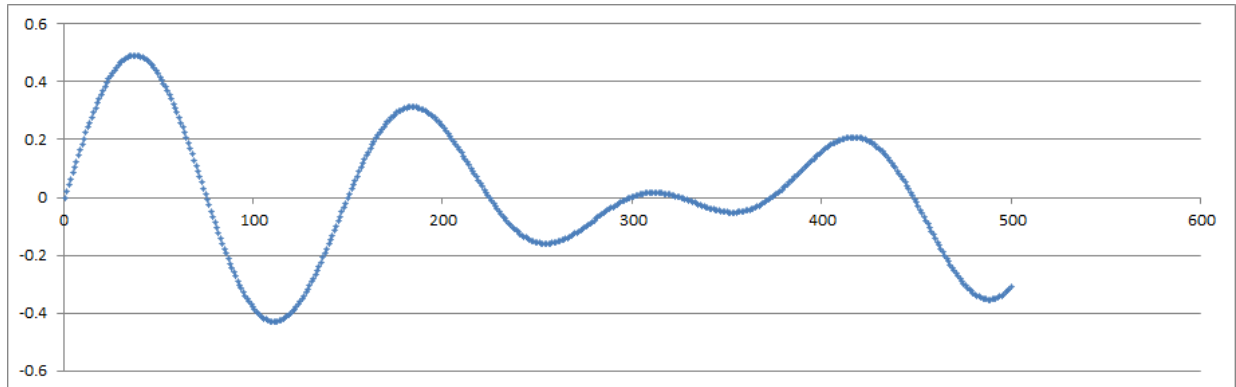
```
byte[] byteBuffer = new byte[buffer.length*2];
int idx = 0;
for (int i = 0; i < byteBuffer.length; ) {
    int x = (int) (buffer[idx++]*32767);
    byteBuffer[i++] = (byte) x;
    byteBuffer[i++] = (byte) (x >>> 8);
}
```

To store 16-bit data, you need a byte buffer **twice as large as the 8-bit solution**. In addition, the byte values are no

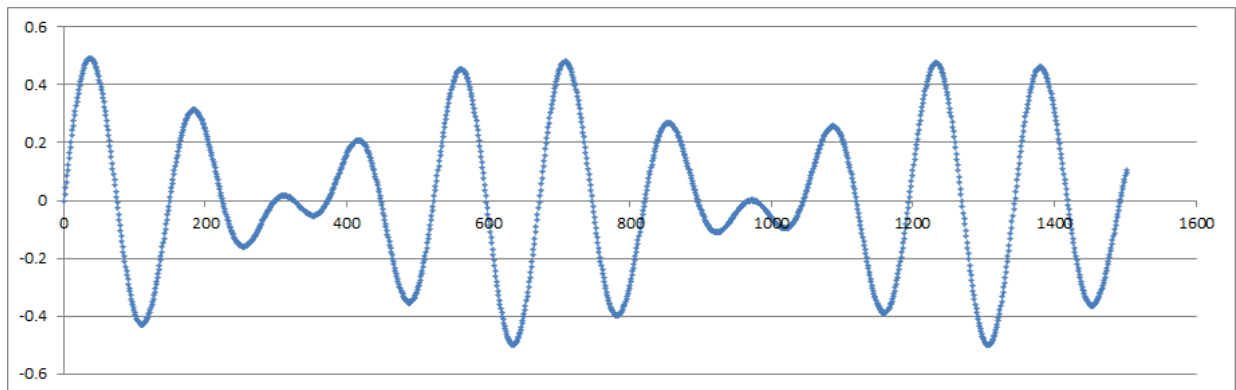
longer simply drawn from the range of 256 values (or 2^8). Instead you need to represent 65,536 (or 2^{16}) different values. To do this, the value is encoded into two neighboring byte values. This code will generate only 65,535 possible encodings, but that's acceptable. Using little-Endian encoding, the lower 8 bits of the encoded value, **x**, is **written out first**, then the **value of x is shifted 8 bits to the right** before it is written out. Finally, the number of bits in the AudioFormat object is changed from 8 to 16 and the number of channels is set to 1. To represent the audio in big-Endian encoding, you would simply swap the order of these two bytes in byteBuffer.

Composed Wave Forms

Let's return to the original plot at the start of this lesson, which contained a blue wave form representing middle C and a red wave representing the E above middle C. Instead of depicting these separately, the image below represents the *combined sound wave* of these two notes playing simultaneously. The wave data is normalized so its values all remain within the [-1, 1] range:



Where before the sound waves showed clear signs of periodicity, this wave form seems unintelligible. However, if you plot more samples, the periodic structure becomes visible:



Modify the **Generate** class as shown:

CODE TO TYPE: Modifications to Generate

```
package fft;

import javax.sound.sampled.*;

public class Generate {
    public static void main(String[] args) throws Exception {
        float sampleRate = 44100;
        double f1 = 261.626;
        double f2 = 329.628;
        double a = .5;
        double twoPiF1 = 2*Math.PI*f1;
        double twoPiF2 = 2*Math.PI*f2;

        double[] buffer = new double [44100*2];
        for (int sample = 0; sample < buffer.length; sample++) {
            double time = sample / sampleRate;
            buffer[sample] = a * (Math.sin(twoPiF1*time) + Math.sin(twoPiF2*time))/2;
        }

        byte[] byteBuffer = new byte[buffer.length*2];
        int idx = 0;
        for (int i = 0; i < byteBuffer.length; ) {
            int x = (int) (buffer[idx++]*32767);
            byteBuffer[i++] = (byte) x;
            byteBuffer[i++] = (byte) (x >>> 8);
        }

        boolean bigEndian = false;
        boolean signed = true;
        int bits = 16;
        int channels = 1;
        AudioFormat format = new AudioFormat(sampleRate,bits,channels,signed,bigEndi
an);

        DataLine.Info info = new DataLine.Info(SourceDataLine.class, format);
        SourceDataLine line = (SourceDataLine) AudioSystem.getLine(info);
        line.open(format);
        line.start();
        long now = System.currentTimeMillis();
        System.out.println("buffer size:" + line.available());
        int written = line.write(byteBuffer, 0, byteBuffer.length);
        System.out.println(written + " bytes written.");
        line.drain();
        line.close();
        long total = System.currentTimeMillis() - now;
        System.out.println(total + " ms.");
    }
}
```



Save and run it. You hear two notes playing simultaneously (Middle C at a frequency of 261.626 and the E note just above it at a frequency of 329.638). The only real difference with earlier code is the construction of the floating-point buffer. Let's review this code:

OBSERVE: Create composed Sound Wave

```
float sampleRate = 44100;
double f1 = 261.626;
double f2 = 329.628;
double a = 0.5;
double twoPiF1 = 2*Math.PI*f1;
double twoPiF2 = 2*Math.PI*f2;

double[] buffer = new double [44100*2];
for (int sample = 0; sample < buffer.length; sample++) {
    double time = sample / sampleRate;
    buffer[sample] = a * (Math.sin(twoPiF1*time) + Math.sin(twoPiF2*time)) / 2;
}
```

The values in **buffer** are the composition of two sounds being played. The values in **buffer** must be in the range [-1, 1]. When adding two Sine values together, **the code divides by 2** to ensure that the resulting value remains within this range.

Analyzing Composed Wave Forms

To analyze a composed wave form, you want to identify the individual sound wave frequencies that represent the dominant components of the composition. In mathematics, there is a *Discrete Fourier Transform* (DFT) that can convert a sampled function into a finite combination of complex sinusoidal functions ordered by their frequencies that has the same sample values. There are three important concepts presented in this one sentence:

- **Convert a Sampled Function:** You may know that you can determine a line uniquely by just two points. That is, given just two pairs of (x,y) values on a line, you can determine its equation. By analogy, the sampled sound frequencies are being treated like individual points, this time with a *t-coordinate* representing the time of that sample and a *y-coordinate* representing the amplitude of the wave at that time unit. Based solely on this information, you're trying to determine a function $f(t)$ that satisfies all of these points.
- **A finite combination of sinusoidal functions:** In the composed wave form example, the resulting chord is computed by adding together two sinusoidal functions. In general, you cannot know in advance how many sinusoidal functions are present in any complex wave form, but you can assume that you are looking for only a finite number.
- **Complex sinusoidal functions:** The DFT is defined over the set of *complex numbers*, which are numbers that can be expressed in the form $a + bi$, where a and b are real numbers and i is the *imaginary unit*, defined as $i^2 = -1$. Every real number is already a complex number (with the imaginary part of $b=0$). Second, we are concerned with the magnitude of the complex numbers being processed. For a complex number of the form $a + bi$, its magnitude is the square root of $(a^2 + b^2)$.

Here is the single formula that "explains" the DFT:

$$X(k) = \sum_{t=0}^{n-1} x(t) * e^{\left(\frac{-2\pi i t k}{n}\right)}$$

Ok, that's a bit much. For this lesson you don't need to understand how this formula was derived, but I'll show you how to implement it in Java. Complex numbers use the special variable i to represent the *imaginary unit*, because you can see i in the above formula, you know that it relies on computations over complex numbers. You can reduce the right half of the above formula by converting the exponential value of e like this:


$$e^{\frac{-2\pi i t k}{n}} = \cos\left(\frac{2\pi t k}{n}\right) - i * \sin\left(\frac{2\pi t k}{n}\right)$$

You accumulate $X(k)$ by $x(t)$ times the above, making sure to deal with the complex values that result from the computation properly. You are given n , which is the number of sampled values, $x(t)$. You only need to determine the range of frequencies for k .

The term $x(t)$ represents the sample value for time unit t ; there are n sample values in all. You want to determine $X(k)$, which represents the signal level for frequency k . Now, for which values of k are you going to compute X ? In any input sample, the highest frequency is $1/2$ the total number of samples (based on the concept of Nyquist Frequency). However, this still leaves you with $n*n/2$ computations, or $O(n^2)$ computations. You can likely execute DFT on only a small number of samples before it becomes too costly to execute.

Once the values of $X(k)$ are computed, you can investigate them to find those maximal values which directly correlate to the existence of a wave form in the input with frequency k .

Let's write some code to compute DFT. You can reuse the chord generation code above. Generally, DFT is meant to process complex values as input; however, your input consists of real valued samples, so the code is a bit simpler than the generic DFT.

 In the **fft** package, create a **DFT** class as shown:

CODE TO TYPE: DFT class

```
package fft;

public class DFT {

    static void dft(double[] inR, double[] outR, double[] outI) {
        for (int k = 0; k < inR.length; k++) {
            for (int t = 0; t < inR.length; t++) {
                outR[k] += inR[t]*Math.cos(2*Math.PI * t * k / inR.length);
                outI[k] -= inR[t]*Math.sin(2*Math.PI * t * k / inR.length);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        float sampleRate = 44100;
        double f1 = 261.626;
        double f2 = 329.628;
        double a = .5;
        double twoPiF1 = 2*Math.PI*f1;
        double twoPiF2 = 2*Math.PI*f2;

        double[] bufferR = new double [2048];
        for (int sample = 0; sample < bufferR.length; sample++) {
            double time = sample / sampleRate;
            bufferR[sample] = a * (Math.sin(twoPiF1*time) + Math.sin(twoPiF2*time))/2;
        }

        double[] outR = new double[bufferR.length];
        double[] outI = new double[bufferR.length];

        dft(bufferR, outR, outI);

        double results[] = new double[outR.length];
        for (int i = 0; i < outR.length; i++) {
            results[i] = Math.sqrt(outR[i]*outR[i] + outI[i]*outI[i]);
        }

        java.io.PrintStream ps = new java.io.PrintStream("Sample.txt");
        for (double d : results) {
            ps.println(d);
        }
        ps.close();
    }
}
```

Let's look at this code more closely. The first half of the **main** method is identical to earlier code that constructs a composed wave form from playing two notes (C and E):

OBSERVE: Invoking DFT on the wave form

```
double[] outR = new double[bufferR.length];
double[] outI = new double[bufferR.length];

dft(bufferR, outR, outI);

double results[] = new double[outR.length];
for (int i = 0; i < outR.length; i++) {
    results[i] = Math.sqrt(outR[i]*outR[i] + outI[i]*outI[i]);
}
```

The `dft` method computes **two buffers, outR and outI**, which contain the n computed complex values of $X(k)$. These are **composed into a single results array** by determining the *magnitude* of the complex number. The magnitude of the complex number $a + bi$ is computed as the square root of $a^2 + b^2$.

