### 1.1 Get oriented to the UC-WISE system.

#### (Display page) Overview

**Overview**

The UC-WISE system that you'll use in CS 61BL this semester lets us organize activities that you'll be doing inside and outside class. Notice the sidebar on the left; it organizes the activities that you'll do during class. Two important activities are "brainstorming" and online discussion. You'll get practice with these concepts in this lesson.

#### (Brainstorm) Brainstorming

What's your favorite restaurant in Berkeley?

#### (Discussion Forum) Online discussion

Explain what's so good about the restaurant(s) you mentioned in the previous step, and comment on one of the opinions of your classmates.

#### (Display page) The "extra brain"

The "Extra Brain"

In the upper right part of the sidebar, there's an icon that's intended to resemble a brain. This is your "extra brain", where you can collect tips, things to remember, interesting programming examples, or anything else you'd like to keep track of in this course. Click on the brain icon and put a comment about a Berkeley restaurant you might want to try in it. Then click on the icon that looks like a diskette to save your brain entry.

### 1.2 Get oriented to the Eclipse programming environment.

#### (Display page) Review of some UNIX commands

#### Review of some UNIX commands

First, use the `mkdir` command to create a directory named `day1` in your home directory. The `mkdir` command takes a single argument, the name of the directory to create. Then use the `cp` command to copy a file from the CS 61B code directory to the `day1` directory. The `cp` command takes two arguments:

- the file to be copied, and
- the destination file or directory.

The file to be copied is `~/cs61b/code/YearCheck.stuff`. Note: that's `~/cs61b`, NOT `~cs61b`.

#### (Brainstorm) Three ways to copy a file

Give at least three different ways to copy the `YearCheck.stuff` file from the CS 61B `code` directory to your `day1` directory. Hint: some ways involve use of the `~` (home directory) and `.` (working directory) abbreviations.

#### (Display page) Introduction to Eclipse

### Introduction to Eclipse

Eclipse is the IDE (integrated development environment) we'll be using in class. Start Eclipse either by typing

```
eclipse &
```

in an xterm window or by choosing it from the application menu produced by clicking the right mouse button outside a window. Eclipse will first ask you where to maintain its "workspace" (information about the programs you create with it). Its default choice, the `workspace` directory in your home directory, is a reasonable choice. (You may click on the check box that says to use the workspace directory by default.) After taking some time to get everything initialized, it will present you with a "Welcome" window. Close this window by clicking on its "X". You will now build a Java program. This is a two-step process: first you create an Eclipse project, then you create the Java program file.

1. Start by selecting "New:Project" from the File menu. Select the "Java Project" wizard. Name your project `day1` (with "Create project in workspace", "Use default JRE", and "Use project folder as root..." selected). Then click the "Finish" button. If it asks you about opening the Java perspective, say yes (and tell it not to ask again).

2. Now select "New:Class" from the File menu. Provide the class name `YearStuff`. Answer the question "Which method stubs would you like to create?" by selecting `public static void main (String [] args)` and unselecting the others. Then click "Finish". Eclipse should provide you with a program framework.

#### (Display page) Format of a Java program

### Format of a Java program

Java programs we'll be writing early in CS 61BL each consist of `class definitions`. A definition provides the name of the class and, you'll recall from CS 61A, its `variables` and `methods`. In Java, a class definition starts with the word "class" (possibly preceded by the word "public" or "private"), continues with the name of the class and the character "{" (a left brace or curly bracket), and ends with the character "}" (a right brace). Inside the braces are definitions of `variables` and `methods`. A method definition similarly starts with naming information followed by a pair of braces that include the statements of the method. The braces act in this way like parentheses in Scheme. Statements within such a group are separated with semicolons. Here's an example of a "Hello world" program in Java (traditionally, the simplest program in a given language):

```java
class SimpleGreeter {
    public static void main (String [] arguments) {
        System.out.print ("hello ");
        System.out.println ("world");
    }
}
```

---

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
At least one of the classes in a Java program must contain a method named `main`. The `main` method is what's called by the operating system to run the program. There are two formats for comments in a Java program. One, intended for short comments, starts with two consecutive slash characters and continues to the end of the line:

```java
// this is one kind of comment
```

The other comment format starts with the character sequence `/*` (slash followed by asterisk); it continues, perhaps over more than one line, and is terminated by the character sequence `*/` (asterisk, slash). Here's an example:

```java
/*
  This is a
  multi-
  line comment */
```

Some commenting styles add single asterisks at the start of each line in the comment after the first, to improve readability:

```java
/*
  * This is a
  * multi-
  * line comment */
```

Eclipse automatically provides the "boilerplate" for a new class: a couple of identifying comments, the header for a class definition, and the header for a `main` method.

### Creating a program

Return to the Eclipse window. (You won't need the Outline or xpath panes, so you can close these by clicking on the "X" in their tabs.) Now copy the contents of the `YearCheck.stuff` file (reproduced below for your convenience) into the `main` method. Don't retype the lines; use the mouse to copy one line at a time from your browser window into the Eclipse window. (Select the text to copy with the left mouse button. Then click in the `YearStuff.java` window at the position where the text should go, and click with the middle mouse button to copy the selection.)

```java
class YearStuff {
    public static void main(String args[]) {
        int year = 2000;
        System.out.println (year + " is a leap year.");
        System.out.println (year + " is not a leap year.");
    }
}
```

The lines of `YearCheck.stuff`, when arranged in correct sequence and augmented by several right braces, will provide a correct body for a method that determines if the value stored in the variable `year` is a legal leap year (divisible by 400, or divisible by 4 and not by 100). You will need more than one copy of some of the lines. Some Java vocabulary used in the code:

- The `/` operator implements the "remainder" operation. Thus the value of `year % 4` will be one of 0, 1, 2, or 3.
- The `==` operator compares two values for equality. One would read the code fragment `if (year % 4 == 0)` as "if the remainder when dividing `year` by 4 is equal to 0, ...".
- The `System.out.println` prints its argument.
- When one of the arguments of the `+` operator is a string, the string concatenation operation is performed.

As you copy each line, observe the automatic indenting and the highlighting of matching braces. A small red rectangle to the right of a line indicates a syntax error on that line. Once you assemble the complete program, you can test this by changing one of the occurrences of `println` to `print`. A red icon to the left of a line indicates that Eclipse can correct the error; click on that icon, then on the appropriate choice to fix the error.

### Running a program

Once you have a syntactically correct program, you run it by selecting "Run As: Java Application" from the Run menu. When Eclipse asks you about saving your program file, click "OK". The output "2000 is a leap year." should appear in the Console window. Re-running the program can be done by selecting "Run Last Launched" from the "Run" menu. (This also has a keyboard equivalent.) You should run your program (at least) four times, changing the value of the `year` variable each time. Test your program on the following `year` values:

- 2000 (a leap year)
- 2004 (a leap year)
- 1900 (not a leap year)
- 2003 (not a leap year)

Any errors you encounter are probably due to putting the tests in the wrong order. Fix the errors by reordering the tests (don't change the tested expressions). The editor keeps track of what lines have changed since the program has last saved. Note how it does this as you make the successive changes to `year`. Try a few other program modifications to see what happens. Record any features of interest in your Extra Brain.

### Executing a program under control of the debugger

Eclipse environment includes a debugger that we'll practice using now. Double-click in the left margin by the first `if` test. A blue dot should appear, indicating that a breakpoint is set at that statement. Now run the program, selecting "Debug As" from the "Run" menu. After saving your program, Eclipse will ask about opening the Debug perspective; click "Remember my decision" and "Yes". It will then display the program with the `if` statement highlighted; it has suspended execution at that point, and is waiting for instructions to proceed. Observe the "Variables" window, which lists all the values of variables accessible at the point where the program is suspended. At this point, the `year` Variable and its value should appear in the window. Other actions available when the program is stopped include:

- The editor keeps track of what lines have changed since the program has last saved. Note how it does this as you make the successive changes to `year`. Try a few other program modifications to see what happens. Record any features of interest in your Extra Brain.

### Executing a program under control of the debugger

The Eclipse environment includes a debugger that we'll practice using now. Double-click in the left margin by the first `if` test. A blue dot should appear, indicating that a breakpoint is set at that statement. Now run the program, selecting "Debug As" from the "Run" menu. After saving your program, Eclipse will ask about opening the Debug perspective; click "Remember my decision" and "Yes". It will then display the program with the `if` statement highlighted; it has suspended execution at that point, and is waiting for instructions to proceed. Observe the "Variables" window, which lists all the values of variables accessible at the point where the program is suspended. At this point, the `year` Variable and its value should appear in the window. Other actions available when the program is stopped include:
is suspended include the following, all accessible through the "Run" menu:

- "Step over" executes the next statement, and suspends the program immediately thereafter.
- "Step into", if the program is suspended at a method call, executes the first statement of the method and suspends the program. Don't step into system-provided methods.
- "Run to line" resumes the program, suspending execution when the statement containing the edit cursor is about to be executed.

Practice with these, and keep notes if necessary in your Extra Brain. Clicking "Java" in the upper right corner of the Eclipse window gets you back to the Java "perspective", ready for more editing.

1.2.8 (Brainstorm) Another editor feature
What is the purpose of the circle with the minus in it, in the second column of the window displaying program text? Why might this feature be useful?

1.2.9 (Self Test) Review
How can you tell which lines of the program have been changed since the program was last saved?

How do you determine which "}" is the matching brace for a given "{"?

How would you tell that there's a syntax error in your program without trying to run it?

How do you tell the debugger to suspend execution at a given line?

Suppose you set a breakpoint, but the program didn't stop there. Explain what you would do about this.

1.3 Practice running Java programs without Eclipse, and using the submission system
(2 steps)

1.3.1 (Display page) The javac and java programs

The javac and java programs

The javac program is the Java compiler. It translates a Java class (together with any other classes it uses) into low-level instructions to the Java Virtual Machine. The arguments to the javac command are all names of Java source files, whose names end in "java". Eclipse has saved your YearStuff.java file in its workspace directory. Find it there, and copy the YearStuff.java file to the day1 directory you created earlier. Then move to the day1 directory, and compile YearStuff.java. This should produce a file named YearStuff.class. The java program is the Java Virtual Machine interpreter. The java command, given the name of a class containing a main method, executes the program. Run the YearStuff program. Its output will appear in your editor window.

1.3.2 (Display page) The submission system

The submission system

The homework submission system is the same one (authored by Paul Hilfinger) as you may have used in CS 61A. Run you YearStuff program again, redirecting the output to a file named YearStuff.out. Do this using the ">" operator:

```
java YearStuff > YearStuff.out
```

Then submit the program and its output using the submit command:

```
submit day1
```

It will complain if your file names are different from what we've specified.

1.5 Homework
(2 steps)

1.5.1 (Display page) Readings and discussion contribution
Readings and discussion contribution

Read the "Intro" and chapters 1 and 2 in Head First Java. Also skim the section "Moving from CS 61A Scheme to CS 61B Java" (accessible online via the "Resources" sidebar item) to get an idea of the similarities and differences between Scheme and Java. Sections 7.1 and 7.2 in Objects, Abstraction, Data Structures and Design using Java provide a review of recursion that will come in handy in upcoming labs. Finally, do some more experimenting with Eclipse, and contribute at least one post to the following discussion.

1.5.2 (Discussion Forum) Interesting aspects of Eclipse
Experiment some more with Eclipse, and contribute at least one post about an interesting result of your experiments that is likely to prove useful later in the semester.

Entire day brainstorm answers™

2 Practice with loops and conditionals 2008-1-24 ~ 2008-1-25 (3 activities)
2.2 Get comfortable with Java loops and conditionals (13 steps)
2.2.1 (Display page) Java statements

Java statements
Chapter 1 of Head First Java and earlier lab activities introduced you to Java statements, the components of Java method bodies. There are several kinds of statements:

- an assignment statement, which stores a value into a variable;
- a method call, for instance, System.out.print ("hello");
- an if statement, which conditionally execute code based on the result of a test;
- a while statement, which conditionally repeats code based on a continuation test.

A while statement is often referred to as a loop. An assignment statement behaves like a set: in Scheme. A method call is like a Scheme procedure call. We assume that you're reasonably comfortable with this—at least in concept—and for remaining activities in this lab will focus on if and while statements.

2.2.2 (Display page) How if and if ... else work

How if and if ... else work

There are three types of if statements. All start with the word if and are followed by a test—a
boolean expression—in parentheses, and a sequence of statements surrounded by braces, for example,

```java
if (year % 4 == 0) {
    System.out.println (year + " is a leap year.");
} else {
    System.out.println (year + " is definitely not a leap year.");
}
```

The braces aren't necessary if there is only one statement in the sequence; however, it is good practice always to include them, since it simplifies later modification of the code. Tests often involve comparisons. The comparison operators in Java are == and == for equality and inequality testing, and >, >=, <, and <= for comparison of magnitudes. The three kinds of if differ in what happens if the test is unsuccessful. For example, the if statement above has no effect if year's value is not divisible by 4. A second kind of if specifies an alternative, using the else keyword and another sequence of statements surrounded by braces. Here's an example:

```java
if (year % 4 == 0) {
    System.out.println (year + " is a leap year.");
} else if (year % 100 == 0) {
    System.out.println (year + " is a leap year.");
} else if (year % 400 == 0) {
    System.out.println (year + " is a leap year.");
} else {
    System.out.println (year + " is definitely not a leap year.");
}
```

A third kind of if adds a test to the failure case, for example:

```java
if (year % 4 = 0) {
    System.out.println (year + " is not a leap year.");
} else if (year % 100 != 0) {
    System.out.println (year + " is a leap year.");
} else if (year % 400 != 0) {
    System.out.println (year + " is not a leap year.");
} else {
    System.out.println (year + " is a leap year.");
}
```

These have straightforward counterparts in Scheme's if and cond special forms:

```scheme
(if (= (remainder year 4) 0)
    (display (cons year '(is a leap year)))
    (display (cons year '(is not a leap year)))
)
```

```scheme
(cond
    [(not (= (remainder year 4) 0))
        (display (cons year '(is a leap year)))
    ]
    [(not (= (remainder year 100) 0))
        (display (cons year '(is not a leap year)))
    ]
    [(not (= (remainder year 400) 0))
        (display (cons year '(is a leap year)))
    ]
    [else
        (display (cons year '(is a leap year)))
    ]
)
```

2.2.3 (Display page) How while works

How while works

The while statement is used to repeat a given sequence of statements. It consists of the word while, followed by a continuation test in parentheses, followed by a sequence of statements to repeat, enclosed in braces. The statement sequence is called the loop body. The while statement works by evaluating the test expression; if it's true, the entire loop body is executed. Then the test is checked again; if it succeeds, the entire loop body is executed again. This continues, possibly infinitely. Here's an example that implements the remainder operation dividend divided by divisor, and produces some output. We assume all variables have already been declared, and that divisor and dividend have already been assigned positive values.

```java
while (dividend >= divisor) {
    dividend = dividend - divisor;
}
```

All statements of the loop body are executed, even if one of them affects the truth value of the test. In the example above, values of 9 for dividend and 4 for divisor result in two lines of output. (Some students just learning a Java-like language think that the behavior of a while is more like what its use in English would suggest, namely suspension of the loop execution in midstream. This is not the case.)

2.2.4 (Display page) Collaboration activities

Collaboration activities

The subsequent activities in today's lab and many of the lab activities later in the course involve collaboration with one of your classmates. Each day for the first couple of weeks, you'll be requested to choose a different person to work with. Why do we care about collaboration? Why don't we just let you work by yourself? There are several reasons.

- Industry is continually nagging us to make sure our graduates have experience working in teams. Toward that end, we include some team programming in CS 61A, and we continue with collaboration activities in CS 61B.
- Most of the upper-division programming courses involve large team projects. If you haven't learned good collaboration techniques by then, you're liable to doom your team.
- The last project in CS 61B will be done in partnership. Earlier collaborative activities in the course will help you pick a partner with whom you work productively.
- In almost all cases, you learn by collaborating:
  - explaining a concept to someone helps you understand it better;
  - having a concept explained to you might provide you with different and more productive ways of understanding the concept;
  - explaining a program to someone else can help you find its bugs;
  - working with someone else helps you identify your own strengths and weaknesses.

So, we request that you take the collaborative activities seriously. They can be not only helpful to learning, but fun!—new friends that will make good partners in future courses and supporters along your way to a Berkeley degree. Sometimes it's difficult to pick a partner. We will provide you with some icebreaking questions (today's is "what's your favorite ice cream flavor?").
Code analyzing

First, introduce yourself to a labmate that you don’t already know, tell this person your favorite ice cream flavor, and find out his or her favorite. Then, working together but not using a computer, determine what each of the following programs prints. Keep track of how you figure out the output in each program. First program:

public class Test1 {
    public static void main (String [ ] args) {
        int x = 1;
        int y = 1;
        while (x < 5) {
            y = x - y;
            System.out.print (x + " + " + y + " ");
            x = x + 1;
        }
        System.out.println ( );
    }
}

Second program:

public class Test2 {
    public static void main (String [ ] args) {
        int x = 1;
        int y = 1;
        while (x < 5) {
            x = x + 1;
            y = y + x;
            System.out.print (x + " + " + y + " ");
            x = x + 1;
        }
        System.out.println ( );
    }
}

Third program:

public class Test3 {
    public static void main (String [ ] args) {
        int x = 1;
        int y = 1;
        while (x < 5) {
            if (y < 5) {
                x = x + 1;
                if (y < 3) {
                    x = x - 1;
                }
            }
            y = y + 2;
            System.out.print (x + " + " + y + " ");
            x = x + 1;
        }
        System.out.println ( );
    }
}

2.2.6 (Brainstorm) Reflection on your problem solving

The answers to the preceding exercises are

Program 1: 10 22 31 43
Program 2: 23 47
Program 3: 13 35 47

If you got them right, explain what you did, either personally or with your partner, that led to your success. If you got one or more wrong, explain your errors and how you might have avoided them. (Your name won’t appear on your answers, so don’t be embarrassed.)

2.2.7 (Display page) A jigsaw puzzle

A jigsaw puzzle

Working again with your lab mate, assemble the following parts plus as many right braces as necessary into a complete program that, when run, produces the output

a-b c-d

You may do this either on paper or on the computer. Here are the parts. (They’re also on page 20 of Head First Java. You may check your answer in the book when you finish.)

Part 1

if (x == 1) {
    System.out.print ("d");
    x = x - 1;
}

Part 2

if (x == 2) {
    System.out.print ("b c");
}

Part 3

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

Part 4

if (x > 2) {
    System.out.print ("a");
}

Part 5

int x = 3;
Part 6

```java
x = x - 1;
System.out.print('-');
```

Part 7

```java
while (x > 0) {
```

2.2.8 (Brainstorm) Reflection on your problem solving

Please answer all three of the following questions:

1. How did you go about solving the "jigsaw puzzle" of the previous step?
2. What caused you the most trouble?
3. How did working in partnership help?

2.2.9 (Display page) Another jigsaw puzzle

A jigsaw puzzle

For the remaining lab activities, go back to working alone. The file
`
~cs61b/code/TriangleDrawer.stuff`

contains a collection of statements. Some of them, together with some extra right braces, form the body of a `main` method that, when executed, will print the triangle

```
*
**
***
****
*****
******
*******
********
*********
**********
```

(Each line has one more asterisk than its predecessor; the number of asterisks in the last line is the value of the `SIZE` variable.) Start up Eclipse, create a new project named `TriangleDrawer` and a new class named `TriangleDrawer`, and copy and paste statements from the `TriangleDrawer.stuff` file into the `main` method along with some right braces. (You won't need all the statements.) Don't use any statements that aren't in the `TriangleDrawer.stuff` file. Hint: Declarations—for example, `int i Of int j`, don't all need to go at the start of a method. Many students encounter infinite loops in their first solutions to this problem. Eclipse seems not to control infinite loops all that well. To save yourself some trouble, run your program—and for that matter, any program where an infinite loop is possible—using the debugger ("Debug As ... Java Application").

2.2.10 (Display page) A fill-in-the-blanks problem

A fill-in-the-blanks problem

The program
`
~cs61b/code/DateConverter.java`

(shown below) is missing two assignment statements.

1. Create an Eclipse project named `DateConverter`, then use the File:Import command to import `DateConverter.java` into the newly created project.
2. Fill in the two missing assignment statements, replacing the lines marked with ***. You will either put both the statements in one of the indicated areas, or one statement in each.
3. Finally, test your code, keeping track of the test data you use. You provide test values by typing into the Console window (Eclipse prints your input in a different color from what it uses for the program's output). The program is an infinite loop that keeps reading test values that you type into the Console window; just kill it by clicking the red "terminate" box in the upper right corner of the console window segment.

Four new things about the code:

- The statement

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

  makes Java library methods involving input and output accessible to the program.

- The statement

  ```java
  BufferedReader keyboard = new BufferedReader (new InputStreamReader (System.in));
  ```

  sets up a `BufferedReader` object from which to retrieve values typed at the keyboard. This will be covered in more detail later in the course.

- The program segment

  ```java
  try {
      dayOfYear = Integer.parseInt (keyboard.readLine ( ));
  } catch (NumberFormatException e) {
      e.printStackTrace();
  } catch (IOException e) {
      e.printStackTrace();
  }
  ```

  reads the characters typed on a line when the program runs and converts the characters to an integer value. The `try` and `catch` reflect the possibility of the input failing. We'll deal with these details later in the course as well (chapter 11 in *Head First Java*).

- The operator "||" (two consecutive vertical bars) means "or".

Here's the code for `DateConverter.java`:

```java
import java.io.*;
public class DateConverter {
    public static void main (String [] args) {
        // This is the contents of a main method that reads a day number in 2004,
        // an integer between 1 and 366, and prints the date in month/day format.
        // The code is missing two assignment statements;
```
2.2.11 (Brainstorm) Reflection on your problem solving

How did you decide what the assignment statements should be and where in the program they should go?

2.2.12 (Brainstorm) Choice of test dayOfYear values

List the dayOfYear values you used to test your program, and explain why you think those values were sufficient to reveal all your bugs.

2.2.13 (Display page) Check digit verification

Check digit verification

Identification numbers are everywhere today: student IDs, driver's license numbers, credit card numbers, bank account numbers, Universal Product Codes (UPCs), and International Standard Book Numbers (ISBNs) are but a few examples. One simple security scheme for verifying the accuracy of an identification number is to have one or more of its digits be computed from the remaining digits. Create an Eclipse project named CheckDigit and import the program ~cs61b/code/CheckDigit.java (also provided below) into the project. Then supply the missing code that determines if the identification number in the variable id is legal by storing a true or false value in the variable isLegal. In a legal ID number, the rightmost digit is the last digit of the sum of all the other digits in the number. For example, 123456786 is legal since 6 is the last digit of 1+2+3+4+5+6+7+8 = 36, while 123456782 is not legal. The code you supply may declare extra variables; give them names that suggest their purpose in the computation.

CheckDigit.java

```java
public class CheckDigit {
    public static void main (String [ ] args) {
        int id = 123456786;
        boolean isLegal;
        // your missing code goes here
        if (!isLegal) {
            System.out.println (id + " is not legal");
        } else {
            System.out.println (id + " is legal");
        }
    }
}
```

2.3 Homework (3 steps)

2.3.1 (Display page) Reading plus discussion contributions

Reading plus discussion contributions

Finish the triangle-drawing, date conversion, and check digit programs if you didn't do so in lab, copy them from the Eclipse workspace to a top-level directory named day2, and submit the working versions. Also, read chapters 3, 4, and 9 (except pages 250-255) in Head First Java, along with the section "Boxes and arrows" (available online), for next lab. Finally, review and summarize the responses to the steps "Reflection on your problem solving" (all three of them) and "Choice of test dayOfYear values" from today, and contribute your summaries to the "brainstorm" activity in the next step. Provide a post and a response in the subsequent discussion activity.

2.3.2 (Brainstorm) Summaries of earlier steps

Summarize the responses to the "Reflection on your problem solving" (all three of them) and "Choice of test dayOfYear values" steps. Your name will appear on the summary, so do a good job.

2.3.3 (Discussion Forum) Programming skills and ways to acquire them

There are two components to this homework discussion.

1. First, post a list of skills that would help a student become good at the tasks you did in lab today (designing code from scratch, filling in missing code, analyzing code, inventing convincing test values, and collaborating with a partner).

2. Then, respond to the list of one of your classmates by suggesting ways that a student could best learn the skills in the list. Draw on personal experience where relevant.
### 3.2 Practice with objects (12 steps)

#### 3.2.1 (Display page) Using objects in Scheme

### Using objects in Scheme

In CS 61A, you used objects in Scheme. As in Java, you defined classes from which objects were instantiated. Each class contained one or more methods, things the objects can do; each class may also contain instance variables, that represent the object’s local state. Here is an example you may have seen in CS 61A.

```scheme
(define-class (counter)
  (instance-vars [myCount 0])
  (method (next)
    [set! myCount (+ myCount 1)]
    myCount)
)
```

This class represents a counter that keeps track of how many times its `next` method is called. Here’s how it might be used:

```scheme
> (define c1 (instantiate counter))
1
> (ask c1 'next)
2
> (define c2 (instantiate counter))
1
> (ask c1 'next)
3
```

Each call to `instantiate` builds and initializes an object. Each object has its own `myCount` variable in which it keeps track of how many times its `next` method has been called.

#### 3.2.2 (Display page) Comparing Scheme object use with Java object use

### Comparing Scheme object use with Java object use

You’ve seen several examples of setting up a class. So far our classes have only contained a `main` method:

```java
public class DateConverter {
  public static void main (String [ ] args) {
    ...
  }
}
```

More methods, and instance variables, would be provided at the same level as `main`, for example,

```java
public class Counter {
  int myCount = 0;
  void increment () {
    myCount = myCount + 1;
  }
  void print () {
    System.out.println ('count = ' + myCount);
  }
}
```

Compare with the Scheme version:

```scheme
(define-class (counter)
  (instance-vars [myCount 0])
  (method (increment)
    [set! myCount (+ myCount 1)])
  (method (print)
    [display myCount]
    [newline])
)
```

Each method header starts with a type (`void` in the examples above), continues with the name of the method, and concludes with the parameter list (empty in the examples above). Then come the statements for the method. We move on to the instantiation and use of an object. Here are some examples of operations in Scheme and their Java equivalents.

### Primitive and reference variables

Each variable in Java must have an associated type. We’ve seen variables of type `int`, which contain integer values between –2147483648 and 2147483647, and `boolean`, which contain `true` or `false`. Other primitive types include `char` (for character values) and `double` (for double-precision floating point values). Reference variables, in contrast to primitive variables, contain references to constructed objects. This distinction between primitive-valued variables and variables bound to more complicated values is the same as in Scheme. Just as a list value in Scheme is associated with a pointer, a Java reference is essentially a pointer to the associated object value. A reference is drawn as an arrow, just as a pointer in Scheme. The reference value `null` corresponds to the null pointer in Scheme. Some diagrams of initialized variables appear below.