The actual DFT computation is performed in the `dft` method, which is simplified because the input contains only real numbers, not complex numbers:

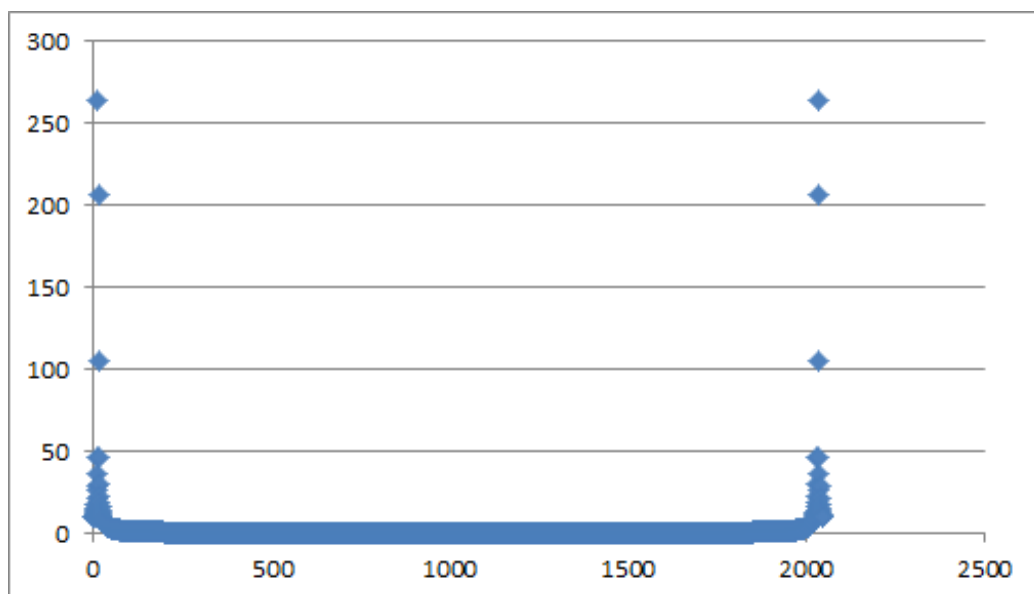
OBSERVE: DFT implementation

```
static void dft(double[] inR, double[] outR, double[] outI) {
    for (int k = 0; k < inR.length; k++) {
        for (int t = 0; t < inR.length; t++) {
            outR[k] += inR[t]*Math.cos(2*Math.PI * t * k / inR.length);
            outI[k] -= inR[t]*Math.sin(2*Math.PI * t * k / inR.length);
        }
    }
}
```

Given n samples in `inR`, this loop performs $n*n$ operations, ultimately accumulating the proper complex number result in `outR` and `outI`. When this method completes, these two arrays contain the complex values of $X(k)$.

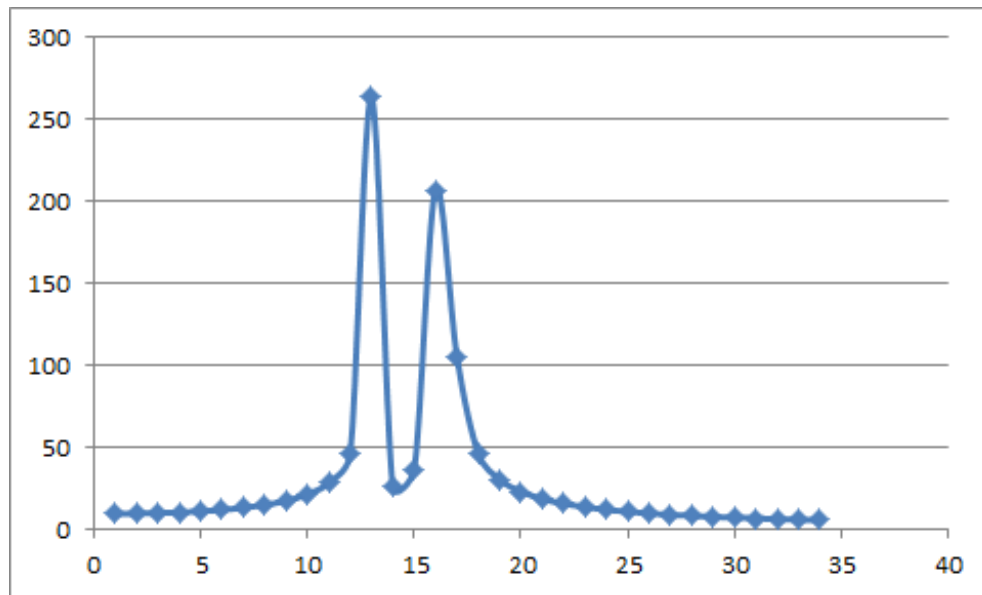
Save and run DFT; this will create a file "Sample.txt" in the current Eclipse project. To see this file, select the enclosing project and right-click to select **Refresh**.

Because there are 44,100 sample values in just one second of sound data, the DFT is inefficient for our purposes. Here, only 2048 samples are used to enable the computation to complete in under a second. However, how do you know that the result is accurate? Retrieve the values from "Sample.txt" and plot them using a program such as Excel.



This graph is symmetric. The x-axis represents a *frequency index*, which divides the 44,100 (the sample rate) possible frequencies into 2,048 (the number of samples being used for DFT) discrete ones. This graph further demonstrates that you only need to consider the first half of these frequencies. Let's focus on the first

35 values:



We are concerned with only the magnitude of these values and the two highest points that occur are the 13th and 16th points. These indices are based on counting from zero, so these are actually frequency indices of 12 and 15. You can convert these frequency indices into actual frequencies like this:

- $12 \cdot 44100 / 2048 = 258.3984375$
- $15 \cdot 44100 / 2048 = 322.998046875$

These two values are really close to the frequencies of the C and E notes in the composed wave form. Think about what this code has accomplished! Given a buffer containing a composed wave form, the DFT was somehow able to isolate the two dominant frequencies. Now let's add some quick and dirty processing code to identify these maximum peaks within the results array of a DFT, which will allow you to detect these frequencies in the composed wave form. For more complex wave forms, you will need a more nuanced approach, but this gives you an idea of what's possible. Modify **DFT** as shown:

CODE TO TYPE: Modifications to DFT

```
package fft;

import java.util.*;

public class DFT {

    static void dft(double[] inR, double[] outR, double[] outI) {
        for (int k = 0; k < inR.length; k++) {
            for (int t = 0; t < inR.length; t++) {
                outR[k] += inR[t]*Math.cos(2*Math.PI * t * k / inR.length);
                outI[k] -= inR[t]*Math.sin(2*Math.PI * t * k / inR.length);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        float sampleRate = 44100;
        double f1 = 261.626;
        double f2 = 329.628;
        double a = .5;
        double twoPiF1 = 2*Math.PI*f1;
        double twoPiF2 = 2*Math.PI*f2;

        double[] bufferR = new double [2048];
        for (int sample = 0; sample < bufferR.length; sample++) {
            double time = sample / sampleRate;
            bufferR[sample] = a * (Math.sin(twoPiF1*time) + Math.sin(twoPiF2*time))/2;
        }

        double[] outR = new double[bufferR.length];
        double[] outI = new double[bufferR.length];

        dft(bufferR, outR, outI);

        double results[] = new double[outR.length];
        for (int i = 0; i < outR.length; i++) {
            results[i] = Math.sqrt(outR[i]*outR[i] + outI[i]*outI[i]);
        }

        java.io.PrintStream ps = new java.io.PrintStream("Sample.txt");
        for (double d : results) {
            ps.println(d);
        }
        ps.close();

        List<Float> found = process(results, sampleRate, bufferR.length, 4);
        for (float freq : found) {
            System.out.println("Found: " + freq);
        }
    }

    static List<Float> process(double results[], float sampleRate, int numSamples,
int sigma) {
        double average = 0;
        for (int i = 0; i < results.length; i++) {
            average += results[i];
        }
        average = average/results.length;

        double sums = 0;
        for (int i = 0; i < results.length; i++) {
            sums += (results[i]-average)*(results[i]-average);
        }

        double stdev = Math.sqrt(sums/(results.length-1));
    }
}
```

```

    ArrayList<Float> found = new ArrayList<Float>();
    double max = Integer.MIN_VALUE;
    int maxF = -1;
    for (int f = 0; f < results.length/2; f++) {
        if (results[f] > average+sigma*stdev) {
            if (results[f] > max) {
                max = results[f];
                maxF = f;
            }
        } else {
            if (maxF != -1) {
                found.add(maxF*sampleRate/numSamples);
                max = Integer.MIN_VALUE;
                maxF = -1;
            }
        }
    }

    return (found);
}

```

The **process** method computes the average and standard deviation of the **results** array and it eliminates from consideration any frequency index f with a *results[f]* that is smaller than $average + 4*stdev$ which should eliminate 99.73% of the frequency indices from consideration. If a magnitude for a particular index is higher than this threshold, it warrants further consideration. Now sweeping f from 0 to $n/2$ where n is the number of computed values in **results** the **for** loop seeks to find a local maximum, max , and its corresponding frequency index value, $maxF$. Then it computes the detected frequency by multiplying the frequency index, $maxF$, by the *sampleRate* and dividing by *numSamples*.



Save and run it; these frequencies are detected in the output:

OBSERVE: Execute DFT
Found: 258.39844
Found: 322.99805

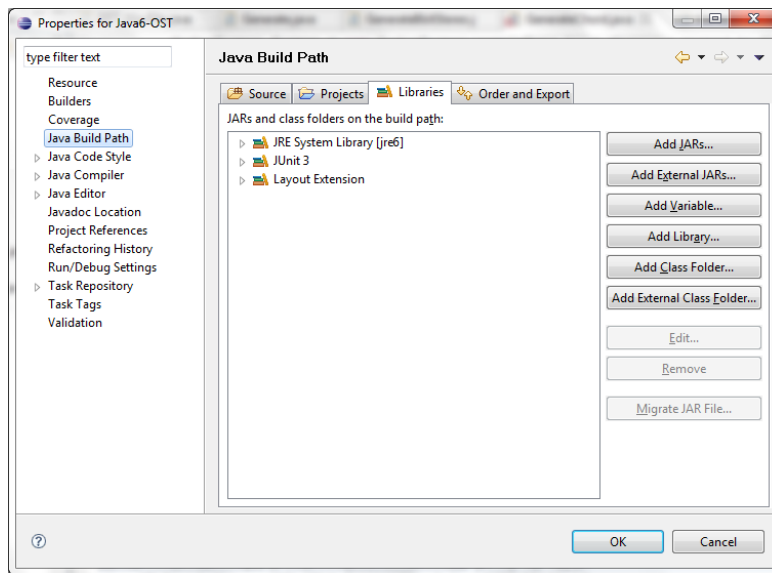
To validate that this code functions as expected, change the **f1** and **f2** values to two different frequencies in the range 27 to 4,186, which represent the full range of the 88 keys on the keyboard. The computations won't identify the frequency precisely because the accuracy of the computation is limited to $44100/2048$ or 21.5 hertz. The only way to increase accuracy is to increase the number of samples processed by DFT. However, doing so will dramatically slow the computation because of the $O(n^2)$ behavior. If you rerun the above code using different samples, 4096 and 8192 respectively, you get these results in roughly the identified time:

- 4096: (3 seconds) 258.39844, 333.76465
- 8192: (10 seconds) 263.78174, 328.38135

There is a more efficient version known as the *Fast Fourier Transform* (FFT). In this lesson, you will learn how to use FFT, rather than implement it, because of the numerical complexity of the algorithm. There are a number of freely available implementations.