<table>
<thead>
<tr>
<th>operation</th>
<th>Scheme</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>instantiation</td>
<td><code>(define c1 (instantiate counter))</code></td>
<td>Counter c1 = new Counter ()</td>
</tr>
<tr>
<td>method call</td>
<td><code>(ask c1 'next)</code></td>
<td>c1.next ()</td>
</tr>
<tr>
<td>instance variable access</td>
<td><code>(ask c1 'myCount)</code></td>
<td>c1.myCount</td>
</tr>
</tbody>
</table>

#### 3.2.3 (Display page) Primitive and reference variables

### Primitive and reference variables

Each variable in Java must have an associated type. We’ve seen variables of type `int`, which contain integer values between –2147483648 and 2147483647, and `boolean`, which contain `true` or `false`. Other primitive types include `char` (for character values) and `double` (for double-precision floating point values). Reference variables, in contrast to primitive variables, contain references to constructed objects. This distinction between primitive-valued variables and variables bound to more complicated values is the same as in Scheme. Just as a list value in Scheme is associated with a pointer, a Java reference is essentially a pointer to the associated object value. A reference is drawn as an arrow, just as a pointer in Scheme. The reference value `null` corresponds to the null pointer in Scheme. Some diagrams of initialized variables appear below.

<table>
<thead>
<tr>
<th>declaration</th>
<th>diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean isOdd;</td>
<td><img src="http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary" alt="Diagram" /></td>
</tr>
</tbody>
</table>
3.2.4 (Brainstorm) Drawing an int variable
Some students incorrectly draw the result of the code:

```
int k;
k = 3;
```
as follows:

Explain these students' misconception.

3.2.5 (Display page) The steps of initializing a reference variable

The steps of initializing a reference variable:

1. declaring the reference variable;
2. using the `new` operator to build an object and create a reference to the object; and
3. storing the reference in the variable.

Here's an example. The Java code:

```
Counter c1 = new Counter ( );
```
first declares the variable `c1` as something that can contain a reference to a `Counter` object. Then the `new` operator builds—an allocates memory for—a `Counter` object. (It also calls a constructor method, which will be discussed later in this lab.) The `new` operator returns a reference to the new object, which is then stored in the `c1` variable. The variable declaration (step 1) and the initialization (steps 2 and 3) may be done separately, for example:

```
Counter c1;
c1 = new Counter ( );
```

3.2.6 (Display page) Calling methods and accessing instance variables

A class's methods can access the class's instance variables and call its methods. In particular, the `main` method for a class is often used to test the class's methods. An instance variable is accessed with the expression `referenceVariable.instanceVariableName`. Similarly, a method is called with `referenceVariable.methodName(parameters)`. Here's the Java `Counter` class again, with method calls and instance variable accesses in the `main` method.

```
public class Counter {
    int myCount = 0;
    void increment ( ) {
        myCount = myCount + 1;
    }
    void print ( ) {
        System.out.println ("count = " + myCount);
    }
    public static void main (String [ ] args) {
        Counter c1 = new Counter ( );
        c1.increment ( );  // c1's myCount is now 1
        c1.increment ( );  // c1's myCount is now 2
        Counter c2 = new Counter ( );
        c2.increment ( );  // c2's myCount is now 1
        c2.increment ( );  // c2's myCount is now 2
        c1.myCount = 0;    // effectively resets the c1 counter
    }
}
```

```
You may be wondering: why don't the `increment` and `print` methods need to use the dot notation to access the `myCount` variable? This is because of the keyword `static` in the header of the `main` method. A static method may be called before any object gets created; in order to use an object, a static method must use `new` to build one. A nonstatic method (or instance variable) may only be called (or referenced) by using the `.` operator; it then may call the methods or use the instance variables of the object referenced on the left of the dot. That object has a name; it's called `this`. Thus the body of the `increment` method may also be written as

```
this.myCount = this.myCount + 1;
```
```

3.2.7 (Brainstorm) Interpreting an error message

Copy the code for the `Counter` class into Eclipse. Then change the first call to `increment` to

```
increment ( );
```
That is, remove the "c1." from the front of the call. Read the error message that Eclipse provides for this, and explain it in your own words.

3.2.8 (Self Test) Assignment statements

Consider a slightly different version of the main program:

```
Counter c1 = new Counter ( );
c1.increment ( ); // c1 now represents the value 1
Counter c2 = new Counter ( ); // c2 now represents the value 0
```

Indicate which of the box-and-pointer diagrams best represents the effect of the assignment statement.
2/11/10 9:19 AM

3.2.9 (Display page) An exercise with a Line class

An exercise with a Line class

The file ~cs61b/code/LineFW1.java (also listed below) contains a framework for a class representing a line segment as a pair of points \((x_1,y_1)\) and \((x_2,y_2)\). Each point is represented by a pair of integer instance variables. Create a project named lines, then import the LineFW1 program. Rename the class name or rename the file so that the names match as required by Java. Complete the framework by filling in the blanks in the printLength and printAngle methods, and by supplying code in the main method that matches the corresponding comments. Recall that the length of the line segment connecting points \((x_1,y_1)\) and \((x_2,y_2)\) is the square root of \((x_2-x_1)^2 + (y_2-y_1)^2\), and the angle is the arctangent of \(y_2-y_1\) and \(x_2-x_1\). Here’s the code. It uses static methods and the static variable \(\pi\) declared in the Math class.

```java
import java.awt.*;

public class Line1 {
    int x1, y1, x2, y2;
    void printLength () {
        double length = Math.sqrt ( ____ ) ;
        System.out.println ("Line length is ", length);
    }
    void printAngle () {
        double angleInDegrees = Math.atan2 ( ____ , ____ ) * 180.0 / Math.PI;
        System.out.println ("Angle is ", angleInDegrees + ", degrees");
    }

    public static void main(String[] args) {
        System.out.println ("testing Line1");
        // Set myLine to contain a reference to a new line object.
        // Initialize myLine’s x1 and y1 to the point (5, 10),
        // and initialize myLine’s x2 and y2 to the point (45, 40).
        // Print the line’s length, which should be 50.
        // Print the line’s angle, which should be around 36.87 degrees.
    }
}
```

3.2.10 (Display page) Another version of the Line class

Another version of the Line class

The file ~cs61b/code/LineFW2.java (also listed below) is like LineFW1.java except that it represents a line segment as a pair of points. It uses the Point class supplied in the java.awt class library. Complete the framework, rename either the file or the class, then test the resulting program. This requires two steps:

1. First access the online documentation at sun.com—the URL is http://java.sun.com/j2se/1.5.0/docs/api/—to find out the names of the Point instance variables. (You should probably bookmark this web site.)
2. Then fill in the blanks in the printLength and printAngle methods, and supply code in the main method that matches the corresponding comments, as you did in the preceding activity.

Here’s the code.

```java
import java.awt.*;

public class Line2 {
    Point p1, p2;
    void printLength () {
        double length = Math.sqrt ( ____ ) ;
        System.out.println ("Line length is ", length);
    }
    void printAngle () {
        double angleInDegrees = Math.atan2 ( ____ , ____ ) * 180.0 / Math.PI;
        System.out.println ("Angle is ", angleInDegrees + ", degrees");
    }

    public static void main(String[] args) {
        System.out.println ("testing Line2");
        // Set myLine to contain a reference to a new line object.
        // Initialize myLine’s p1 to the point (5, 10),
        // and initialize myLine’s p2 to the point (45, 40).
        // Print the line’s length, which should be 50.
        // Print the line’s angle, which should be around 36.87 degrees.
    }
}
```

3.2.11 (Self Test) More assignment statements

What gets printed by the following program? Figure out the answer without using the computer.

```java
import java.awt.*;

public class Test1 {
    public static void main (String [] args) {
        Point p1 = new Point ( );
        p1.x = 1;
        p1.y = 2;
        Point p2 = new Point ( );
        p2.x = 3;
        p2.y = 4;
        // now the fun begins
        p2.x = p1.y;
    }
}
```
p1 = p2;  
p2.x = p1.y;  
System.out.println (p1.x + " " + p1.y  
  + " " + p2.x + " " + p2.y);  
}
}

Consult with a classmate if your answer is incorrect.

3.2.12 (Brainstorm) Possible difficulties with boxes and arrows?
If you got the answer to the previous exercise wrong, explain why. If you got it right, explain a misconception that a student might have that would produce a wrong answer.

3.3 Work with typed methods and method parameters. (10 steps)

3.3.1 (Display page) Typed methods

Typed methods

Most of the methods we've seen don't return any value; they are useful only for their side effect. Not surprisingly, however, we can write methods that return values. The method header contains the type of value returned instead of the word "void"; the body of the method contains at least one return statement that specifies the value the method should return. As an example, here is another version of the Counter class seen earlier.

```
public class Counter {
    int myCount = 0;
    void increment () {
        myCount = myCount + 1;
    }
    int value () {
        return myCount;
    }
    void print () {
        System.out.println ("count = " + myCount);
    }
}
```

3.3.2 (Display page) Parameters and constructors

Parameters and constructors

Methods can also have parameters. Parameter names and types appear in the method header, in parentheses right after the method name, separated by commas. A common use for parameters is a constructor method, which by convention initializes the instance variables of an object so that they all contain valid values. The constructor's name is the same as the name of the class. It is called automatically when the new operator is used with a class name. Here's an example:

```
public class Line {
    Point endPoint1, endPoint2;
    public Line (int x1, int y1, int x2, int y2) {
        endPoint1 = new Point (x1, y1);
        endPoint2 = new Point (x2, y2);
    }
    ...
}
```

The Line constructor would be called as a result of executing the statement

```
// Create a line connecting points (1, 2) and (7, 41).
Line segment = new Line (1, 2, 7, 41);
```

When no constructor is supplied, Java provides one with no arguments. Examples from Head First Java and the lab activities have so far set up objects without constructors, then provided values for their instance variables via "setter" methods. Though this served the pedagogical goal of simplifying the introduction of methods, it is not good programming practice; the need for "setter" methods is actually rather rare. From now on, all our constructors will return having either initialized instance variables with legal reasonable values, or will signal some sort of error indicating that it was impossible to do so. A vocabulary note: When describing a nonstatic method, we will often refer to the associated object as "this object", that is, the object whose reference is contained in this.

3.3.3 (Display page) An example: a jar of M&Ms

An example: a jar of M&Ms

A friend of ours keeps a jar of M&Ms on her desk. She fills it up when it's almost empty; she and her co-workers make "withdrawals" throughout the day. Create a new project named MMs. Then model the M&M jar with a Java class named MMjar. It will have three methods:

- a one-argument constructor whose parameter says how many M&Ms the jar initially contains;
- a void method named add whose single parameter specifies the number of M&Ms with which to restock the jar; and
- an eat method, also void, whose single parameter specifies the number of M&Ms that someone wants to remove from the jar.

The eat method should check after decrementing the stash of M&Ms whether the supply is too low. If there are fewer than 100 M&Ms remaining in the jar after the given number are removed, the method should print a message warning of the impending shortage. Your MMjar class should not have a main method. Instead, you will use a tester class named MMjarTester to test it. MMjarTester should go in the same project and package as your MMjar class. (Choose “New class” in Eclipse to create it.) The main method in MMjarTester should initialize a jar of 500 M&Ms, then eat 300, then eat 99 more, then 1 more, then 1 more again (to produce the warning message). When you compile the MMjarTester class (or when Eclipse does it internally), the MMjar class that it uses is compiled automatically as well.

3.3.4 (Display page) Public vs. private

Public vs. private
So far we have not worried about access to the instance variables in a class. Often we would prefer to restrict access. For example, if the contents of the M&Ms jar could be decremented without going through the eat method, we would get no warning of the impending disaster of being out of M&Ms. The two keywords public and private define the accessibility of an instance variable or method. Public means accessible from outside the class; private means accessible only from within the class, in the confines of one’s own home so to speak. Almost all instance variables should be private. A central assumption underlying the design techniques we’ll discuss this semester is that the most likely change in a program is the re-implementation of a data type. Allowing direct access to an instance variable rather than requiring the use of access methods makes it much more difficult to change a representation. As noted above, we may also wish to restrict access to instance variables so we can log changes or perform consistency checks before modifying the variable values. The Point class, which has public x and y instance variables, is an exception. Point is mainly used to group two coordinates into a convenient package. In many applications, a pair of integer coordinates is the only reasonable way to implement a point, so information hiding isn’t necessary. The Point class also provides access methods getX and getY, however, just in case the programmer wants to insulate client code from the possibility of switching a point represented in Cartesian coordinates to, say, three-dimensional space or polar coordinates. Most of your methods will be public. However, you should try to limit the number of public methods you provide. In particular (contrary to what Head First Java implies), you don’t always need “getter” methods for your variables.

3.3.5 (Display page) Overloaded methods

Overloaded methods

Java allows more than one method in a class to have the same name. This is called overloading, and was not allowed in Scheme. The various methods with the same name all must have the same return type but different parameter lists. For example, some of you may have noticed overloaded methods in the java.awt.Point class:

- A zero-argument constructor that creates the point (0, 0);
- A two-argument constructor that, given integer coordinates x and y, creates the point (x, y);
- A one-argument constructor that, given a Point as argument, creates a copy of that Point;
- Three setLocation methods, two of which take pairs of coordinates as arguments, the other taking a single Point as argument.

3.3.6 (Display page) Bank account management

Bank account management

The next several exercises involve modifications to an Account class, which models a bank account. You’ll be submitting the Account class for homework (along with some other files), so work on it by yourself. Create an Account project and import the files ~cs61b/code/Account.java and ~cs61b/code/AccountTester.java into it. The Account class, like MMjar, allows deposits and withdrawals. Instead of warning about a balance that’s too low, however, it merely disallows a withdrawal request for more money than the account contains. Here’s the code:

```java
public class Account {
    // This class represents a bank account whose current balance is a nonnegative amount in US dollars.
    // Initialize an account with the given balance.
    public Account (int balance) { myBalance = balance; }
    // Add the given amount to the account.
    public void deposit (int amount) {
        if (amount < 0) {
            System.out.println("Cannot deposit a negative amount.");
        } else {
            myBalance = myBalance + amount;
        }
    }
    // Subtract the given amount from the account
    // if possible. If the amount would leave a negative balance, print an error message and
    // leave the balance unchanged.
    public void withdraw (int amount) {
        if (amount < 0) {
            System.out.println("Cannot withdraw a negative amount.");
        } else if (myBalance < amount) {
            System.out.println("Insufficient funds.");
        } else {
            myBalance = myBalance - amount;
        }
    }
    // Return the number of dollars in the account.
    public int balance () {
        return myBalance;
    }
    private int myBalance;
}
```

A few notes about the code:

- It doesn’t have a main method. You test it by using a tester class, provided for you in ~cs61b/code/AccountTester.java, similar to what you provided for MMjar.
- The code follows the convention of prefixing the word “my” onto instance variables, to reinforce the notion that each object has its own copy. This convention also deals with the difficulty of naming closely related variables. For instance, the argument to the constructor and the instance variable each represents a balance, but they can’t both be named “balance”. An alternative would be to name the instance variable “balance” and the constructor argument “initBalance”.
- The instance variable is listed after the methods. Defenders of this practice argue that a prospective user of the class should not be concerned with or distracted by the class’s
"private parts". On the other hand, someone reading the code will have to know what the instance variables are to make any sense of the methods, so for that purpose it's better to have them up front.

3.3.7 [Display page] Account management: modifying withdrawal behavior

Modifying withdrawal behavior

The withdraw method is currently defined as a void method. Modify it to return a boolean, true if the withdrawal succeeds (along with actually doing the withdrawal) and false if it fails. Also add tests of this feature to AccountTester.java.

3.3.8 [Display page] Account management: merging accounts

Merging accounts

Define a merge method on accounts that takes the balance of the argument account, adds it to the current balance of this account, and sets the argument's balance to zero. Here is a skeleton for the new method:

```java
public void merge (Account anotherAcct) {
    // put something here
}
```

Add appropriate tests to AccountTester.java. The code for this method demonstrates the limits of private access. Inside the merge method, you have access to two objects, one named this and the other named anotherAcct. Moreover, you can also access both your own myBalance variable and that of the argument account! (This surprises some students who, reasoning from real life, think that an object should have private information that not even another object of the same class can access.)

3.3.9 [Display page] Account management: overdraft protection

Overdraft protection

A convenient feature of some bank accounts is overdraft protection; rather than bouncing a check when the balance would go negative, the bank would deduct the necessary funds from a second account. One might imagine such a setup for a student account, provided the student's parents are willing to cover any overdrafts (!). Another use is to have a checking account that is tied to a saving account, where the savings account covers overdrafts on the checking account. Implement and test overdraft protection for Account objects, by completing the following steps.

1. Add a parentAccount instance variable to the Account class; this is the account that will provide the overdraft protection, and it may have overdraft protection of its own.
2. Add a two-argument constructor. The first argument will be the initial balance as in the existing code. The second argument will be an Accounts reference with which to initialize the instance variable you defined in step 1.
3. In the one-argument constructor, set the parent account to null.
4. Modify the withdraw method so that, if the requested withdrawal can't be covered by this account, the difference is withdrawn from the parent account. This may trigger overdraft protection for the parent account, and then its parent, and so on! You are not allowed to assume a limit on the number of accounts connected in this way. If the account doesn't have a parent and it can't cover the withdrawal, the withdraw method should merely print an error message as before and not change any account balances. Here's an example of the desired behavior, with the Account Object kathy providing overdraft protection for the Account object megan.

<table>
<thead>
<tr>
<th>kathy balance</th>
<th>megan balance</th>
<th>attempted withdrawal from megan</th>
<th>desired result</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>100</td>
<td>50</td>
<td>megan has 50, kathy has 500</td>
</tr>
<tr>
<td>500</td>
<td>100</td>
<td>200</td>
<td>megan has 0, kathy has 400</td>
</tr>
<tr>
<td>500</td>
<td>100</td>
<td>700</td>
<td>return false without changing either balance</td>
</tr>
</tbody>
</table>

Use recursion, not a loop, to implement this feature.

5. Add tests to AccountTester.java. Sufficient to exercise all cases in the modified withdraw method.

6. Copy the updated Account.java and AccountTester.java (including the merge method) to a directory named cs61b. (That will be your homework submission directory. You will also need to include the results of the next activity, "Debug some recursive code", in that directory.)

3.3.10 [Brainstorm] Account management: merging revisited

One proposed solution for merging accounts is the following:

```java
public void merge (Account otherAccount) {
    this.myBalance = this.myBalance + otherAccount.myBalance;
    otherAccount = new Account (0);
}
```

This doesn't work. Explain why not.

3.4 Debug some recursive code (3 steps)

3.4.1 [Display page] contains1MoreThan

contains1MoreThan

Consider the following code, online in ~cs61b/code/Debug.java. It uses Java's String class, whose methods are described in http://java.sun.com/j2se/1.5.0/docs/api/. There is a bug in the contains1MoreThan method; you'll work with it (on your own) for the rest of lab, and submit what you've found out for homework.

```java
public class Debug {
    String myString;
    public Debug (String s) {
```

http://spring08.ucwise.org/builder/builderPortal.php?BUILD MENU=curriculumSummary
myString = s;

// Return true when myString is the result of inserting
// exactly one character into s, and return false otherwise.
public boolean contains1MoreThan (String s) {
    if (myString.length ( ) == 0) {
        return false;
    } else if (s.length ( ) == 0) {
        return myString.length ( ) == 1;
    } else if (myString.charAt(0) == s.charAt(0)) {
        Debug remainder = new Debug (myString.substring(1));
        return remainder.contains1MoreThan (s.substring(1));
    } else {
        return myString.substring(1) == s;
    }
}

public static void main (String [] args) {
    check ("abc", "def");  // should be false
    check ("abc2", "abc"); // should be true
}

public static void check (String s1, String s2) {
    Debug d = new Debug (s1);
    if (d.contains1MoreThan (s2)) {
        System.out.println (s1 + " is the result of adding a single character to " + s2);
    } else {
        System.out.println (s1 + " is not the result of adding a single character to " + s2);
    }
}

3.4.2 (Display page) When it works, and when it doesn't

When it works, and when it doesn't

Copy the Debug class into Eclipse, and experiment with it by adding more calls to check. (Don't delete any of the calls you try; we'd like to see them in the file you submit for homework.) Determine answers to the following questions, either by experiment or by analyzing the code. Put these answers into a file named bug.info.

1. Describe all pairs of arguments to check for which contains1MoreThan correctly returns true.
2. Describe all pairs of arguments to check for which contains1MoreThan correctly returns false.
3. Describe all pairs of arguments to check for which contains1MoreThan incorrectly returns true, that is, when the first string argument to check is not the result of inserting exactly one character into the second.
4. Describe all pairs of arguments to check for which contains1MoreThan incorrectly returns false, that is, when the first string argument to check is the result of inserting exactly one character into the second.
5. Describe all pairs of arguments to check for which contains1MoreThan crashes.

Note that the answer for any category may be "all pairs of strings" or "no pairs of strings".

3.4.3 (Display page) Explanation of the bug

Explanation of the bug

Determine what's wrong with the contains1MoreThan method, and how you figured it out. Add this information to your file bug.info.

3.5 Homework(1 step)

3.5.1 (Display page) Coding homework + readings

Reading + discussion

Finish today's lab activities if you haven't already. In particular, submit the following as assignment "day3":

- your Account.java and AccountTester.java that provide overdraft protection and account merging;
- your Debug.java program with whatever extra calls to check you added while analyzing it;
- your file bug.info.

We'll be covering arrays and testing in the next lab. Arrays have been introduced in Head First Java already; section A.8 in Objects, Abstraction, Data Structures and Design Using Java provides a good summary. Also read chapters 1 and 2 in Practical Unit Testing.

4 Practice with objects and arrays; testing 2008-1-31 ~ 2008-2-01 (6 activities)

4.2 Practice with arrays. (5 steps)

4.2.1 (Display page) Array definition and use

Array definition and use

An array is an indexed sequence of elements, all the same type. Real-life examples of arrays include the following:

- post office boxes;
- book pages;
- egg cartons;
- chess/checkerboards.

Index values go from 0 to the length of the array, minus 1. To access an array element, we first give the name of the array, and then supply an index expression for the element we want in square brackets, for example:

```
values[k+1]
```

If the value of the index expression is negative or greater than or equal to the length of the array, an exception results. We declare an array variable by giving the type of its elements, a pair of square brackets, and the variable name, for example:

```
myString = s;
```
int [] values;

Note that we don't specify the length of the array in its declaration. Arrays are basically objects with some special syntax. To build an array, we use the new operator as we do with objects; the argument to new is the type of the array, which includes the length. For example, the statement

```java
values = new int [7];
```

stores a reference to a 7-element integer array in values. This initializes the array variable itself, but not its elements; they can be initialized with assignment statements. An array has an instance variable named length that stores the number of elements the array can hold. For the values array just defined, values.length is 7. The length variable can't be changed; once we create an array of a given length, we can't shrink or expand that array.

4.2.2 (Display page) Arrays in Scheme

Arrays in Scheme

Arrays in Scheme are called vectors. You should have encountered them in CS 61A. The table below lists Scheme counterparts for the array operations just described.

<table>
<thead>
<tr>
<th>Java</th>
<th>Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>values = new int [7]</td>
<td>(make-vector 7)</td>
</tr>
<tr>
<td>values.length</td>
<td>(vector-length)</td>
</tr>
<tr>
<td>values[k] = 29;</td>
<td>(vector-set! values k 29)</td>
</tr>
<tr>
<td>values[k], used elsewhere</td>
<td>(vector-ref values k)</td>
</tr>
</tbody>
</table>

4.3 (Display page) Version 3 of the Line class

Version 3 of the Line class

The file ~cs61b/code/LineFW3.java contains a version of the Line class that represents a line segment as an array of four integers: the x and y values of one of the endpoints (in elements 0 and 1), and the x and y values of the other endpoint (in elements 2 and 3). As before, complete and test the framework.

4.3.1 (Display page) Playing with references to arrays

What gets printed by each call to System.out.println in the code below?

```java
public class Test {
    public static void main (String [] args) {
        int [] vals1, vals2, vals3;
        vals1 = new int[2];
        vals1[0] = 1;
        vals1[1] = 3;
        vals2 = new int[2];
        vals2[0] = vals1[1];
        vals2[1] = vals1[0];
        vals1[0] = 5;
        System.out.println (vals2[0] + " " + vals2[1]); // 1
        vals3 = vals1;
        vals3[1] = 27;
        System.out.println (vals3[0] + " " + vals3[1]); // 2
        vals3[0] = 25;
        System.out.println (vals3[0] + " " + vals3[1]); // 3
    }
}
```

4.3.2 (Display page) Playing with references in arrays of objects

Playing with references to arrays

What gets printed by each call to System.out.println in the code below?

```java
public class Test {
    public static void main (String [] args) {
        int [] vals1, vals2, vals3;
        vals1 = new int[2];
        vals1[0] = 1;
        vals1[1] = 3;
        vals2 = new int[2];
        vals2[0] = vals1[1];
        vals2[1] = vals1[0];
        vals1[0] = 5;
        System.out.println (vals2[0] + " " + vals2[1]); // 1
        vals3 = vals1;
        vals3[1] = 27;
        System.out.println (vals3[0] + " " + vals3[1]); // 2
        vals3[0] = 25;
        System.out.println (vals3[0] + " " + vals3[1]); // 3
    }
}
```

http://spring08.ucwise.org/builder/builderPortal.php?BUILDERS_menu=curriculumSummary
Playing with references in arrays of objects

What is printed by the program below?

```java
import java.awt.*;

public class Test {
    public static void main (String [] args) {
        Point [] line1, line2;
        line1 = new Point [2];
        line2 = new Point [2];
        line1[0] = new Point ();
        line1[0].x = 1;
        line1[0].y = 3;
        line1[1] = line1[1];
        line1[1].x = 7;
        line1[1].y = 9;
        line2[0] = line1[1];
        line2[1] = line1[0];
        line1[0].x = 11;
        line1[1] = line1[0];
        System.out.println (line2[0].x + " " + line2[0].y
           + " " + line2[1].x + " " + line2[1].y);
    }
}
```

4.3.4 (Display page) Argument passing

Argument passing

Arguments in Java are all passed by value. That means that each argument is copied into the corresponding parameter in the method being called. When the parameter is a reference, the effect is the same as an assignment statement involving two references. In box-and-pointer terms, the copy results in two arrows pointing to the same box. What this means in practice is an argument of a primitive type can't be changed in a method to which it is passed, since the method is working with a copy. A reference argument can't be changed either, but variables in the referenced object can be changed.