To get this library, [right-click this link](#) and save the file in your workspace.

Then, add the **commons-math3-3.2.jar** library to be part of the build path in Eclipse. Right-click on your project icon within the workspace and select the **Properties** entry; on the left side, select **Java Build Path**. You see this dialog:



Click the **Add JARs...** button on the right and use the provided window to browse in your project to the **libs** folder where the **commons-math3-3.2.jar** file exists. Select it and click **OK**. Now you are ready to start using the FFT code which is part of this JAR file.

The best way to learn FFT is to use it. Modify the **DFT** class to execute FFT on the buffer of double values that it creates. The only requirement that FFT has is that the input size is a perfect power of 2. Otherwise, it produces exactly the same result format as DFT:

CODE TO TYPE: Modifications to DFT

```
package fft;

import java.util.*;

import org.apache.commons.math3.complex.Complex;
import org.apache.commons.math3.transform.*;

public class DFT {

    static void dft(double[] inR, double[] outR, double[] outI) {
        for (int k = 0; k < inR.length; k++) {
            for (int t = 0; t < inR.length; t++) {
                outR[k] += inR[t]*Math.cos(2*Math.PI * t * k / inR.length);
                outI[k] -= inR[t]*Math.sin(2*Math.PI * t * k / inR.length);
            }
        }
    }

    public static void main(String[] args) throws Exception {
        float sampleRate = 44100;
        double f1 = 261.626;
        double f2 = 329.628;
        double a = .5;
        double twoPiF1 = 2*Math.PI*f1;
        double twoPiF2 = 2*Math.PI*f2;

        double[] bufferR = new double [2048];
        for (int sample = 0; sample < bufferR.length; sample++) {
            double time = sample / sampleRate;
            bufferR[sample] = a * (Math.sin(twoPiF1*time) + Math.sin(twoPiF2*time))/2;
        }

        double[] outR = new double[bufferR.length];
        double[] outI = new double[bufferR.length];

dft(bufferR, outR, outI);
        FastFourierTransformer fft = new FastFourierTransformer(DftNormalization.STANDARD);
        Complex resultC[] = fft.transform(bufferR, TransformType.FORWARD);

        double results[] = new double[outR.length];
for (int i = 0; i < outR.length; i++) {
    results[i] = Math.sqrt(outR[i]*outR[i] + outI[i]*outI[i]);
}
        for (int i = 0; i < resultC.length; i++) {
            double real = resultC[i].getReal();
            double imaginary = resultC[i].getImaginary();
            results[i] = Math.sqrt(real*real + imaginary*imaginary);
        }

        List<Float> found = process(results, sampleRate, bufferR.length, 4);
        for (float freq : found) {
            System.out.println("Found: " + freq);
        }
    }

    static List<Float> process(double results[], float sampleRate, int numSamples,
int sigma) {
        double average = 0;
        for (int i = 0; i < results.length; i++) {
            average += results[i];
        }
        average = average/results.length;

        double sums = 0;
        for (int i = 0; i < results.length; i++) {
```



```


        sums += (results[i]-average)*(results[i]-average);
    }

    double stdev = Math.sqrt(sums/(results.length-1));

    ArrayList<Float> found = new ArrayList<Float>();
    double max = Integer.MIN_VALUE;
    int maxF = -1;
    for (int f = 0; f < results.length/2; f++) {
        if (results[f] > average+sigma*stdev) {
            if (results[f] > max) {
                max = results[f];
                maxF = f;
            }
        } else {
            if (maxF != -1) {
                found.add(maxF*sampleRate/numSamples);
                max = Integer.MIN_VALUE;
                maxF = -1;
            }
        }
    }

    return (found);
}
}

```

 Save and run it; you get the same result as before, but much more quickly. FFT makes it practical to accurately process composed wave forms. The next step, naturally, is to execute FFT on actual recorded audio samples. Let's get started.

Using FFT on WAV file samples

Your project should come with some existing WAV resources for you to use. Once you finish this lesson, make your own sound recordings to see if you can replicate the processing done here. Working with real sound files introduces a number of additional issues. Let's see how to load up a WAV sound file containing 16-bit encoded data that needs to be converted into **double** values; this sequence is essentially the reverse of the sound generation you did at the beginning of this lesson.

There are five sound files to be processed:

- CFA_MajorChord.wav: 7 seconds of a C-F-A chord played on a regular piano
- CMajorChord.wav: 7 seconds of a C-E-G major chord played on a regular piano
- ClavinovaCMajorChord.wav: 7 seconds of a C-E-G major chord played on an electric Yamaha Clavinova
- CSeventhChord.wav: 7 seconds of a C-E-G-Bb chord (Major C-7th) played on a regular piano
- CrystalGlass.wav: 4 seconds of the ringing of a crystal glass

Let's process the clearest signal—the crystal glass.

 In the **fft** package, create a **WAVProcessing** class as shown:

CODE TO TYPE: WAVProcessing

```
package fft;

import java.io.*;
import java.util.*;
import javax.sound.sampled.*;
import org.apache.commons.math3.complex.Complex;
import org.apache.commons.math3.transform.*;

public class WAVProcessing {
    public static void main(String[] args) throws Exception {
        File fileIn = new File("chords\\CrystalGlass.wav");
        AudioInputStream audioInputStream = AudioSystem.getAudioInputStream(fileIn);
        System.out.println(audioInputStream.getFormat());
        int size = audioInputStream.available();
        byte[] bytesIn = new byte[size];
        audioInputStream.read(bytesIn);

        AudioFormat format = audioInputStream.getFormat();
        float rate = format.getFrameRate();

        int numChannels = format.getChannels();
        double[] buffer = new double [1048576];
        int idx = 0;
        for (int i = 0; i < bytesIn.length && idx < buffer.length; i += 2) {
            byte blow = bytesIn[i];
            byte bhigh = bytesIn[i+1];

            buffer[idx++] = (blow & 0xFF | bhigh << 8)/32767;
            if (numChannels == 2) { i += 2; }
        }

        FastFourierTransformer fft = new FastFourierTransformer(DftNormalization.STANDARD);
        Complex resultC[] = fft.transform(buffer, TransformType.FORWARD);

        double[] results = new double[resultC.length];
        for (int i = 0; i < resultC.length; i++) {
            double real = resultC[i].getReal();
            double imaginary = resultC[i].getImaginary();
            results[i] = Math.sqrt(real*real + imaginary*imaginary);
        }

        List<Float> found = DFT.process(results, rate, resultC.length, 7);
        HashMap<String,Float> keys = new HashMap<String,Float>();
        System.out.println("Found:" + found);
        for (float freq : found) {
            keys.put(closestKey(freq), freq);
        }
        for (String note : keys.keySet()) {
            System.out.println("Found: " + note + " @ freq=" + keys.get(note));
        }
    }

    static String[] notes = {"A", "A#", "B", "C", "C#", "D", "D#", "E", "F", "F#", "G", "G#"};

    public static String closestKey(double freq) {
        int key = closestKeyIndex(freq);
        if (key <= 0) { return null; }
        int range = 1+(key-1)/notes.length;
        return notes[(key-1)%notes.length] + range;
    }

    public static int closestKeyIndex(double freq) {
        return 1+(int) ((12*Math.log(freq/440)/Math.log(2) + 49) - 0.5);
    }
}
```

```
}
```



Save and run it:

OBSERVE: CrystalGlass Analysis

```
PCM_SIGNED 44100.0 Hz, 16 bit, stereo, 4 bytes/frame, little-endian
Found: [1645.6919, 1646.7013, 1647.5845, 1651.4116, 1664.2811, 3217.153]
Found: G#6 @ freq=1664.2811
Found: G7 @ freq=3217.153
```

The code may have trouble distinguishing frequencies at the higher octaves on the piano. Ideally the sound of crystal glass would have only one harmonic subtone one octave above the base tone of **G#6**. You can see that the identified frequencies are nearly double each other.

Most of this code is familiar to you by now. Let's review the new additions:

OBSERVE: Converting Frequency into a piano note

```
static String[] notes = {"A", "A#", "B", "C", "C#", "D", "D#", "E", "F", "F#",
"G", "G#"};

public static String closestKey(double freq) {
    int key = closestKeyIndex(freq);
    if (key <= 0) { return null; }
    int range = 1+(key-1)/notes.length;
    return notes[(key-1)%notes.length] + range;
}

public static int closestKeyIndex(double freq) {
    return 1+(int) ((12*Math.log(freq/440)/Math.log(2) + 49) - 0.5);
}
```

A number of frequencies were detected by FFT. The code we write next consolidates these frequencies into distinct pitches using a **HashMap** to associate the detected frequency with the closest key as computed above.

The **notes** static field records the 12 distinct notes as found on a piano. Each tone occurs at a given *octave* number. The lowest note on the 88-key piano keyboard is key number 1 (A0); the highest note is key number 88 (C8). The **closestKeyIndex** method takes a frequency and returns the corresponding key number on the piano in the range from 1-88. This formula is derived from the logarithmic nature of the frequencies. The **closestKey** function converts this number into a human-readable string representing the note on the piano that most closely corresponds to the given frequency:

OBSERVE: consolidate frequencies into pitches

```
List<Float> found = DFT.process(results, rate, resultC.length, 7);
HashMap<String,Float> keys = new HashMap<String,Float>();
System.out.println("Found:" + found);
for (float freq : found) {
    keys.put(closestKey(freq), freq);
}
```

Frequencies that are "close together" become consolidated in the HashMap, so only two detected notes appear in the output.

The notes on an ideal piano range from a low frequency of 27.5 to a high of 4186.01. Instead of being evenly spaced, the notes are arranged in octaves that are multiples of each other. For example, middle C is the frequency 261.626, while the C one octave higher is 523.251. The **closestKeyIndex** method computes the piano key index with 1 being the lowest key on the piano and 88 being the highest key. If the frequency is lower than the lowest key on the piano, this method returns a number smaller than 1; that's why the **closestKey** method protects against this situation.

Lessons Learned

- **Sound wave data has a structure that you can manipulate:** Sound data is encoded in bytes to represent the wave forms.
- **Real-world sound data is not perfect:** The sound data you generate has a near-perfect representation as sinusoidal wave forms. Recorded sounds rarely have this structure, so the FFT results are indicative of existing frequencies, rather than clear and definitive.

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

Conclusion

Lesson Objectives

When you finish this lesson, you will be able to:

- implement a data structure that conforms to the **Set** interface.
 - write code to remove an element from an unbalanced binary search tree.
 - explain why kd-trees are unable to easily support element removal.
 - explain how to rebalance a self-balancing binary tree after deleting values.
-

Concluding Lesson For Algorithms

In this final lesson, we'll go over removing elements from a collection. Throughout the course, you have learned how to use a variety of data structures to represent information. In all cases, the presentation focused on how to construct entire representations from the beginning. However, it is also important to be able to describe how to remove an element from a collection that you have spent so much time constructing. We'll conduct this topic in these contexts:

- Arrays
- Binary Search Trees
- AVL Binary Search Trees
- kd-trees

This lesson provides a *capstone experience* by pulling together most of the important data structures you've seen and explaining new functionality that you might expect to see.

Removing Elements From a Sorted Array

Given a sorted array of elements, you can use an Binary Array Search to locate an element in the array in $O(\log n)$ time. To remove an element, however, you have two choices:

- Allocate a new array to contain all elements from the original set minus the one being removed.
- Shift elements down within the existing array and maintain an additional value, **number**, that records the number of elements in the array (note that $\text{number} < \text{length}$).

Past lessons have demonstrated both of these options, and neither leads to an efficient implementation. Specifically, inserting an element into—or removing an element from—an array-based structure requires $O(n)$ in the worst case because you have to copy $n-1$ elements within the same array. Neither of these choices lets you amortize the costs across multiple remove or add requests.

You might consider a third choice, using an **ArrayList** object to store the sorted elements, but then you become responsible for inserting new values at their proper locations. This can be less difficult to implement than either of the first two choices. When you use **ArrayList**, make sure that **SortedSet** conforms to the **Set** interface of the Java Collections Framework.



Create a new Java project named **Conclusion** and assign it to the **Java6_Lessons** working set.

Copy the packages and programs from your **BinaryTree** project into your **Conclusion** project.



In your **Conclusion** project **/src** source folder, **binary** package, create a **SortedSet** class as shown:

CODE TO TYPE: SortedSet class

```
package binary;

import java.util.*;

public class SortedSet<E extends Comparable<E>> implements Set<E> {
    ArrayList<E> list = new ArrayList<E>();

    int binarySearch(E e) {
        int low = 0;
        int high = list.size()-1;
        while (low <= high) {
            int mid = (low + high)/2;
            int rc = e.compareTo(list.get(mid));
            if (rc < 0) {
                high = mid - 1;
            } else if (rc > 0) {
                low = mid + 1;
            } else {
                return mid;
            }
        }
        return -(low + 1);
    }

    public boolean add(E e) {
        int idx = binarySearch(e);
        if (idx >= 0) { return false; }

        list.add(-(idx+1), e);
        return true;
    }

    public boolean remove(Object o) {
        int idx = binarySearch((E)o);
        if (idx < 0) { return false; }

        list.remove(idx);
        return true;
    }
}
```

The **SortedSet** class uses an **ArrayList** object to store a set of elements in sorted order; the set contains no duplicates.

The code won't compile just yet because there are still some methods that you have to write to satisfy the **Set** interface. Let's take a closer look at the initial functionality:

OBSERVE: binarySearch on a sorted ArrayList

```
int binarySearch(E e) {
    int low = 0;
    int high = list.size()-1;
    while (low <= high) {
        int mid = (low + high)/2;
        int rc = e.compareTo(list.get(mid));
        if (rc < 0) {
            high = mid - 1;
        } else if (rc > 0) {
            low = mid + 1;
        } else {
            return mid;
        }
    }
    return -(low + 1);
}
```

The `binarySearch` method assumes the underlying *list* **ArrayList** stores its items in order. The code is similar to the *binarySearch* implemented in an earlier lesson; the only difference is that it must access each element in *list* using the `get()` method. **binarySearch** returns a non-negative value (that is, greater than or equal to zero), when it finds the desired element *e* in the **ArrayList**. When **binarySearch** returns a negative number *x*, element *e* should be inserted at $-(x+1)$. For example, when $x=-1$ is returned, element *e* is to be inserted at position 0. You can see this behavior in the code for **add**:

OBSERVE: Methods to add element to and remove element from sorted ArrayList

```
public boolean add(E e) {
    int idx = binarySearch(e);
    if (idx >= 0) { return false; }

    list.add(-(idx+1), e);
    return true;
}

public boolean remove(Object o) {
    int idx = binarySearch((E)o);
    if (idx < 0) { return false; }

    list.remove(idx);
    return true;
}
```

In the contract defined by the Java Collections Framework, the **add** method in the **Set** interface **must return true** whenever its contents have changed. The **remove** method similarly **returns true** only when its contents have changed. To conform to the **Set** contract, the **remove** method takes a generic **Object** as its parameter.

The **Set** interface defines a **contains** method that you can add to the end of the **SortedSet** class now:

CODE TO TYPE: Add method to the end of SortedSet

```
...
public boolean contains(Object o) {
    return (binarySearch((E)o) >= 0);
}
}
```

The Collections Framework defines a number of *bulk operations* to perform on a set. Add these methods to the **SortedSet** class.

CODE TO TYPE: Add bulk operation methods to end of class

```
...
    public boolean addAll(Collection<? extends E> c) {
        boolean changed = false;
        for (E e : c) {
            changed |= add(e);
        }
        return changed;
    }

    public boolean removeAll(Collection<?> c) {
        boolean changed = false;
        for (E e : (Collection<E>)c) {
            changed |= remove(e);
        }
        return changed;
    }

    public boolean containsAll(Collection<?> c) {
        for (E e : (Collection<E>)c) {
            if (binarySearch(e) < 0) { return false; }
        }
        return true;
    }

    public boolean retainAll(Collection<?> c) {
        boolean changed = false;
        for (int idx = list.size() - 1; idx >= 0; idx--) {
            if (!c.contains(list.get(idx))) {
                list.remove(idx);
                changed = true;
            }
        }
        return changed;
    }
}
```

The **addAll(c)** and **removeAll(c)** methods iterate over elements in the **Collection** parameter *c* and add or remove that element from the **ArrayList** storage.

The **containsAll(c)** method iterates over every element, *e*, in *c* to determine if the **SortedSet** contains *e*, returning **false** immediately when a non-member element, *e*, is detected. If all elements in *c* belong to the **SortedSet**, it returns **true**.

The **retainAll(c)** method demands a more complicated implementation. Specifically, this method removes all elements in **SortedSet** that do not exist within *c*; it does so by iterating through its elements in reverse order, removing each element that does not exist in *c*. If the **retainAll(c)** method changes the set in any way, it must return **true**, based on the contract for the **Set** interface.

To complete the implementation of the necessary methods required by **Set**, add the following methods to the end of the **SortedSet** class:

CODE TO TYPE: Complete SortedSet implementation

```
...
    public int size() { return list.size(); }
    public Object[] toArray() { return list.toArray(); }
    public <T> T[] toArray(T[] a) { return list.toArray(a); }
    public void clear() { list.clear(); }
    public boolean isEmpty() { return list.isEmpty(); }
    public Iterator<E> iterator() { return list.iterator(); }
}
```

In each case, the required method delegates each request to the internal *list* **ArrayList** object.

Even though the **SortedSet** class now compiles, you still have to implement some methods to conform to

the Java Collections Framework. Specifically, for **SortedSet** to truly satisfy the **Set** interface, its **hashCode** method must be implemented to return the sum of the **hashCode** of the values it contains. Add the following method to the end of **SortedSet**:

CODE TO TYPE: Add hashCode method to SortedSet

```
...
    public int hashCode() {
        int hash = 0;
        for (int i = 0; i < list.size(); i++) {
            hash += list.get(i).hashCode();
        }
        return hash;
    }
}
```

The final change is to ensure that the **equals(o)** method returns **true** if and only if *o* is a set, the two sets have the same size, and every member of *o* is contained in this set. Add this method to the end of the **SortedSet** class:

CODE TO TYPE: Add equals method to SortedSet

```
...
    public boolean equals(Object o) {
        if (o == null) { return false; }
        if (!(o instanceof Set)) { return false; }
        Set<E> s = (Set<E>) o;
        if (s.size() != list.size()) { return false; }
        for (E e : s) {
            if (binarySearch(e) < 0) { return false; }
        }
        return true;
    }
}
```

Once the **equals** method determines that it is comparing against another **Set** object, *s*, it iterates through each element, *e* in *s*, to determine whether the **SortedSet** contains *e*, returning **false** at the first difference. Once all elements are determined to be contained within the **SortedSet**, it can safely return **true**.

Congratulations! You have completed your first **Set** implementation! Write the **StressTest** class to demonstrate its functionality and compare its performance with the **TreeSet** implementation.

 In your **Conclusion** project **/src** source folder, **binary** package, create a **StressTest** class as shown:

CODE TO TYPE: StressTest class

```
package binary;

import java.util.*;
public class StressTest {
    final static double AddProb      = 0.20;
    final static double ContainsProb = 0.70;
    final static int     SetSize      = 5000;
    final static int     TrialSize     = 50000;
    final static String[] Types      = {"Add", "Contains", "Remove" };

    static void fail(String err) {
        System.err.println("Failed on:" + err);
        System.exit(-1);
    }

    public static void main(String[] args) {
        TreeSet<Integer> base = new TreeSet<Integer>();
        SortedSet<Integer> set = new SortedSet<Integer>();
        double baseCount[] = new double[3];
        double setCount[] = new double[3];
        int counts[] = new int[3];
        long start;
        boolean b,s;
        for (int t = 0; t < TrialSize; t++) {
            int n = (int) (Math.random()*SetSize);
            double choice = Math.random();
            if (choice < AddProb) {
                start = System.nanoTime();
                b = base.add(n);
                baseCount[0] += (System.nanoTime() - start);
                start = System.nanoTime();
                s = set.add(n);
                setCount[0] += (System.nanoTime() - start);
                if (b != s) { fail(Types[0]); }
                counts[0]++;
            } else if (choice < ContainsProb) {
                start = System.nanoTime();
                b = base.contains(n);
                baseCount[1] += (System.nanoTime() - start);
                start = System.nanoTime();
                s = set.contains(n);
                setCount[1] += (System.nanoTime() - start);
                if (b != s) { fail(Types[1]); }
                counts[1]++;
            } else {
                start = System.nanoTime();
                b = base.remove(n);
                baseCount[2] += (System.nanoTime() - start);
                start = System.nanoTime();
                s = set.remove(n);
                setCount[2] += (System.nanoTime() - start);
                if (b != s) { fail(Types[2]); }
                counts[2]++;
            }
        }
        for (int i = 0; i < counts.length; i++) {
            if (counts[i] != 0) {
                baseCount[i] /= counts[i];
                setCount[i] /= counts[i];
            }
            System.out.println(Types[i] + " base=" + (int)baseCount[i]+ " set=" + (int)
setCount[i]);
        }
    }
}
```

Run it to compare the behavior of **SortedSet** against **TreeSet** when 500,000 random operations are performed on each collection where 20% of the time a random integer (up to 5000) is added, 50% of the time a contains query is executed, and 30% of the time a random integer (up to 5000) is requested to be deleted. Fine-grained timing statistics are recorded for each operation on the two data structures. The sample output below shows that, on average, **SortedSet** is always slower than **TreeSet** (almost three times slower for *add* and *remove*). Your performance results will likely vary from those shown:

OBSERVE: Output from StressTest execution

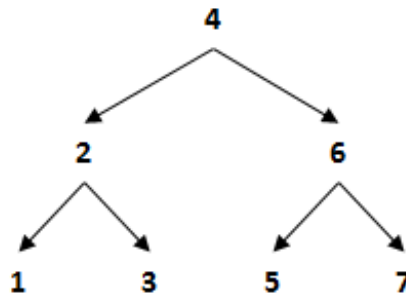
```
Add base=255 set=856
Contains base=215 set=283
Remove base=245 set=659
```

Ultimately the **TreeSet** class will outperform **SortedSet** because of the extra cost of growing the array that contains the elements of the **SortedSet**; still this was a worthwhile exercise.