4.3.5 (Display page) Some exercises with argument passing

Some exercises with argument passing

What output is produced by each call to System.out.println in the program below?

```java
public class SwapTest {
    private int k1, k2;

    public SwapTest (int a, int b) {
        k1 = a;
        k2 = b;
    }

    public static void swap (SwapTest ref1, SwapTest ref2) {
        SwapTest temp = ref1;
        ref1 = ref2;
        ref2 = temp;
    }

    public static void swap (int a, int b) {
        int temp = a;
        a = b;
        b = temp;
    }

    public static void swapks (SwapTest ref) {
        int temp = ref.k1;
        ref.k1 = ref.k2;
        ref.k2 = temp;
    }

    public static void main (String [] args) {
        SwapTest test1 = new SwapTest (1, 3);
        SwapTest test2 = new SwapTest (5, 7);
        swap (test1, test2);
        System.out.println (test1.k1 + " " + test1.k2 + " " + test2.k1 + " " + test2.k2);
        swap (test1.k1, test1.k2);
        System.out.println (test1.k1 + " " + test1.k2 + " " + test2.k1 + " " + test2.k2);
        swapks (test2);
        System.out.println (test1.k1 + " " + test1.k2 + " " + test2.k1 + " " + test2.k2);
    }
}
```

4.3.6 (Brainstorm) Tips for dealing with references

Describe some errors that a student might make in working through the previous three exercises, and suggest some tips for avoiding those errors.

4.3.7 (Display page) Checking your understanding

Checking your understanding

Compiled versions of the three programs you just considered are in the -cs61b/code directory: LabTest1.class, LabTest2.class, and SwapTest.class. To check your answers, run each program in the UNIX shell using the java command:

```
java -classpath -cs61b/code LabTest1
java -classpath -cs61b/code LabTest2
java -classpath -cs61b/code SwapTest
```

4.3.8 (Display page) The toString method

The toString method

The printing methods System.out.print and System.out.println both take an argument of type String (one of Java's built-in classes). (We have seen the String concatenation operator + used to build a String argument to be printed.) The String class knows how to do "the right thing" with
values of primitive types; for example, it converts an integer to the corresponding sequence of digit characters. However, given a reference as argument, it will produce the String representation of the memory address represented by the reference. For example: the statement

```
SwapTest test1 = new SwapTest (1, 3);
System.out.println (test1);
```

produces the output
```
SwapTest@47b480
```

We can fix this behavior by supplying our own conversion method named toString. If we define a toString method as follows:

```
public String toString ( ) {
   ...
}
```

System.out.print and System.out.println will call it whenever they receive a reference of the corresponding type as argument. (This is an instance of using inheritance to override behavior in a parent class. In Java, all classes inherit from the Object class, whose toString produces the cryptic result mentioned above. We'll investigate the details of inheritance in a couple of weeks.)

4.3.9 (Display page) The equals method

The equals method

Another useful method is equals, conventionally the method to determine whether two objects contain the same information. (Compare this with the == operator, which determines whether its operands reference the same object.) Just as System.out.print and System.out.println call an object's toString method to determine its printable representation, there are a variety of situations where an object's equals method is called as a result of some other processing. Again because of inheritance, the header to the equals method has to be defined as follows to override the Object class's equals method:

```
public boolean equals (Object obj) {
   ...
}
```

For right now, the easiest way to code equals is to use toString:

```
public boolean equals (Object obj) {
   return this.toString ( ).equals (obj.toString ( ));
}
```

4.3.10 (Display page) Explanation of toString and equals

Explanation of toString and equals

In your own words, explain toString and equals to your partner in the context of the Counter class we worked with in an earlier lab. (It's online in ~/cs61b/code/Counter.java.) Once you have these worked out, go back to working alone.

4.4 Work with JUnit.

4.4.1 (Display page) Background

JUnit

JUnit is a testing framework that's integrated into the Eclipse programming environment. (It's introduced in chapter 2 of Pragmatic Unit Testing.) It provides a convenient way to structure and evaluate your program tests.

To set up the testing of the class you're currently working on in Eclipse, select New JUnit Test Case from the File menu. When you do this, Eclipse will ask you about adding the JUnit library to your build path; answer yes to this. It will then set up a test class whose name is the name of your class with "Test" appended.

Eclipse then presents you with a stub for your test class. You should now start defining void methods whose names start with "test" and describe what is being tested. They will call a number of assertion methods provided in JUnit, the most useful of which are the following.

```
void assertTrue (boolean condition);
void assertTrue (String errmsg, boolean condition);
void assertNull (Object obj);
void assertNull (String errmsg, Object obj);
void assertNotNull (Object obj);
void assertNotNull (String errmsg, Object obj);
void assertEquals (Object expected, Object actual);
void assertEquals (String errmsg, Object expected, Object actual);
```

```
assertEquals succeeds when expected.equals(actual). Otherwise it prints the message (if supplied).
assertDoes also does the right thing for primitive types; for example, you can supply two int values as arguments.
```

Perhaps you have been accustomed to testing your code using print statements. You don't do that here; the various test... methods you write will communicate their results to you by the success or failure of the assertions. Once you type in your test methods, run your test class. JUnit will give you a list of the assertions that fail, or a green bar if they all pass.

4.4.2 (Display page) Running JUnit

Running JUnit

The Counter class mentioned earlier is in the file ~/cs61b/code/Counter.java. The contents of a
4.5.2 (Display page) Testing principle 1

We think that test-driven development will reduce the tedium, and we'll address the difficulty in the next few activities.

4.4.4 (Display page) Testing a bank account class

The technique of test-driven development involves designing test cases for program features before designing the code that implements those features. We will do a bit of that here. First go to Counter.java. Double-click the word "Counter" in the class header; it should now be selected. Select "Rename" from the Refactor menu, and type "ModNCounter" in the dialog box that appears. The effect of this change is to change the name of the Counter class to ModNCounter, not only in Counter.java but also in CounterTest.java. "Mod N" counters count up to a specified amount (the "N"), and then cycle back to zero. A mod N counter has an extra instance variable—a good name for it is myN—and is initialized with a 1-argument constructor whose argument is the intended myN value. Thus the code

```java
ModNCounter c = new ModNCounter (2);
System.out.println (c.value ( ));
c.increment ( );
assertTrue (c.value()  == 2);
c.increment ( );
assertTrue (c.value() == 2);
```

will print 0, then 1, then 0. Now for the test-driven development. Go to CounterTest.java, supply an argument for each of the constructor calls, and add code in testIncrement that checks (using assertions) that wraparound of the cycling counter works correctly. Go to Counter.java and add an argument to the constructor. Both files should now be free of compile-time errors. Don't make any other changes in Counter.java. Run CounterTest as a JUnit test case. All tests should pass except the wraparound check. Now go fix the increment method and run the JUnit test again. If all goes well, your updated code should pass all the tests. So you've done some test-driven development. First, you supplied test cases in the CounterTest class. Then you changed as little code as possible in ModNCounter to remove compile-time errors, then you provided the code to pass the tests.

4.4.3 (Display page) A bit of test-driven development

More testing

From the standpoint of test-driven development, this activity is backward. It starts with an already implemented BankAccount class, and then designs the testing code. However, we hope that the practice with JUnit will be beneficial. The file ~/cs61b/code/BankAccount.java is a copy of the Account class you worked with earlier. Create a project and import BankAccount.java. Then create a new JUnit test case with the following methods:

- testInit should check that the balance of a newly created account is the amount provided to the constructor.
- testInvalidArgs should make sure that the result of supplying a negative number to deposit and withdraw doesn't change the account's balance.
- testOverdraft should make sure that an attempt to withdraw more than the account contains doesn't change the account's balance.
- testDeposit should make sure that the account balance reflects the result of a legal call to deposit.
- testWithdraw should make sure that the account balance reflects the result of a legal call to withdraw.

Run the test cases until they all succeed. You'll be submitting your JUnit test file for homework.

4.4.4 (Display page) Testing a bank account class

More testing

From the standpoint of test-driven development, this activity is backward. It starts with an already implemented BankAccount class, and then designs the testing code. However, we hope that the practice with JUnit will be beneficial. The file ~/cs61b/code/BankAccount.java is a copy of the Account class you worked with earlier. Create a project and import BankAccount.java. Then create a new JUnit test case with the following methods:

- testInit should check that the balance of a newly created account is the amount provided to the constructor.
- testInvalidArgs should make sure that the result of supplying a negative number to deposit and withdraw doesn't change the account's balance.
- testOverdraft should make sure that an attempt to withdraw more than the account contains doesn't change the account's balance.
- testDeposit should make sure that the account balance reflects the result of a legal call to deposit.
- testWithdraw should make sure that the account balance reflects the result of a legal call to withdraw.

Run the test cases until they all succeed. You'll be submitting your JUnit test file for homework.

4.5 Think about testing in general. (12 steps)

4.5.1 (Display page) Introduction

Testing is regarded by some programmers as tedious, and by almost all programmers as difficult. We think that test-driven development will reduce the tedium, and we'll address the difficulty in the next few activities.

4.5.2 (Display page) Testing principle 1

The most obvious principle of program testing is that test values must "exercise" every statement in the program, since any statement that's not tested may contain a bug. Our tests of the BankAccount
class basically did that, covering the three cases in withdraw and the two cases in deposit. As in the testing of BankAccount, exercising all statements may require more than one run of a program. Recall the leap year tester from an earlier exercise:

```java
if (year % 400 == 0) {
    System.out.println("leap year");
} else if (year % 100 == 0) {
    System.out.println("not leap year");
} else if (year % 4 == 0) {
    System.out.println("leap year");
} else {
    System.out.println("not leap year");
}
```

The code contains four cases, exactly one of which is executed for any particular value of year. Thus we must test this code with at least one year value per case, so at least four values of year are needed for testing:

- a year that's divisible by 400;
- a year that's divisible by 100 but not by 400;
- a year that's divisible by 4 but not by 100;
- a year that's not divisible by 4.

Principle 1 by itself is insufficient, however. Here's an example:

```java
isLeapYear = false;
if (year % 4 == 0) {
    isLeapYear = true;
}
if (year % 100 == 0) {
    isLeapYear = false;
}
if (year % 400 == 0) {
    isLeapYear = true;
}
```

A year value of 2000 causes all the statements in this program segment to be executed.

### 4.5.3 (Brainstorm) An undetected bug

Here's the code just described:

```java
isLeapYear = false;
if (year % 4 == 0) {
    isLeapYear = true;
}
if (year % 100 == 0) {
    isLeapYear = false;
}
if (year % 400 == 0) {
    isLeapYear = true;
}
```

Describe a small change that would not affect its behavior for a year value of 400 but would produce incorrect behavior for other year values.

### 4.5.4 (Brainstorm) Reasons the bug might be undetected

Here's the code again.

```java
isLeapYear = false;
if (year % 4 == 0) {
    isLeapYear = true;
}
if (year % 100 == 0) {
    isLeapYear = false;
}
if (year % 400 == 0) {
    isLeapYear = true;
}
```

Explain why a single test value that exercises every statement in this code is insufficient to reveal all bugs in the code.

### 4.5.5 (Display page) Testing principle 2

**Testing principle 2**

An elaboration of principle 1 addresses the need to test various paths through the program. Here are examples of path counting in a program segment with two top-level statements:

- There is just one path through two assignment statements.
- There are two paths through an assignment statement followed by an if: the path corresponding to a successful test, and the path corresponding to an unsuccessful test.
- Now consider two consecutive if statements:

```java
if (...) {
    ...
} else if (...) {
    ...
}
```

There are two possibilities for each test, success or failure. Thus there are four paths through the two statements, corresponding to the four possibilities

- success, success
- success, failure
- failure, success
- failure, failure

This explains why one test was insufficient to exercise all the code in

```java
isLeapYear = false;
if (year % 4 == 0) {
    isLeapYear = true;
}
if (year % 100 == 0) {
    isLeapYear = false;
}
if (year % 400 == 0) {
    isLeapYear = true;
}
```
isLeapYear = true;
}

From the previous discussion, we see a need for eight tests, corresponding to the eight paths through the three tests. They are listed below. We form the entire table, then remove the tests that are logically impossible:

- `year % 4 == 0, year % 100 == 0, and year % 400 == 0` (which just means that `year % 400 == 0`)
- `year % 4 == 0, year % 100 == 0, and year % 400 == 0` (not possible)
- `year % 4 == 0, year % 100 == 0, and year % 400 == 0` (not possible)
- `year % 4 == 0, year % 100 == 0, and year % 400 == 0` (not possible)
- `year % 4 == 0, year % 100 == 0, and year % 400 == 0` (not possible)
- `year % 4 == 0, year % 100 == 0, and year % 400 != 0` (equivalently, `year % 4 != 0`)

This leaves the four tests we needed for the earlier version.

4.5.6 (Brainstorm) Resolving an inconsistency

Given below is code from an earlier quiz.

```java
if (a == 0) {
    if (b == 0) {
        if (c == 0) {
            good = false;
        } else {
            good = true;
        }
    } else if (c == 0) {
        good = true;
    } else {
        good = false;
    }
} else if (b != 0) {
    good = false;
} else if (c == 0) {
    good = true;
} else {
    good = false;
}
```

There are only seven assignment statements. Was a case missed? Explain why or why not.

4.5.7 (Brainstorm) Another application of principle 2

Here is part of the solution framework for the `DateConverter` program in an earlier exercise.

```java
int month, dateInMonth, daysInMonth;
month = 1;
daysInMonth = 31;
while (dayOfYear > daysInMonth) {
    // *** Here is one possible place to put assignment statements.
    if (month == 2) {
        daysInMonth = 29;
    } else if (month == 4 || month == 6 || month == 9 || month == 11) {
        daysInMonth = 30;
    } else {
        daysInMonth = 31;
    }
    // *** Here is another possible place to put assignment statements.
}
dateInMonth = dayOfYear;
System.out.println (month + "/" + dateInMonth);
```

A CS 61B student observes that a test date in December causes `month` to take on all values between 1 and 12, and therefore results in all cases of the `if ... else` being executed. Describe a small change to the code that would not affect the behavior for a date in December but would introduce a bug for some other case.

4.5.8 (Display page) Principle 3 for testing loops

A principle for testing loops

A loop can vastly increase the number of logical paths through the code, making it impractical to test all paths. Here are some guidelines for testing loops, drawn from *Program Development in Java* by Barbara Liskov and John Guttag, a book used in previous CS 61B offerings.