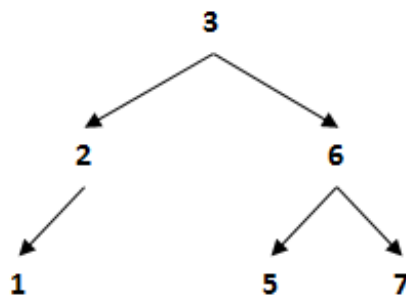
Now let's look at how to handle the removal of elements from highly structured data structures.

Removing Elements From Binary Search Trees

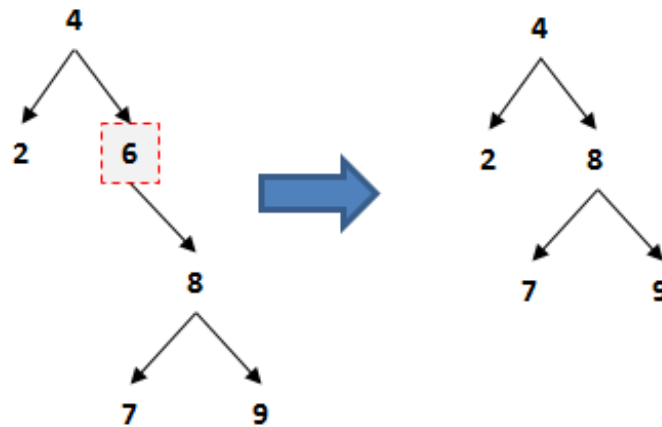
Binary Search Trees (BSTs) offer the first data structure for which removing an element should be an $O(\log n)$ operation; after all, it takes $O(\log n)$ performance to determine whether the BST contains the element in the first place. You need to identify a deterministic way to *reconstruct* the BST after removing an element. If you are removing a leaf node, the BST already is properly formed; however, what if you remove an element that has one or more children? Consider this small BST with seven elements from which you would like to remove the element 4:



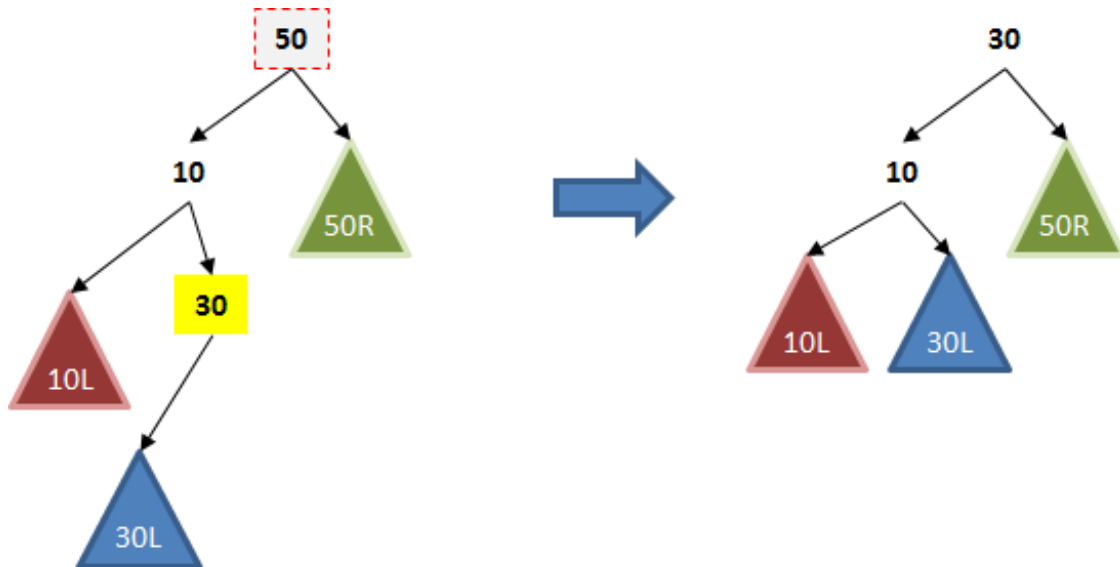
As you can see, this value is currently the root node of the tree. What can be done? You could always recreate a new BST by starting from scratch and inserting all $n-1$ elements, but that would be inefficient; in fact, that single operation would require on the order of $O(n \log n)$ operations. Instead, compare the following six-element BST with the earlier seven-element BST.



You can observe that the right sub-trees are the same in both BSTs and that the second tree continues to support the *Binary Search Tree Property*. The second tree was not formed by deleting the node 4. Rather, the value associated with this node was replaced with the *largest element in its left sub-tree*, in this case, 3. There is one small caveat; if the node you want to delete has no left sub-tree, it can be replaced by the entire right sub-tree of the node to be deleted, as shown in the image below, which describes the result of removing the value 6 from a sample BST:



Given a node n with value that is to be removed from the tree, the largest element in its left sub-tree is exactly the *right-most descendant of the left child of n* . As you can see, this value is smaller than all of the values in the right sub-tree of the node being removed (because the tree is a BST). This value is also larger than all of the other nodes in the BST rooted by the left child of the node being removed. Once you find X , the right-most descendant of the left child of the node being removed, you can swap its value with the node being removed. Now, what if the node for X has any child nodes? Then it cannot have any right children, otherwise it would not be the right-most descendant of the left child of the node being removed. However, X might have a left child; indeed, it might have an entire sub-tree rooted by that left child. Fortunately, as with the rotations described earlier in the AVL lesson, you can "lift" up X 's left sub-tree to replace X in the BST, and the BST properties will once again hold. The image below shows the transformation of the BST when requested to remove the value **50** from the BST. $X=30$, is the largest element in the left sub-tree of **50**. The inner node **10** has only its right sub-tree changed to be the entire sub-tree **30L**, as shown:



Make these changes to the **BinaryTree** class in the **binary** package:

CODE TO TYPE: Modifications to BinaryTree

```
package binary;

public class BinaryTree<E extends Comparable<E>> {

    BinaryNode<E> root = null;

    public int size() {
        if (root == null) { return 0; }

        return root.size();
    }

    public int height () {
        if (root == null) { return 0; }

        return height(root);
    }

    int height (BinaryNode<E> n) {
        if (n == null) { return 0; }

        return 1 + Math.max( height(n.left), height(n.right));
    }

    public void add (E k) {
        if (root == null) {
            root = new BinaryNode<E>(k);
            return;
        }

        root = root.add(root, k);
    }

    public boolean contains (E k) {
        return contains(root, k);
    }

    boolean contains (BinaryNode<E> parent, E k) {
        if (parent == null) { return false; }

        int rc = k.compareTo(parent.key);
        if (rc == 0) {
            return true;
        } else if (rc < 0) {
            return contains(parent.left, k);
        } else {
            return contains(parent.right, k);
        }
    }

    public void remove (E k) {
        if (root == null) { return; }

        root = remove(root, k);
    }

    BinaryNode<E> remove (BinaryNode<E> parent, E k) {
        if (parent == null) { return null; }
        int rc = k.compareTo(parent.key);
        if (rc == 0) {
            return parent.updateNodes();
        } else if (rc < 0) {
            parent.left = remove(parent.left, k);
        } else {
            parent.right = remove(parent.right, k);
        }
    }
}
```

```
        return parent;
    }
}
```

The structure of the new **remove** method is similar to the **contains** method—to delete an element, you must first determine whether it exists in the BST. Once **k.compareTo(parent.key)** returns **0**, you have found the node that contains the value to be removed. At this point, you need to write an additional method in **BinaryNode** that properly updates the node.

Modify **BinaryNode** as shown:

CODE TO TYPE: Modifications to BinaryNode

```
package binary;

public class BinaryNode<E extends Comparable<E>> {
    final E key;
    BinaryNode<E> left;
    BinaryNode<E> right;

    public BinaryNode(E k) {
        this.key = k;
    }

    public int size() {
        return 1 + size(left) + size(right);
    }

    int size(BinaryNode<E> n) {
        if (n == null) { return 0; }
        return n.size();
    }

    void add (E k) {
        int rc = k.compareTo(key);
        if (rc <= 0) {
            left = add(left, k);
        } else {
            right = add(right, k);
        }
    }

    BinaryNode<E> add(BinaryNode<E> parent, E k) {
        if (parent == null) {
            return new BinaryNode<E>(k);
        }

        parent.add(k);
        return parent;
    }

    public BinaryNode<E> updateNodes() {
        if (left == null && right == null) { return null; }
        if (left == null) { return right; }
        if (right == null) { return left; }

        BinaryNode<E> child = left;
        BinaryNode<E> grandChild = child.right;
        if (grandChild == null) {
            left = child.left;
            key = child.key;
        } else {
            while (grandChild.right != null) {
                child = grandChild;
                grandChild = grandChild.right;
            }
            key = grandChild.key;
            child.right = grandChild.left;
        }

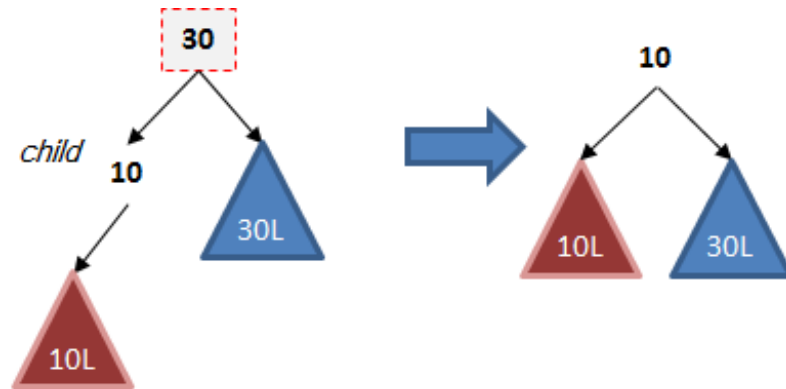
        return this;
    }
}
```

The **updateNodes** method is called on a node that has been targeted for deletion. The value returned must be the (possibly new) node that will take the place of this node in the BST and which may have its own left and right sub-trees.

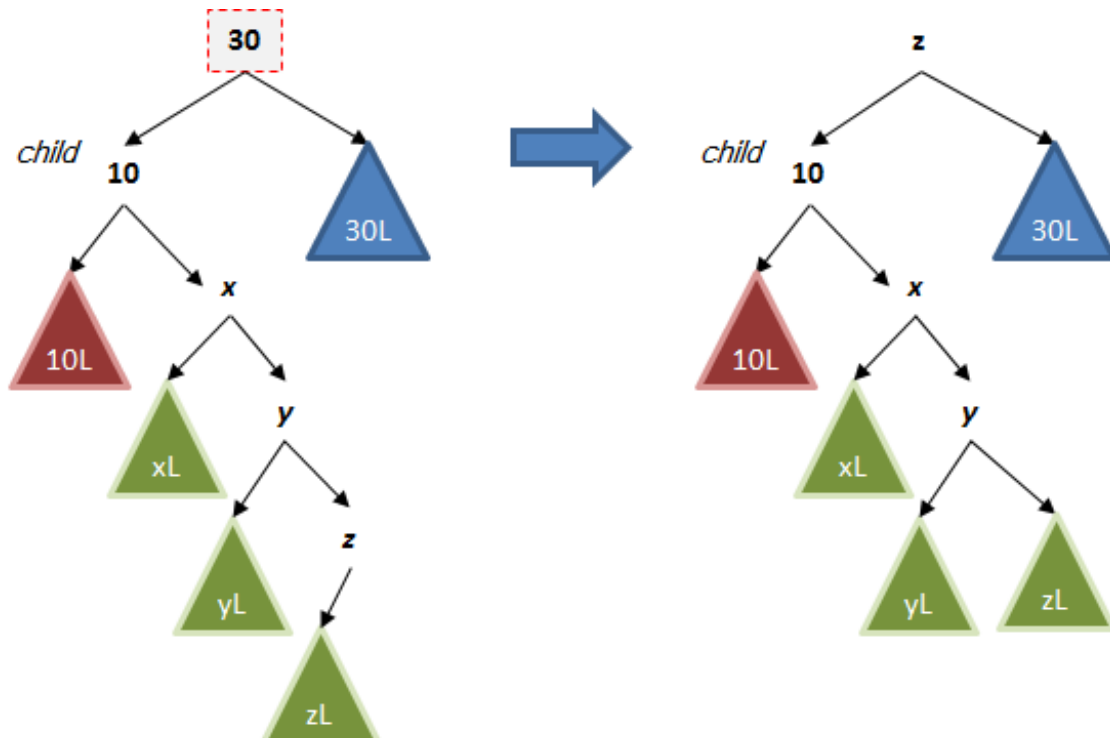
The first three **if** statements in the **updateNodes** method handle the following cases (in this order):

1. The node being deleted is a leaf node, in which case it can be removed entirely.
2. The node being deleted has only a right child, in which case that child is returned.
3. The node being deleted has only a left child, in which case that child is returned.