- For loops with a fixed amount of iteration, we use two iterations. We choose to go through the loop twice rather than once because failing to reinitialize after the first time through a loop is a common programming error. We also make certain to include among our tests all possible ways to terminate the loop.
- For loops with a variable amount of iteration, we include zero, one, and two iterations, and in addition, we include test cases for all possible ways to terminate the loop. [The zero iteration case] is another situation that is likely to be a source of program error.

Liskov and Guttag go on to say:

This approximation to path-complete testing is, of course, far from fail-safe. Like engineers' induction "One, two, three—that's good enough for me," it frequently uncovers errors but offers no guarantees.

4.5.9 (Display page) Black-box testing

Black-box testing

All the test principles discussed so far focused on testing features of the code. Since they assume that we can see into the program, these techniques are collectively referred to as glass-box testing. Another testing approach is called black-box testing. It involves generating test cases based on the problem specification, not on the code itself. There are several big advantages of this approach:

- The test generation is not biased by knowledge of the code. For instance, a program author might mistakenly conclude that a given situation is logically impossible and fail to include tests for that situation; a black-box tester would be less likely to fall into this trap.
• Since black-box tests are generated from the problem specification, they can be used without change when the program implementation is modified.
• The results of a black-box test should make sense to someone unfamiliar with the code.
• Black-box tests can be easily designed before the program is written.

Black-box testing goes hand in hand with test-driven development.

4.5.10 (Display page) Principles for black-box testing

Principles for black-box testing

In black-box testing as in glass-box testing, we try to test all possibilities of the specification. These include typical as well as boundary cases, which represent situations that are extreme in some way, e.g. where a value is as large or as small as possible. There are often a variety of features whose "boundaries" should be considered. For example, in the DateConverter program, boundary cases would include not only dates in the first and last months of the year, but also the first and last dates of each month.

4.5.11 (Brainstorm) Devising boundary cases

Consider the problem of printing the days of a month in calendar format, given the number of days in the month and an integer that says what day the month starts on. For instance, given 30 as the number of days in the month and 3 (Wednesday) as the starting day, the output should be:

```
1  2  3  4
5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
```

Devise as many categories of boundary cases as you can.

4.5.12 (Display page) Some parting advice (for now)

Some parting advice (for now)

We'll encounter many more testing techniques later in the course. Here are some last bits of advice for now. Write tests as if you were testing your worst enemy's code. You're generally too familiar with your own code and might read a given line as containing what you meant to write rather than what you did write. Also, programmers generally don't want to find bugs in their own code. Test your program with values that are as simple as possible. If the program is supposed to work for a set of 1000 data values, make sure it works on a set of 3 values first. Wrapping a loop around your code may allow you to test it with multiple values in a single run. Make sure you know how your program is supposed to behave on a given set of test data. Often lazy programmers try a test and either don't pay attention to the output or just scan through it thinking that it "looks right". Often such a programmer is embarrassed later to note that he or she computed a product cost that's greater than the national debt or a quantity that's greater than the number of atoms in the universe. Examples of careless selection of test data may have come up in your testing of the DateConverter program. (Refer to the earlier brainstorm activity to check this.) Here's how we would have tested that program:

• We start with dates in January: 1 and 31. (These values give us as much or more information as values between 2 and 30.)
• We continue with the corresponding dates in February, computing the expected answers from January's dates. February 1 is just January 31 + 1 = 32; February 29 is 32 + 28 more days = 60.
• We design March's test values from February's, April's from March's, and so on through December. December 31 should be day number 366, so that provides a check on our arithmetic.

4.6 Homework (3 steps)

4.6.1 (Display page) A measurement class

A measurement class

Homework will involve testing and coding a class named Measurement to represent feet+inches measurements. The class will have several methods:

• One constructor will take no arguments and initialize this object to be a measurement of 0 feet, 0 inches.
• Another constructor takes a number of feet as its single argument, using 0 as the number of inches.
• A third constructor takes the number of feet in the measurement and the number of inches as arguments (in that order), and does the appropriate initialization.
• plus and minus each take a Measurement as argument and do the appropriate arithmetic, returning a new object that represents the indicated sum or difference. (i.e. this object doesn't change.) You may assume that the argument measurement will always be smaller than this object in a call to minus.
• multiple takes a nonnegative integer argument n, and returns a new object that represents the result of multiplying this object's measurement by n. For example, if this object represents a measurement of 7 inches,

        multiple (3)

should return an object that represents 1 foot, 9 inches.
• toString should return the string representation of this object in the form

        f"i"

that is, a number of feet followed by a single quote followed by a number of inches less than 12 followed by a double quote (with no blanks).

You need not error-check arguments to constructors. You may also choose any internal representation for a measurement; the only constraint is what toString must return.

4.6.2 (Display page) Reading, collaboration, programming

Reading, collaboration on a test suite, programming

For next lab, read chapter 5 in Head First Java except for page 116. For those of you who have been
wondering about the term "static", pages 273-287 and 292-293 in chapter 10 may prove useful; you will need to understand this material by the first exam. Also read sections 3.1 through 3.3 and chapters 4 and 5 in Pragmatic Unit Testing. After reading the specification of the Measurement class just described, design a thorough set of tests that will provide as much evidence as possible of the correctness of the various methods. Post your test suite design to the Brainstorm activity in the next step. (Post only English, not code. Your name will appear with your post, and you won't get to change your submission, so do a good job.) For programming homework, you are to complete the following:

- a JUnit test suite for the Account class you turned in for the previous lab;
- a thorough JUnit test suite for the Measurement class; and
- the code for the Measurement class.

Submit the three Java files AccountTest.java, MeasurementTest.java, and Measurement.java in the directory day4.

4.6.3 (Brainstorm) Tests of the Measurement class
Describe your test suite for the Measurement class. You don't get to edit your submission, and your name will appear with it, so do a good job.

5 More practice with arrays; collection classes
2008-2-05 ~ 2008-2-06 (6 activities)

5.2 Here is some background on for statements (5 steps)

The for statement

The for statement provides another way in Java to repeat a given program segment. It starts with the word "for," continues with loop information inside parentheses, and ends with the segment to be repeated—the loop body—enclosed in curly braces. Loop information consists of initializations, followed by a semicolon, followed by a test, followed by a semicolon, followed by increments. Any of these may be omitted. If there are more than one initialization or increment, they’re separated by commas. The test succeeds when the loop should continue. Loop execution proceeds as follows. First, the initializations are performed. Then the test is evaluated; if it's false, the loop is finished and execution continues with the following statement. If the test is true, the loop body is executed. Then the increments are performed. The test is evaluated again; a failed test halts the loop, while a successful test results in another execution of the loop body. Local variables may be declared in the initialization section of the loop.

Here are several equivalent uses of loops to compute n factorial (the product of all the positive integers up through n).

```java
for (int k=n, product=1; k>0; k=k-1) {
    product = product * k;
}
```

```java
product = 1;
for (int k=n; k>0; k=k-1) { // product initialized outside loop
    product = product * k;
}
```

```java
product = 1;
for (int k=n; k>0; ) { // test performed inside loop
    product = product * k;
    k=k-1;
}
```

```java
product = 1;
int k;
while (k>0) { // the while equivalent
    product = product * k;
    k=k-1;
}
```

The for loop is basically a repackaged while loop that puts all the information about how long the loop should continue in one place. Thus a for loop is generally easier to understand than an equivalent while loop. Head First Java describes for loops on pages 112 and 113.

5.2.2 (Display page) Nested for loops

Nested for loops

Here is the triangle drawing code you devised in a previous lab exercise.

```java
int row = 0;
while (row < 10) {
    int col = 0;
    while (col <= row) {
        System.out.print ('*');
        col = col + 1;
    }
    row = row + 1;
    System.out.println ( );
}
```

Here it is again, coded with for loops.

```java
for (int row=0; row<10; row=row+1) {
    for (int col=0; col<=row; col=col+1) {
        System.out.print ('*');
    }
    System.out.println ( );
}
```

5.2.3 (Display page) Shortcuts to incrementing and decrementing

Shortcuts to incrementing and decrementing

Suppose k is an integer variable. Then the statement

```java
k++;
```
has the same effect as the statement

```java
k = k + 1;
```
There is a similar shortcut for decrementing:

```
k--;  
```

has the same effect as

```
k = k - 1;  
```

The motivation for this shorthand notation is that the operations of incrementing and decrementing by 1 are very common. It is legal in Java to increment or decrement a variable within a larger expression, for example, `System.out.println(values[k++]);`. Don't do this; it's asking for off-by-one errors. We will actively discourage you (sometimes with point deductions) from using the `++` or `-` operators anywhere but on a line by themselves.

5.2.4 (Display page) For statements used with arrays

For statements go hand-in-glove with arrays. Consider, for example, an array named `values`. It is very common to see code like the following:

```
for (int k=0; k<values.length; k=k+1) {  
    // process values[k]  
}
```

5.2.5 (Display page) The break statement

The `break` statement "breaks out of" a loop (either a `for` or a `while`). That is, it stops the execution of the loop body, and then continues with the statement immediately following the loop. An example of its use would be a program segment that searches an array named `values` for a given value, setting the variable `found` to `true` if the value is found and to `false` if it isn't. (This would be similar to the `member` procedure in Scheme.)

```
for (int k=0, found=false; k<values.length; k++) {  
    if (values[k] == value) {  
        found = true;  
        break;  
    }
}
```

5.2.2 (Display page) Insert and delete

Insert and delete

Create a new project named "array operations", and copy the two files `~cs61b/code/ArrayOperations.java` and `~cs61b/code/ArrayOperationsTest.java` to it. Then fill in the blanks in the `ArrayOperations` class. Your methods should pass the tests in `ArrayOperationsTest`.

The `insert` method takes three arguments: an `int` array, a position in the array, and an `int` to put into that position. All the subsequent elements in the array are moved over (i.e. up one position) to make room for the new element; the last value in the array is lost. For example, if an array named `values` contains the five elements 1, 2, 3, 4, and 5, calling `insert(values,2,7)` would result in `values` containing 1, 2, 7, 3, and 4.

The `delete` method takes two arguments: an `int` array and a position in the array. The subsequent elements are moved down one position; the value 0 is assigned to the last array element. For example, if an array named `values` contains the five elements 1, 2, 3, 4, and 5, calling `delete(values,2)` would result in `values` containing 1, 2, 4, 5, and 0.

5.3.2 (Display page) Zip

Zip

Add a method named `zip` to the `ArrayOperations` class. Given two arrays `array1` and `array2` of the same length, `zip` should return an array that's twice as long, in which the elements of `array1` and `array2` are interleaved. That is, element 0 of the result array is `array1[0]`, element #2 is `array1[1]`, element #3 is `array2[1]`, and so on. The `zip` method should not change its arguments. Here is a `testZip` method to copy into the `ArrayOperationsTest` class.

```
public void testZip ( ) {  
    int [ ] array1 = {1, 2, 3};  
    int [ ] array2 = {4, 5, 6};  
    int [ ] zipResult = {1, 4, 2, 5, 3, 6};  
    check (ArrayOperations.zip (array1, array2), zipResult);  
    // Shouldn't change arguments.  
    int [ ] array1copy = {1, 2, 3};  
    int [ ] array2copy = {4, 5, 6};  
    check (array1, array1copy);  
    check (array2, array2copy);  
    check (ArrayOperations.zip (new int[0], new int[0]), new int[0]);  
    // Check boundary case.  
    check (ArrayOperations.zip (new int[0], new int[0]), new int[0]);  
}
```

5.3.3 (Brainstorm) Testing insertion

A lazy CS 61B student says that only one test case of `insert` is necessary, namely insertion into the middle of the array. Argue with this student.

5.3.4 (Display page) Zip

Zip

Add a method named `zip` to the `ArrayOperations` class. Given two arrays `array1` and `array2` of the same length, `zip` should return an array that's twice as long, in which the elements of `array1` and `array2` are interleaved. That is, element 0 of the result array is `array1[0]`, element #2 is `array1[1]`, element #3 is `array2[1]`, and so on. The `zip` method should not change its arguments. Here is a `testZip` method to copy into the `ArrayOperationsTest` class.

```
public void testZip ( ) {  
    int [ ] array1 = {1, 2, 3};  
    int [ ] array2 = {4, 5, 6};  
    int [ ] zipResult = {1, 4, 2, 5, 3, 6};  
    check (ArrayOperations.zip (array1, array2), zipResult);  
    // Shouldn't change arguments.  
    int [ ] array1copy = {1, 2, 3};  
    int [ ] array2copy = {4, 5, 6};  
    check (array1, array1copy);  
    check (array2, array2copy);  
    check (ArrayOperations.zip (new int[0], new int[0]), new int[0]);  
    // Check boundary case.  
    check (ArrayOperations.zip (new int[0], new int[0]), new int[0]);  
}
```

5.3.5 (Brainstorm) Opportunities for errors

What errors did you encounter in working through the array exercises? (If you didn't encounter any yourself, suggest some errors that you were careful to avoid.)

5.4 Examine a collection class

(4 steps)

5.4.1 (Display page) Collection classes

Collection classes
The remaining activities will involve the implementation of collection classes, which represent collections of data. Most of the data structures we will study in the rest of the semester are used to represent collections. The most commonly used collections are sets and sequences. Sets support (at least) the following operations:

- initialization (with a constructor);
- adding an element to the set if it's not already there;
- removing an element from the set;
- checking whether a given item is in the set; and
- checking whether the set is empty.