If the node to be deleted has both left and right children, the more complicated logic must be followed. The goal is to find the right-most descendant of the left child of the node being deleted (**this**). To start, **child** is set to the left child and **grandChild** is the right child (if it exists) of **child**. If **child** has no right child (that is, *grandChild is null*), then **child** itself is the right-most descendant of **this**. The image below describes this case:



The code "lifts" the left sub-tree of **child** to become the left sub-tree of **this** and the key value associated with **this** is set to the **child's** key value. However, if **child** has a right child, the code seeks to find the right-most descendant by traversing the *right* links continually until **grandChild.right** is **null** (in other words, **grandChild** is known to be the right-most child). The image below describes this case. Here the node identified as **z** is the right-most descendant of **child**. It has no right child of its own.



In the resulting modified BST, the key of the node with the value that was removed has been changed to **z** and this will maintain the BST property of the overall tree; in addition, the left sub-tree of **z** (if it exists) has been "lifted up" to be the right child of **y**, which was **z's** former parent.

Note

You could also have selected the left-most descendant of the right child of the node being removed and the logical results would have been the same.

With this modification, you have fully implemented the Binary Search Tree. Modify the **StressTest** code you

have just written to compare **BinaryTree** against **TreeSet**. **BinaryTree** does not enforce set semantics, so **StressTest** makes sure to convert *add* requests for elements already in the set into *contains* requests:

CODE TO TYPE: StressTest class

```

package binary;

import java.util.*;
public class StressTest {
    final static double AddProb      = 0.20;
    final static double ContainsProb = 0.70;
    final static int     SetSize      = 5000;
    final static int     TrialSize     = 50000;
    final static String[] Types      = {"Add", "Contains", "Remove" };

    static void fail(String err) {
        System.err.println("Failed on:" + err);
        System.exit(-1);
    }

    public static void main(String[] args) {
        TreeSet<Integer> base = new TreeSet<Integer>();
        SortedSetBinaryTree<Integer> set = new SortedSetBinaryTree<Integer>();
        double baseCount[] = new double[3];
        double setCount[] = new double[3];
        int counts[] = new int[3];
        long start;
        boolean b,s;
        for (int t = 0; t < TrialSize; t++) {
            int n = (int) (Math.random()*SetSize);
            double choice = Math.random();
            if (choice < AddProb && !base.contains(n)) {
                start = System.nanoTime();
                b = base.add(n);
                baseCount[0] += (System.nanoTime() - start);
                start = System.nanoTime();
                s = set.add(n);
                setCount[0] += (System.nanoTime() - start);
                if (b != s)if (base.contains(n) != set.contains(n)) { fail(Types[0]); }
                counts[0]++;
            } else if (choice < ContainsProb) {
                start = System.nanoTime();
                b = base.contains(n);
                baseCount[1] += (System.nanoTime() - start);
                start = System.nanoTime();
                s = set.contains(n);
                setCount[1] += (System.nanoTime() - start);
                if (b != s) { fail(Types[1]); }
                counts[1]++;
            } else {
                start = System.nanoTime();
                b = base.remove(n);
                baseCount[2] += (System.nanoTime() - start);
                start = System.nanoTime();
                s = set.remove(n);
                setCount[2] += (System.nanoTime() - start);
                if (b != s)if (base.contains(n) != set.contains(n)) { fail(Types[2]); }
                counts[2]++;
            }
        }
        for (int i = 0; i < counts.length; i++) {
            if (counts[i] != 0) {
                baseCount[i] /= counts[i];
                setCount[i] /= counts[i];
            }
            System.out.println(Types[i] + " base=" + (int)baseCount[i]+ " set=" + (int)
setCount[i]);
        }
    }
}

```



Save and run **StressTest** again; the results are much more favorable, although **TreeSet** still outperforms all operations. Your results will likely vary:

OBSERVE: Output of revised StressTest

```
Add base=252 set=330
Contains base=202 set=258
Remove base=247 set=302
```

Successive additions and removals will often result in an unbalanced BST. Since AVL trees are a self-balancing structure, you must now add the removal functionality to AVL trees so they can rebalance themselves after the removal of an element.

Removing Elements From AVL Trees

We can reuse the same logic for deleting a value from an AVL tree to replace its value with the value of the right-most descendant of the left child of the node being removed. Once this action is done, you may have to rebalance a number of other nodes, along the path between the parent of the right-most descendant to the root.

Add this method to the end of the **AVLBinaryTree** class in the **avl** package:

CODE TO TYPE: Modifications to AVLBinaryTree

```
public void remove (E k) {
    if (root == null) { return; }
    root = root.remove(root, k);
}
```

All of the real work takes place in the **AVLBinaryNode** class. You'll need to add these methods to the end of the class:

CODE TO TYPE: Modifications to AVLBinaryNode

```
AVLBinaryNode<E> remove(AVLBinaryNode<E> parent, E k) {
    if (parent == null) { return null; }
    return parent.remove(k);
}

AVLBinaryNode<E> remove (E k) {
    int rc = k.compareTo(key);
    AVLBinaryNode<E> newRoot = this;

    if (rc == 0) {
        if (left == null) {
            return right;
        }

        AVLBinaryNode<E> child = left;
        while (child.right != null) {
            child = child.right;
        }

        E childKey = child.key;
        left = remove(left, childKey);
        key = childKey;

        if (heightDifference(this) == -2) {
            if (heightDifference(right) <= 0) {
                newRoot = this.rotateLeft();
            } else {
                newRoot = this.rightLeftRotation();
            }
        }
    } else if (rc < 0) {
        left = remove(left, k);
        if (heightDifference(this) == -2) {
            if (heightDifference(right) <= 0) {
                newRoot = this.rotateLeft();
            } else {
                newRoot = this.rightLeftRotation();
            }
        }
    } else {
        right = remove(right, k);
        if (heightDifference(this) == 2) {
            if (heightDifference(left) >= 0) {
                newRoot = this.rotateRight();
            } else {
                newRoot = this.leftRightRotation();
            }
        }
    }

    computeHeight(newRoot);
    return newRoot;
}
```

This is a lot to take in! Let's start with the first helper method, **remove(parent,k)**, which removes the value *k* from the sub-tree rooted at **parent**. The real work occurs within the **remove(k)** method. This method has nearly the same structure as the **add(k)** method that already exists (and is repeated below):

OBSERVE: Existing add method in AVLBinaryNode

```
AVLBinaryNode<E> add (E k) {
    int rc = k.compareTo(key);
    AVLBinaryNode<E> newRoot = this;

    if (rc <= 0) {
        left = add(left, k);
        if (heightDifference(this) == 2) {
            if (k.compareTo(left.key) <= 0) {
                newRoot = rotateRight();
            } else {
                newRoot = leftRightRotation();
            }
        }
    } else {
        right = add(right, k);
        if (heightDifference(this) == -2) {
            if (k.compareTo(right.key) > 0) {
                newRoot = rotateLeft();
            } else {
                newRoot = rightLeftRotation();
            }
        }
    }

    computeHeight(newRoot);
    return newRoot;
}
```

The key point to observe is that **whenever the heightDifference of a node exceeds the allowed thresholds**, a rotation occurs. The **remove** method will have three cases to handle; the two cases shown below explain how rotations take place whenever the removal of an element from a node's left sub-tree (or right sub-tree) causes that node to become unbalanced).

OBSERVE: remove method structure

```
AVLBinaryNode<E> remove (E k) {
    int rc = k.compareTo(key);
    AVLBinaryNode<E> newRoot = this;

    if (rc == 0) {
        // perform the deletion
    } else if (rc < 0) {
        left = remove(left, k);
        if (heightDifference(this) == -2) {
            if (heightDifference(right) <= 0) {
                newRoot = this.rotateLeft();
            } else {
                newRoot = this.rightLeftRotation();
            }
        }
    } else {
        right = remove(right, k);
        if (heightDifference(this) == 2) {
            if (heightDifference(left) >= 0) {
                newRoot = this.rotateRight();
            } else {
                newRoot = this.leftRightRotation();
            }
        }
    }

    computeHeight(newRoot);
    return newRoot;
}
```

Let's go over how to perform the deletion:

OBSERVE: Perform deletion of node in AVL tree

```
if (rc == 0) {
    if (left == null) {
        return right;
    }

    AVLBinaryNode<E> child = left;
    while (child.right != null) {
        child = child.right;
    }


    E childKey = child.key;
    left = remove(left, child.key);
    key = childKey;

    if (heightDifference(this) == -2) {
        if (heightDifference(right) <= 0) {
            newRoot = this.rotateLeft();
        } else {
            newRoot = this.rightLeftRotation();
        }
    }
}
```

This code immediately checks **whether there is even a left child for the node being deleted**; if not, the right sub-tree is "lifted" to take its place.

If the left child is present, the code **locates the right-most descendant quickly, child**. The method then uses double recursion to invoke **remove(left, child.key)** to remove the **child.key** value from the sub-tree rooted at **left** and replace the key for the node being deleted with **child.key**. Once this task is complete, you know that the sub-tree rooted at **left** is balanced properly. Then our code **checks to see if any rotation is needed**; because you only looked for the right-most descendant on the left side of the tree, you only need to consider two rotation cases, which are complementary to the cases in the **add** method (except that you are removing elements, not adding them).

As with the **add** method, the rebalancing may occur at any time between the original location of the value being deleted and the path from that node to the root in the tree. So, there may be a total of $O(\log n)$ rotations whenever you remove an element from an AVL tree. For this reason, the *Red-Black Tree* implementation of **TreeSet** in the Java Collections Framework is more efficient than an AVL implementation when inserting elements into the tree. At the same time, the AVL tree is more compact than the *Red-Black Tree* implementation, which means the *contains* queries are going to be faster. Review the above code to make sure you understand how each of the constituent elements works to self-balance the tree automatically.