Sequences differ from sets in that one may talk about the position of an element in the sequence. Sequence operations will thus include the following:

- initialization;
- adding an element to the sequence at a given position, or at the end of the sequence;
- removing the element at a given position, or at the end;
- identifying the position of a given element of the sequence; and
- checking whether the sequence is empty.

Implementation of a sequence normally involves storing the elements of the sequence explicitly. For example, one might use an array whose 0th element is the first sequence item, whose 1st element is the second sequence item, and so forth. We'll consider this implementation in a later activity today. A set has weaker requirements, and therefore more implementation alternatives. A set of nonnegative integers, for example, may be represented as a boolean array, with element \( k \) being true if \( k \) is in the set and false otherwise. We will explore this implementation now.

5.4.2 (Display page) Collection constructors

**Collection constructors**

We have seen several ways to initialize instance variables in *Head First Java*:

- using assignments in the main method (chapters 2 through 4);
- using "setter" methods (chapters 4 and 5);
- using constructors (chapter 9).

In general, the most preferable by far of these ways is the last one. By convention, the constructor of a class initializes the instance variables of the class in such a way that the resulting object is internally consistent and complete.

5.4.3 (Self Test) Constructor contents

We are representing a set of nonnegative integers as a boolean array as just described. The constructor will be passed the largest integer to be stored in the set as an argument. Which of the following will the constructor contain?

5.4.4 (Display page) Completing a set implementation

**Completing a set implementation**

The file `~cs61b/code/SeqFW.java` contains the framework of a `Set` class that uses an array of booleans as just described. The file `~cs61b/code/SetTest.java` is a JUnit test class for this class. Complete and test the `Set` class.

5.5 Examine another collection class. (11 steps)

5.5.1 (Display page) An IntSequence class

**An IntSequence class**

Consider now a class that represents a sequence of an arbitrary number of integers (possibly including negative integers). As noted previously, an array is a reasonable implementation of the sequence. However, a complication arises from not knowing ahead of time how many integers there will be. (Recall that arrays, unlike Scheme lists, have a specified number of elements they can hold.) If you choose an array size—say, 20—and you end up with only 7 integers, there will still be values in the last 13 array elements that you won't be able to distinguish from the other integers. A solution is to store as instance variables not only an integer array but also a count that tells how much of the array contains actual interesting integers. We implement this solution in a class named `IntSequence`.

Two diagrams appear below. The shaded array cells represent the "uninteresting" part of the array; if the code is working correctly, the values in the shaded cells should never be examined.

<table>
<thead>
<tr>
<th>a partially full IntSequence</th>
<th>an empty IntSequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>myCount</td>
<td>myValues</td>
</tr>
<tr>
<td>6</td>
<td>3 21 4 0 -7 6</td>
</tr>
<tr>
<td>myCount</td>
<td>myValues</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

5.5.2 (Display page) A constructor

**A constructor**

Create a new project and a class file for the `IntSequence` class. Then supply the constructor by filling in the indicated blanks in the `IntSequence` framework below.

```java
public class IntSequence {
    // instance variables
    private int[] myValues; // sequence elements
    int myCount; // number of array cells used by the sequence

    // Constructor; the argument will be the actual size
    // of the array, or equivalently, the (temporary) maximum
    // number of elements it can hold.
    public IntSequence (int capacity) {
        // implementation...
    }
}
```

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
5.5.3 (Display page) Some simple methods

Some simple methods

Add three methods to the IntSequence class:

- **public boolean isEmpty ( )**
  - returns true when this sequence is empty and returns false otherwise.
- **public int size ( )**
  - returns the number of values in this sequence (note the distinction between the size and the capacity).
- **public int elementAt (int pos)**
  - returns the value at the given position in the sequence. For example, if the sequence contains the elements 3, 1, and 4, `elementAt(0)` should return 3.

5.5.4 (Display page) An add method

An add method

Add the `add` method to the `IntSequence` class. `add` appends its `int` argument to the stored integer sequence. The argument integer will be placed in the first unused spot in the array, as shown in the following diagram:

```
// Include the argument among those stored in myValues
// by placing it in the first unused spot in the array
// and incrementing the count.
// Assumption: myValues isn't full.
public void add (int toBeAdded) {
    ________
}
```

5.5.5 (Display page) A toString method

A toString method

Supply a `toString` method for the `IntSequence` class. It will return a `String` that contains the elements of the sequence separated by blanks. You now have enough of the `IntSequence` to write a JUnit test. Use `assertEquals` with the `toString` method to examine the contents of a sequence.

5.5.6 (Display page) Tests for remove

Tests for remove

Consider an `IntSequence` method named `remove` that, given a single argument—the position of the sequence element to remove—removes the specified element. An example of how it should work is shown below.

```
before call to remove
myCount myValues
7 3 -7 42 -11 0 6
```

```
after remove (2)
myCount myValues
6 3 -7 42 -11 0 9
```

Devises JUnit tests to test the `remove` method, and add them to your `IntSequenceTest.java` file.

5.5.7 (Display page) Deletion of the element at a given position

Deletion of the element at a given position

Rename your `delete` method from earlier in lab to be `remove`, then adapt it to work with `IntSequence` objects (that is, to be a method of the `IntSequence` class). You should already have invented tests for it in the previous step.

5.5.8 (Display page) Tests for insert

Tests for insert

Now invent tests for an `insert` method that, given an integer and a position in this sequence at which to insert it, moves all the later sequence elements over to make room for the new element, then copies the integer into the appropriate place. Add the tests to your `IntSequenceTest.java` file.

5.5.9 (Brainstorm) A buggy insert method

The following `insert` method has a bug. Did your tests from the previous step expose it? Explain why or why not, and describe the bug.

```
// Insert toInsert into the sequence at position insertPos,
// shifting the later elements in the sequence over to make room
// for the new element.
// Assumptions: the array isn't full, i.e. myCount < myValues.length.
// Also, insertPos is between 0 and myCount, inclusive.
public void insert (int toInsert, int insertPos) {
    for (int k=insertPos+1; k<=myCount; k++) {
        myValues[k] = myValues[k-1];
    }
    ________
}
```
5.5.10 (Display page) Insertion, revisited

Insertion, revisited

Adapt your insert method from earlier in lab to be a method of the IntSequence class. The insert method will take two arguments: the integer value to insert into the sequence, and the position at which that value is to be inserted.

5.5.11 (Display page) A contains method

A contains method

Add tests to your IntSequenceTest.java file to call a contains method. Given an int argument—we'll call it k—contains returns true if k is one of the elements of this sequence, and returns false if not. Then implement your contains method and run the tests.

5.6 Homework (2 steps)

5.6.1 (Display page) Programming + reading

Programming + reading

The homework assignment described in the next step is due prior to your Tuesday or Wednesday section next week (February 12 or 13). You should finish up today's lab activities, and read Appendix B, sections 7 and 2, in Head First Java. Section 4.3 in Objects, Abstractions, Data Structures, and Design also contains material relevant to IntSequence, but it assumes you know about more advanced Java features. It may help, however, to skim this chapter to familiarize yourself with the content for when we revisit it next week.

5.6.2 (Display page) Programming: a deck class

Programming: a deck class

This homework is due prior to your lab on Tuesday or Wednesday (February 12 or 13). Put your name and lab section time in everything you turn in. Submit three files, Deck.java, DeckTest.java, and readme, as your solution. Design and test a Deck class that represents a deck of playing cards. It will have at least private three IntSequence instance variables, one named myDealCards that stores cards that have been dealt from the deck, another named myCards to store the cards remaining in the deck, and a third named myDiscards that stores cards that once were in the deck but have been discarded and are no longer in use. The Deck class will support the following operations:

- a public 1-argument constructor.
  - The argument, an int, specifies the number of cards initially in the deck; we'll call it deckSize. The myCards Object should be initialized to a random permutation (that is, a random ordering) of the integers 0, 1, ..., deckSize-1; myDealCards and myDiscards should each be initialized to an empty IntSequence. If deckSize is less than or equal to zero, the constructor should print an error message using System.err.println and call System.exit(1).
- public Card dealtCard ( )
  - If no more cards remain in myCards, replace myCards with an IntSequence that contains a random permutation of the integers in myDiscards; then replace myDiscards with an empty IntSequence Object. There should now be at least one card in myCards—we assume that myCards and myDiscards are not both empty—so remove the first card from myCards, add it to myDealCards, and return it as the value of dealtCard. If dealtCard is called when myCards and myDiscards are both empty, dealtCard should print an error message using System.err.println and call System.exit(1).
- public void discard (Card c)
  - The argument c must be in myDealCards. Remove it from myDealCards and add it to myDiscards. If discard is called with an argument that's not in myDealCards, it should print an error message using System.err.println and call System.exit(1).
- public boolean isOK ( )
  - This provides a consistency check on the deck instance variables, by checking that no cards have been lost or duplicated, i.e. that myDealCards, myDiscards, and myCards together contain all the cards, and that their combined sizes sum to the original deck size.
- public void invalidate (int k)
  - Depending on the value of the argument, invalidates the contents of one or more of the instance variables so that isValid returns false. Values from 0 to 15 are acceptable arguments.

The Deck class should also include a nested class named Card. Here is the code.

```java
public class Card {
    private int myValue;
    private Card (int index) {
        myValue = index;
    }
    private int toInt ( ) {
        return myValue;
    }
    public boolean equals (Object obj) {
        Card c = (Card) obj;
        return c.myValue == myValue;
    }
}
```

It is organized in this way to ensure that client classes may use Card objects but may not create them. A somewhat tricky part of this assignment is the coding of a shuffle method, which should take time proportional to the size of the deck being shuffled; we'll discuss this in the next lab. Finally, you are to provide a JUnit test suite in the file DeckTest.java and a text file named readme that justifies your test cases according to testing principles described in earlier labs. DeckTest.java can only access private class information indirectly through the isOK method. However, it should include tests of as much observable Deck behavior as possible. For example, you are able to determine that a shuffle of the deck doesn't crash.) In the readme file, describe how you chose your tests and what
potential errors you were unable to test. To grade your solution, we will use our own version of the 
IntSequence class. We will also supply our own main method for the Deck class; your Deck.java file 
should not include a main method.

6 Loop invariants; data invariants and iterators (2008-2-07 ~ 2008-2-08) (5 activities)

6.2 Think about loop invariants (in partnership) (11 steps)

6.2.1 (Display page) Definition of loop invariant

Find a partner as usual; tell him or her your favorite music groups. Then work on this batch of 
activities. This material is somewhat abstract; we encourage you to explain each example to your 
partner, to solidify your understanding. Each part of any program we write should have a concise 
and well-defined goal. (That aids the process of abstraction that you learned about in CS 61A.) For 
example—once we start writing methods—we should be able to sum up any method we write in a 
one-or two-sentence comment. The well-defined goal should simplify the design of the code and 
increase our confidence in its correctness. The same applies to loops: any execution of a loop body 
should have a well-defined effect that's essentially the same for the last execution as for the first.

We generally define this effect with a loop invariant, a property that is true each time the loop test is 
evaluated. ("Invariant" means "not varying" or "not changing"). You’ve seen the recursive counterpart 
—a recursion invariant—in CS 61A, in particular with tail recursive code:

```java
;; Return the sum of the elements of the list L.
(define (sum L) (sum-helper L 0 0))

;; Invariant property:
;; so-far is the sum of the first k elements of the original argument to sum.
(define (sum-helper L so-far k) 
  (if (null? L) 
    so-far 
    (sum-helper (cdr L) (+ (car L) so-far) (+ k 1)) )
```

6.2.2 (Display page) Examples of loop invariants

Examples of loop invariants

An example of a loop invariant arises in the triangle drawing code from an earlier activity; the 
comments in the code below specify the invariant properties.

```java
int row = 0;
while (row < 10) {
    // at this point, rows 1 through row have been printed.
    int col = 0;
    while (col <= row) {
        // at this point, col asterisks have been printed in the row'th row
        System.out.print ('*');
        col = col + 1;
    }
    row = row + 1;
    System.out.println ( );
}
```

Another example is a loop that simulates a winning strategy for a simple two-player game. The 
game starts with a pile of 20 marbles. The two players take turns, each taking 1, 2, or 3 marbles 
from the pile. The winner is the player who takes the last marble. Here is pseudocode for the loop.

```java
while (true) {
    player 1 takes 1, 2, or 3 marbles;
    if (no more marbles) {
        print 'player 1 wins';
        break;
    }
    player 2 takes 1, 2, or 3 marbles;
    if (no more marbles) {
        print 'player 2 wins';
        break;
    }
}
```

A winning strategy for player 2 in this game is to make sure that the number of marbles is always a 
multiple of 4. That is the invariant property for the loop. Pseudocode that uses the invariant appears 
below.

```java
while (true) {
    // Invariant property:
    // the number of marbles is divisible by 4.
    player 1 takes 1, 2, or 3 marbles;
    if (no more marbles) {
        print 'player 1 wins';
        break;
    }
    player 2 takes the number of marbles to restore the invariant;
    if (no more marbles) {
        print 'player 2 wins';
        break;
    }
}
```

We see the pattern: The invariant property is true at the start of the loop; there may be some 
processing within the loop that invalidates the invariant, but by the end of the loop the invariant 
property is true again.

6.2.3 (Display page) Loop invariants used for analysis

Loop invariants used for analysis

Loop invariants can be used to reason about existing code. Consider, for example, the debugging 
activity you did last week. Here's the code.

```java
public boolean contains1MoreThan (String s) {
    if (myString.length ( ) == 0) {
        return false;
    } else if (s.length ( ) == 0) {
        return myString.length ( ) == 1;
    } else if (myString.charAt(0) == s.charAt(0)) {
        Debug remainder = new Debug (myString.substring(1));
        return remainder.contains1MoreThan (s.substring(1));
    } else {
        return myString.substring(1) == s;
    }
```
This recursion either hits a base case in the first, second, or fourth condition, or makes a recursive call in the third condition. The recursive call happens when the first characters of the two strings match.