 In your **Conclusion** project **/src** source folder, **avl** package, create an **AVLStressTest** class as shown. This class is complicated because it contains code to validate that the *AVL Property* of the **AVLBinaryTree** is not violated after any addition or deletion:

CODE TO TYPE: AVLStressTest class

```
package avl;

import java.util.*;

public class AVLStressTest {
    final static double AddProb      = 0.20;
    final static double ContainsProb = 0.70;
    final static int     SetSize      = 5000;
    final static int     TrialSize     = 50000;
    final static String[] Types      = {"Add", "Contains", "Remove" };

    static void fail(String err) {
        System.err.println("Failed on:" + err);
        System.exit(-1);
    }

    static int height (AVLBinaryNode<?> n) {
        if (n == null) { return 0; }
        return 1 + Math.max( height(n.left), height(n.right));
    }

    public static int height (AVLBinaryTree<?> tree) {
        if (tree.root == null) { return -1; }
        return height(tree.root);
    }

    static boolean validateAVLProperty (AVLBinaryNode<?> n) {
        if (n == null) { return true; }

        int leftHeight = 0;
        if (n.left != null) { leftHeight = height(n.left); }
        int rightHeight = 0;
        if (n.right != null) { rightHeight = height(n.right); }

        int diff = leftHeight - rightHeight;
        if (diff < -1 || diff > 1) { return false; }

        return validateAVLProperty (n.left) && validateAVLProperty (n.right);
    }


    static boolean validateAVLProperty(AVLBinaryTree<?> tree) {
        if (tree.root == null) { return true; }
        return validateAVLProperty(tree.root);
    }

    public static void main(String[] args) {
        TreeSet<Integer> base = new TreeSet<Integer>();
        AVLBinaryTree<Integer> set = new AVLBinaryTree<Integer>();
        double baseCount[] = new double[3];
        double setCount[] = new double[3];
        int counts[] = new int[3];
        long start;
        boolean b,s;
        for (int t = 0; t < TrialSize; t++) {
            int n = (int) (Math.random()*SetSize);
            double choice = Math.random();
            if (choice < AddProb && !base.contains(n)) {
                start = System.nanoTime();
                b = base.add(n);
                baseCount[0] += (System.nanoTime() - start);
                start = System.nanoTime();
                set.add(n);
                setCount[0] += (System.nanoTime() - start);
                if (!validateAVLProperty(set)) { fail(Types[0]); }
                if (base.contains(n) != set.contains(n)) { fail(Types[0]); }
                counts[0]++;
            } else if (choice < ContainsProb) {
```

```

        start = System.nanoTime();
        b = base.contains(n);
        baseCount[1] += (System.nanoTime() - start);
        start = System.nanoTime();
        s = set.contains(n);
        setCount[1] += (System.nanoTime() - start);
        if (b != s) { fail(Types[1]); }
        counts[1]++;
    } else {
        start = System.nanoTime();
        b = base.remove(n);
        baseCount[2] += (System.nanoTime() - start);
        start = System.nanoTime();
        set.remove(n);
        setCount[2] += (System.nanoTime() - start);
        if (!validateAVLProperty(set)) { fail(Types[2]); }
        if (base.contains(n) != set.contains(n)) { fail(Types[2]); }
        counts[2]++;
    }
}
for (int i = 0; i < counts.length; i++) {
    if (counts[i] != 0) {
        baseCount[i] /= counts[i];
        setCount[i] /= counts[i];
    }
    System.out.println(Types[i] + " base=" + (int)baseCount[i] + " set=" + (int)
setCount[i]);
}
}
}

```

 Save and run it. It will take longer to complete because of the validation code. The **validateAVLProperty** method validates that the height difference for every node in the AVL binary tree is within the required tolerance as demanded by the *AVL Property*.

OBSERVE: AVLStressTest output

```

Add base=305 set=464
Contains base=270 set=227
Remove base=360 set=411

```

When reviewing this result, observe that the baseline **TreeSet** implementation still outperforms AVL binary trees when it comes to *adding* and *removing* elements. However, the *contains* query is now almost 20% faster in the AVL binary tree, because of its more compact structure. It's always satisfying when empirical evidence supports the expected results.

Removing Elements From KD-trees

Given the success we've had deleting nodes from binary trees, you'd expect to be able to do the same with the kd-tree. Unfortunately, this will be impossible because the kd-tree alternates its horizontal and vertical partitioning levels within its structure. That is, while the structure of a kd-tree resembles a Binary Search Tree, you cannot simply "lift" nodes one level up as you have been able to do for the BST and AVL trees described earlier in this lesson.

Since you can't remove elements from a kd-tree easily, what can you do? Instead of rebuilding the kd-tree with each deletion, consider a strategy that marks elements as *deleted* (which takes $O(\log n)$ time) and then the kd-tree can reconstitute itself automatically whenever the ratio of deleted nodes to present nodes in the tree exceeds some threshold.

There is a comparative precedent for this behavior in hashtables, such as **HashMap**, to automatically resize themselves when the number of entries exceeds some inner threshold based on the *load capacity* of the storage.

Copy the **kd** package from your **Multidimension** project into your **Conclusion** project **/src** folder. In the **kd** package, modify the **KDNode** class as shown:

CODE TO TYPE: Modifications to KDNode

```
package kd;

import java.awt.Point;

public class KDNode {
    final Point point;
    final int direction;
    Region region;
    KDNode above;
    KDNode below;
    boolean deleted;

    public static final int HORIZONTAL = 0;
    public static final int VERTICAL = 1;

    public KDNode(Point p, int dir, Region r) {
        this.point = new Point (p);
        this.direction = dir;

        this.region = new Region(r);
    }

    public KDNode(Point p, int dir) {
        this (p, dir, Region.max);
    }

    public boolean isBelow(Point p) {
        if (direction == VERTICAL) {
            return p.x < point.x;
        } else {
            return p.y < point.y;
        }
    }

    public boolean isAbove(Point p) {
        if (direction == VERTICAL) {
            return p.x >= point.x;
        } else {
            return p.y >= point.y;
        }
    }

    public boolean isDeleted() { return deleted; }

    public void boolean add (Point p) {
        if (p.equals(point)) {
            if (deleted) {
                deleted = false;
                return true;
            }
            return false;
        }

        if (isBelow(p)) {
            if (below == null) {
                below = createChild (p, true);
                return true;
            } else {
                return below.add(p);
            }
        } else {
            if (above == null) {
                above = createChild (p, false);
                return true;
            } else {
                return above.add(p);
            }
        }
    }
}
```

```

    }
}

KDNode createChild (Point p, boolean below) {
    Region r = new Region (region);
    if (direction == VERTICAL) {
        if (below) {
            r.x_max = point.x;
        } else {
            r.x_min = point.x;
        }
    } else {
        if (below) {
            r.y_max = point.y;
        } else {
            r.y_min = point.y;
        }
    }
    return new KDNode(p, 1-direction, r);
}
}

```

The first modification is to associate a **deleted** attribute with each **KDNode** object. If this value is **true** for a node, its associated point no longer belongs in the set, but it remains within the kd-tree to provide the necessary structure. An **isDeleted** method is provided to determine the status of a **KDNode** object.

In the initial implementation, the **add** method had nothing to return. Now you need to know whether the set changed as a result of the invocation. This is the same behavior designed by the Java Collections Framework. The code cannot return **false** if the point being added already exists within the kd-tree because, after all, it may have previously been deleted, in which case the code flips the **deleted** attribute value to **false** before returning **true** to signal that the set has changed.

We'll make more significant changes in the **KDTree** class now:

CODE TO TYPE: Modifications to KDTree

```
package kd;

import java.awt.Point;

public class KDTree {
    KNode root;
    int deletedCount;
    int totalCount;
    float loadFactor;
    static float DEFAULT_LOAD_FACTOR = 0.5f;

    public KDTree() {
        this (DEFAULT_LOAD_FACTOR);
    }

    public KDTree(float factor) {
        root = null;
        loadFactor = factor;
    }

    public booleanvoid add (Point value) {
        if (root == null) {
            root = new KNode(value, KNode.VERTICAL);
            totalCount = 1;
            return true;
        } else {
            if (root.add(value) +) {
                totalCount++;
                return true;
            }
            return false;
        }
    }

    void recreate() {
        KNode oldRoot = root;
        root = null;

        int remaining = totalCount - deletedCount;
        totalCount = deletedCount = 0;
        if (remaining == 0) {
            return;
        }

        fill(oldRoot);
    }

    void fill(KNode n) {
        if (n == null) { return; }

        if (!n.deleted) {
            add(n.point);
        }

        fill(n.below);
        fill(n.above);
    }

    public boolean remove (Point p) {
        KNode exist = find(p);
        if (exist != null && !exist.deleted) {
            exist.deleted = true;
            deletedCount++;

            if (deletedCount*1.0/totalCount >= loadFactor) {
                recreate();
            }
        }
    }
}
```

```

    }
    return true;
}
return false;
}

public KDNode find(Point p) {
    return find(root, p);
}

KDNode find (KDNode node, Point p) {
    if (node == null) { return null; }
    if (node.point.distance(p) < 5) { return node; }

    if (node.isBelow(p)) {
        return find(node.below, p);
    } else {
        return find(node.above, p);
    }
}
}

```

Let's look at the code more closely:

OBSERVE: KDTree Modified State

```

int deletedCount;
int totalCount;
float loadFactor;
static float DEFAULT_LOAD_FACTOR = 0.5f;

public KDTree() {
    this (DEFAULT_LOAD_FACTOR);
}

public KDTree(float factor) {
    root = null;
    loadFactor = factor;
}

```

Each **KDTree** object maintains a **totalCount** of values in the kd-tree, as well as the **deletedCount** of points that have been removed. Whenever the ratio of deleted points to total points is greater than or equal to **loadFactor**, the kd-tree is recreated to contain only the non-deleted points. The user can specify a **load factor** on construction; if they don't specify, **the default is 0.5**.

OBSERVE: Modified add method

```

public boolean add (Point value) {
    if (root == null) {
        root = new KDNode(value, KDNode.VERTICAL);
        totalCount = 1;
        return true;
    } else {
        if (root.add(value)) {
            totalCount++;
            return true;
        }
        return false;
    }
}

```

The **add** method is changed to update the **totalCount** of values in the kd-tree; now it also **returns true** to reflect a change to its underlying set.

The new functionality is contained in the **remove** method:

OBSERVE: Remove method added to KDTree

```
public boolean remove (Point p) {
    KDNode exist = find(p);
    if (exist != null && !exist.deleted) {
        exist.deleted = true;
        deletedCount++;

        if (deletedCount*1.0/totalCount >= loadFactor) {
            recreate();
        }
        return true;
    }
    return false;
}
```

The **remove** method **returns true** when the set has changed. Accordingly, it must first check to see if **the point even exists within the tree**; if it does, it must make sure that **the associated node has not already been deleted**. Assuming it has not, **the node is marked as being deleted and the corresponding deletedCount is incremented** for the kd-tree. At this point it is possible that **the ratio of deleted nodes to total nodes exceeds the loadFactor threshold**, at which point the entire kd-tree is reconstructed to contain only the non-deleted nodes. This is accomplished in the **recreate** and **fill** methods:

OBSERVE: Code to reconstruct kd-tree from non-deleted nodes

```
void recreate() {
    KDNode oldRoot = root;
    root = null;

    int remaining = totalCount - deletedCount;
    totalCount = deletedCount = 0;
    if (remaining == 0) {
        return;
    }

    fill(oldRoot);
}

void fill(KDNode n) {
    if (n == null) { return; }

    if (!n.deleted) {
        add(n.point);
    }


    fill(n.below);
    fill(n.above);
}
```

The **recreate** method checks to see **if all points have been deleted**. If so, it can **return** after **setting root to null**. If there are any points left though, **all of the non-deleted points in the oldRoot are processed by fill**.

The **fill** method performs a pre-order traversal of the kd-tree; **for all non-deleted nodes**, the **associated n.point is inserted into the new kd-tree**. The recursive call makes sure to traverse both the **below** and **above** sub-trees for each node.

To demonstrate the new capability in action, write the following application, which allows the user to add points to the kd-tree by clicking with the left button and to remove points by clicking with the right button.

The **KDTree** class makes an important design decision to enable the **find** method to return the associated **KDNode** for the requested point being searched. This is important because it is up to the caller to determine whether the node represents a deleted point in the kd-tree or a valid point. In doing so, the code is able to draw deleted nodes with an "X" while non-deleted nodes are filled in squares.

 In the **/src** source folder **kd** package, create a **KDAppletDelete** class as shown:

CODE TO TYPE: KDAppletDelete class

```
package kd;

import java.awt.*;
import java.awt.event.*;

public class KDAppletDelete extends java.applet.Applet {
    KDTree tree = new KDTree();
    KNode match = null;
    Image bufferImage;
    Graphics bufferGraphics;

    int toAWT(int y) {
        if (y == Region.maxValue) { return 0; }
        int awty = getHeight();
        if (y != Region.minValue) { awty -= y; }
        return awty;
    }

    int toCartesian(int awty) { return getHeight() - awty; }

    public void init() {
        setSize(400,400);

        addMouseListener (new MouseAdapter() {
            public void mouseClicked(MouseEvent me) {
                Point pt = new Point (me.getX(), toCartesian(me.getY()));
                if (me.getButton() == MouseEvent.BUTTON3) {
                    KNode match = tree.find(pt);
                    if (match != null) {
                        tree.remove(match.point);
                        redraw();
                        drawNode(bufferGraphics, match.point, true, true);
                        repaint();
                    }
                } else {
                    tree.add(pt);
                    redraw();
                    repaint();
                }
            }
        });

        addMouseMotionListener (new MouseAdapter() {
            public void mouseMoved(MouseEvent me) {
                Point pt = new Point (me.getX(), toCartesian(me.getY()));
                KNode newMatch = tree.find(pt);
                if (match != newMatch) {
                    match = newMatch;
                    redraw();
                    if (match != null) {
                        drawNode(bufferGraphics, match.point, true, match.deleted);
                    }
                    repaint();
                }
            }
        });
    }

    void drawNode(Graphics g, Point p, boolean selected, boolean deleted) {
        if (selected) {
            g.setColor(Color.RED);
            g.clearRect(p.x - 4, toAWT(p.y) - 4, 8, 8);
        }
        if (deleted) {
            g.drawRect(p.x - 4, toAWT(p.y) - 4, 8, 8);
            g.drawLine(p.x - 4, toAWT(p.y) - 4, p.x + 4, toAWT(p.y) + 4);
        }
    }
}
```

```

        g.drawLine(p.x - 4, toAWT(p.y) + 4, p.x + 4, toAWT(p.y) - 4);
    } else {
        g.fillRect(p.x - 4, toAWT(p.y) - 4, 8, 8);
    }
    g.setColor(Color.BLACK);
}

public void paint(Graphics g) {
    if (bufferImage == null) {
        bufferImage = createImage(getWidth(), getHeight());
        bufferGraphics = bufferImage.getGraphics();
    }

    if (tree.root == null) {
        g.drawString("Click to add points", 150, 200);
    } else {
        g.drawImage(bufferImage, 0, 0, this);
    }
}

void redraw() {
    bufferGraphics.clearRect(0, 0, getWidth(), getHeight());
    visit(bufferGraphics, tree.root);
}

void drawPartition (Graphics g, Region r, Point p, int type, boolean deleted)
{
    if (type == KDNode.VERTICAL) {
        g.drawLine(p.x, toAWT(r.y_min), p.x, toAWT(r.y_max));
    } else {
        int xlow = r.x_min;
        if (r.x_min == Region.minValue) { xlow = 0; }
        int xhigh = r.x_max;
        if (r.x_max == Region.maxValue) { xhigh = getWidth(); }
        g.drawLine(xlow, toAWT(p.y), xhigh, toAWT(p.y));
    }
    drawNode(g, p, false, deleted);
}

void visit (Graphics g, KDNode n) {
    if (n == null) { return; }
    drawPartition(g, n.region, n.point, n.direction, n.deleted);

    visit (g, n.below);
    visit (g, n.above);
}
}

```

Much of this code is similar to the applets you wrote for an earlier lesson, but there are a some differences. Let's look at the code more closely:

OBSERVE: draw Method

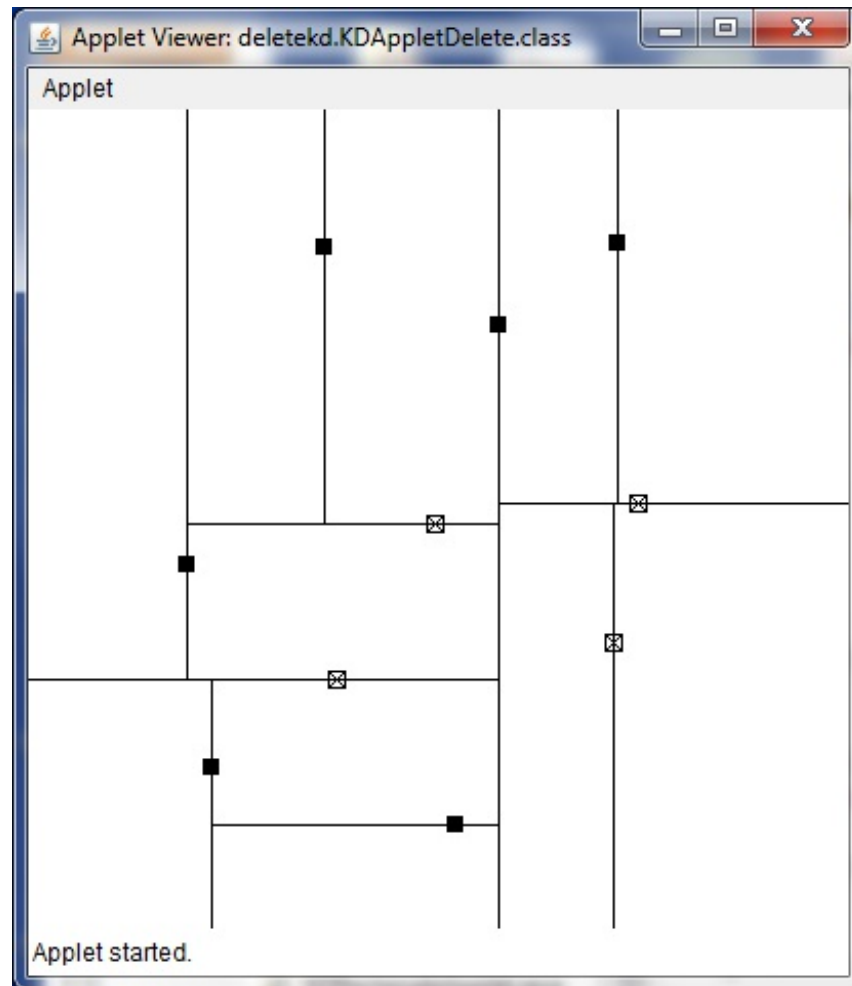
```

void drawNode(Graphics g, Point p, boolean selected, boolean deleted) {
    if (selected) {
        g.setColor(Color.RED);
        g.clearRect(p.x - 4, toAWT(p.y) - 4, 8, 8);
    }
    if (deleted) {
        g.drawRect(p.x - 4, toAWT(p.y) - 4, 8, 8);
        g.drawLine(p.x - 4, toAWT(p.y) - 4, p.x + 4, toAWT(p.y) + 4);
        g.drawLine(p.x - 4, toAWT(p.y) + 4, p.x + 4, toAWT(p.y) - 4);
    } else {
        g.fillRect(p.x - 4, toAWT(p.y) - 4, 8, 8);
    }
    g.setColor(Color.BLACK);
}

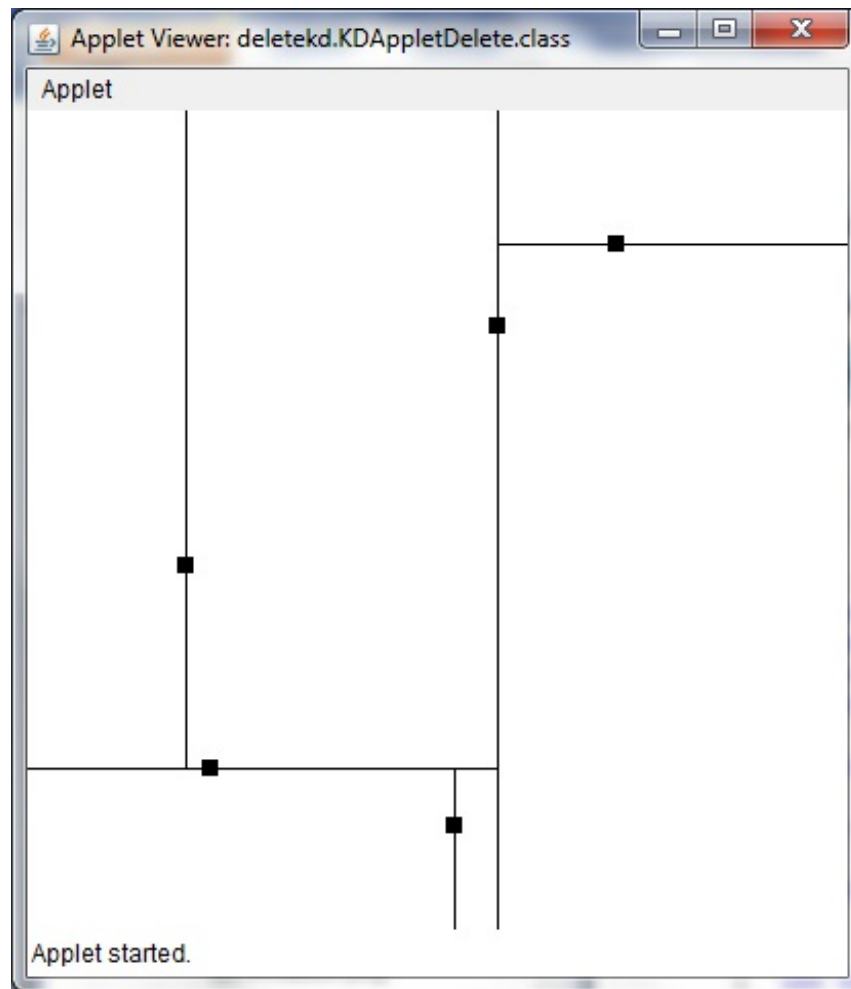
```


The **draw** method draws a node accurately, whether it is marked for deletion or selected by the user. The **mouseClicked**, **mouseMoved**, and **drawPartition** methods all invoke **draw** as needed. The mouse handlers operate as before, but now right mouse clicks (as designated by **MouseEvent.BUTTON3**) are used to remove points from the kd-tree.

▶ Run **KDAppletDelete** and add ten points using the left mouse button. Then select five different points for deletion. As you select the first four points, the applet redraws those points using a small "x" as shown:



Once you select the fifth point for deletion, the kd-tree will reassemble itself automatically with only five points because the ratio of deleted points to actual points has hit the predetermined ratio of 50%. The image below shows the resulting kd-tree once reconstructed:



The resulting reconstructed kd-tree is not likely balanced, but there is an algorithm, described in the *Algorithms in a Nutshell* book, which enables you to create a balanced kd-tree from any selection of points.

Lessons Learned

- **Complicated data structures have invariants that must be maintained under addition and removal.**
- **Methods that return void miss an opportunity to return useful information:** Consider the `add` method in the Collections Framework and how it returns `true` when the collection changes but `false` otherwise. This bit of information is extremely helpful in several algorithms.
- **Divide and Conquer is an extremely powerful strategy:** The algorithms that deliver $O(n \log n)$ performance often do so by using this technique to divide a problem into two (or more) smaller subproblems, whose results are combined to produce the appropriate answer. You have seen this in **MergeSort**.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.