Further thinking suggests a good comment for the method, namely that all the characters seen so far in one of the strings match their counterparts in the other. This statement is just a restatement of the recursion invariant. We can then use that invariant to check the rest of the code.

1. if (myString.length == 0) ...
   myString has run out of characters either before or at the same time as s has, so it can't possibly be the result of inserting a single character into s.
2. if (s.length == 0) return myString.length == 1;
   The insertion was at the end since all the earlier characters match.
3. else return myString.substring(1) == s;
   The invariant is invalidated and we can't restore it. Therefore we must return something that checks that the mismatch we just encountered is the only mismatch between the two strings.

The invariant property we noticed tells us that the first two conditions are handled correctly, and the bug must therefore be in the last condition. (The bug is that == was used instead of equals.)

6.2.4 (Display page) Unhelpful invariants

Unhelpful invariants

There are numerous properties that are true throughout execution of a typical loop or recursion. Consider, for example, the corrected code from the previous step:

```java
public boolean contains1MoreThan (String s) {
  if (myString.length ( ) == 0) {
    return false;
  } else if (s.length ( ) == 0) {
    return myString.length ( ) == 1;
  } else if (myString.charAt(0) == s.charAt(0)) {
    Debug remainder = new Debug (myString.substring(1));
    return remainder.contains1MoreThan (s.substring(1));
  } else {
    return myString.substring(1).equals(s);
  }
}
```

s.length and myString.length are always nonnegative. Big deal. For an invariant to be useful in showing that code works correctly, it has to say something about all relevant variables in the code. Such invariants typically say something about the number of iterations executed so far in a loop, or the number of recursive calls. Here's an example for the contains1MoreThan method.

Let k be the number of recursive calls so far, and let origS and origString be the argument and the contents of myString in the first call to contains1MoreThan. Then the first k characters of origS match the first k characters of origString, and the current values of s and myString result from removing the first k characters of origS and origString respectively.

This lets us relate the result from the current call to contains1MoreThan to the result to be returned by the initial call.

6.2.5 (Brainstorm) Finding another invariant

Given below is a method that, given x (a double) and n (an int), computes x^n quickly.

```java
public static double power (double x, int n) {
  double runningPower = 1.0;
  int nn;
  xx = x;
  nn = n;
  while (nn > 0) {
    if (nn%2 == 1) {
      runningPower = runningPower * xx;
    }
    nn = nn/2;
    xx = xx*xx;
    // *** What invariant property holds here?
  }
  return runningPower;
}
```

Determine an invariant property that, at the end of every iteration through the loop, relates the variables runningPower, xx, nn, x, and n.

6.2.6 (Display page) An application: debugging a binary search

An application: debugging a binary search

Many bugs arise from incorrect processing in a loop that fails to restore the invariant property. The program described below is an example. The file ~cs61b/code/NumberGuesser.java contains a program that uses the binary search technique to guess the user's number. Binary search starts with a range of values all in order—here, the integers between 0 and 20. To find a particular value, we first look at the middle value in the range. If the middle value is not what we're looking for, we check whether what we want is higher or lower than the middle value. If it's higher, we don't need to search the lower values; if it's lower, we don't need to search the higher values. Here's a sample dialog with the program.

Please think of an integer between 0 and 20 (inclusive).
Is your number 10? (Type y or n.)
  n
Is 10 too high? (Type y or n.)
  y
Is your number 5? (Type y or n.)
  n
Is 5 too high? (Type y or n.)
  n
Is your number 7? (Type y or n.)
  n
Is 7 too high? (Type y or n.)
Is your number 8? (Type y or n.)

Y

The loop invariant for this search involves two variables, low and high. These represent the endpoints of the range of values that could possibly contain the user's value. low starts out at 0 and high at 20, reflecting the assumption that the user's number is somewhere between 0 and 20. At each guess, the program shrinks the range of values that can contain the user's value, either by lowering high (and discarding large values) or by raising low (discarding small values). The code has a bug relating to incorrectly maintaining the loop invariant. Find the bug—this is a good opportunity to familiarize yourself better with Eclipse's debugging facility—then answer the next two activities.

6.2.7 (Brainstorm) Values that expose the bug

Describe the values (there may be only one) that expose the bug, and explain how you figured them out.

6.2.8 (Brainstorm) Buggy update of the invariant

Explain the error in updating the invariant, that is, why low or high is assigned an incorrect value.

6.2.9 (Display page) Invariants in array processing

Here is a common invariant pattern that shows up in array processing. The processing involves a loop (we assume the array's name is values):

```java
for (int k=0; k<values.length; k++) {
    // At this point, the invariant is valid for the subarray
    //   that contains elements 0, 1, ..., k-1 of values.
    Process element k.
    // At this point, the invariant is valid for the subarray
    //   that contains elements 0, 1, ..., k of values.
}
```

Often the processing of element k consists of including it somehow among elements 0, 1, ..., k-1. The loop invariant property says something about the first k elements or elements 0, 1, ..., k. Thus the invariant pattern is

```java
for (int k=0; k<values.length; k++) {
    // At this point, the invariant is valid for the subarray
    //   that contains elements 0, 1, ..., k-1 of values.
    Process element k.
    // At this point, the invariant is valid for the subarray
    //   that contains elements 0, 1, ..., k of values.
}
```

Here's an example. Suppose that values is an array of ints whose elements are all either 0 or 1, and suppose we want to rearrange the elements so that all the 0's precede all the 1's.

<table>
<thead>
<tr>
<th>array before processing</th>
<th>array after processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>1 1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>1 0 1 0 0</td>
<td>0 0 0 1 1</td>
</tr>
</tbody>
</table>

The loop invariant will be that all the elements of the appropriate subarray are arranged with all the 0's preceding all the 1's. The code below uses a variable named posOfFirst1, which either contains the position of the first 1 encountered so far in the array or, if no 1's have yet been seen, 1 plus the position of the last 0.

```java
posOfFirst1 = 0;
// no zeroes seen yet, so pos of last 0 may be viewed as -1
for (int k=0; k<values.length; k++) {
    // elements 0 through k-1 are arranged zeroes first
    if (values[k] == 0) {
        // exchange values[k] with values[posOfFirst1]
        int temp = values[k];
        values[k] = values[posOfFirst1];
        values[posOfFirst1] = temp;
        // now values[posOfFirst1] is 0, so the first 1 (if any)
        // is right next to it
        posOfFirst1++;
    } else {
        // do nothing, since the position of the first 1 hasn't changed
        // elements 0 through k are now arranged zeroes first
    }
}
```

Work through this with your partner. The code is available in ~cs61b/code/Arrange01.java. One way to experiment with the code is to set a breakpoint at the first statement in the loop and then to verify by inspection that the invariant property is true.

6.2.10 (Display page) Using an invariant to design a sorting method

Here's another example of the same pattern. Given below is the framework of a Sorter class that contains a single method named sort. Complete the framework so that your code matches the comments. (The method will implement an algorithm called insertion sort.) Then test it with the JUnit tests in ~cs61b/code/SorterTest.java.

```java
public class Sorter {
    public static void sort (int [] values) {
        for (int k=1; k<values.length; k++) {
            // Elements 0 through k-1 are in nondecreasing order:
            // values[0] <= values[1] <= ... <= values[k-1].
            // Insert element k into its correct position, so that
            // values[0] <= values[1] <= ... <= values[k].
            ...
        }
    }
}
```

Here's how insertion sort is supposed to work. At the start of the kth time through the loop, the leftmost k elements of the values array are in order. The loop body inserts values[k] into its proper place among the first k elements (probably moving some of those elements up one position in the array), resulting in the first k+1 elements of the array being in order. The table below shows a trace of what happens when an array initially contains 3, 1, 5, 4, 2 is sorted.
### 6.2.11 (Display page) Summary of what you've learned

#### Summary of what you've learned

Summarize what you've learned about invariants for your partner. Make sure that both of you are clear on the concept before you go back to working alone.

### 6.3 Get some hints on the homework (working alone) (2 steps)

#### 6.3.1 (Display page) Producing a random integer

**Producing a random integer**

Return to working alone. One occasionally wishes to produce a randomly selected integer from a given range, say, 0 to $n-1$. The `Math` class library contains a method named `random` that returns a randomly selected `double` that's greater than or equal to 0 and less than 1. We can use this to produce a random integer as follows:

1. Multiply the value returned by `Math.random` by $n$, and store the product in a `double` named `temp`. The stored value is at least 0 and less than $n$.
2. The random integer between 0 and $n-1$ (inclusive) results from truncating `temp`, that is, using a cast as described on page 117 of *Head First Java*. Here's code that does that:

   ```java
   int randInt = (int) temp;
   ```

### 6.3.2 (Display page) Application to the homework

**Application to the homework**

Here's a hint for the homework. One way to shuffle a deck of cards (randomly permute its elements) is another example of the array processing pattern described earlier. The invariant at the start of the loop is:

- elements 0, ..., $k-1$ are shuffled.

After the last statement in the loop body, we have:

- elements 0, ..., $k$ are shuffled.

The processing done in the loop body will include selecting a random card, using the procedure for producing a random integer described in the previous step.

### 6.4 Return elements one by one from an array (7 steps)

#### 6.4.1 (Display page) Iterators defined

**Iterators defined**

One obvious way to process every element of an array of values is a `for` loop:

```java
for (int k = 0; k < the length of the array; k++) {
    process the kth array element;
}
```

In CS 61B, though, we're sensitive to the possibility of having to change the data structure in which we store the values. If we store the values in a list or tree, we obviously won't be able to use the array-based loop to process the values. To minimize changes that result from replacing one data structure by another (in CS 61A terms, to build a stronger abstraction barrier), we need a more abstract way for accessing the elements of a structure in sequence. It's called an *iterator*. An iterator is a collection of three methods, plus an private instance variable.

- One method initializes the iteration. We'll call it `initIterator`.
- The second is a boolean method—here, we'll call it `hasNext`—that says whether there are any more values remaining to return.
- The third method, which we'll call `next`, successively returns the values one by one. The first call to `next` returns the first value, the second call to `next` returns the second value, and so on.
- In order for `next` to know what value to return next, there must also be some “place-keeping” information that keeps track of the state of the iteration. That's where the instance variable comes in.

With the three methods just described, we can construct equivalent code to the `for` loop mentioned earlier:

```java
initIterator();
while (hasNext()) {
    value = next();
    process value;
}
```

One should not assume that calls to `next` always alternate with calls to `hasNext`. For example, the program segment:

```java
initIterator();
boolean moreRemain = hasNext();
boolean moreStillRemain = hasNext();
value = next();
```

### Table

<table>
<thead>
<tr>
<th>$k$</th>
<th>contents of values at start of loop body</th>
<th>contents of values at end of loop body</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3, 1, 5, 4, 2</td>
<td>1, 3, 5, 4, 2</td>
<td>now the first two elements are known to be in order</td>
</tr>
<tr>
<td>2</td>
<td>1, 3, 5, 4, 2</td>
<td>1, 3, 5, 4, 2</td>
<td>now the first three elements are known to be in order</td>
</tr>
<tr>
<td>3</td>
<td>1, 3, 5, 4, 2</td>
<td>1, 3, 4, 5, 2</td>
<td>now the first four elements are known to be in order</td>
</tr>
<tr>
<td>4</td>
<td>1, 3, 4, 5, 2</td>
<td>1, 2, 3, 4, 5</td>
<td>now the whole array is in order</td>
</tr>
</tbody>
</table>
should still result in value storing the first item in the sequence. Consecutive calls to `next` are also allowed.

6.4.2 (Display page) Iterators for arrays

Iterators for arrays

Here, we will design the `IntSequence` iterator, `hasNext`, and `next` methods for returning all the values stored in an array, one by one. A key to iterator design is the state information that keeps track of where we are in the iteration. We have to decide information what to store, and how to update it.

With an array, a reasonable way to keep track of the iteration is to store a position in the array. If this position contains 0, we're on the first element; if it's 1, we're on the second element, and so on.

We'll give this variable the name `iteratorIndex`. We guess that the `hasNext` method will assign the value 0 to `iteratorIndex`; the next method will increment that value, thereby "moving" to the next array element; and the `moreRemain` method will check `iteratorIndex` to see if it has gone past the end of the array. In earlier activities, we considered the idea of a loop invariant, a property that held at the start of every iteration of the loop. Choosing the invariant first helps us design the loop, then also helps us verify that it works correctly. Similarly, we should look for a property involving `iteratorIndex` that holds between calls to the various iterator methods. (This is basically a loop invariant for the loop that's calling the iterator methods, except that it will be invisible to the caller.) Two choices for `iteratorIndex` are to have it be the position of the element of the array `next to be returned`, or to have it be the position of the element `just returned`. Since it's not clear what the position of the element just returned should start at (when no elements have been returned yet), we make the first choice. Here is some pseudocode.

```java
private int iteratorIndex;

// The next element to return is the first, namely element 0.
public void initIterator() {
    iteratorIndex = 0;
}

// iteratorIndex is the position of the element to return.
// We save that element, then increment iteratorIndex to preserve the invariant.
public int next() {
    int valToReturn = myValues[iteratorIndex];
    iteratorIndex++;
    return valToReturn;
}

// More elements remain exactly when the element at position iteratorIndex exists,
// iteratorIndex < length of the array.
public boolean hasNext() {
    return iteratorIndex < length of the array;
}
```

Note that `hasNext` doesn't change the iterator state, so it may be called several times in succession without interfering with the sequence of items returned by `next`.

6.4.3 (Display page) Adding an iterator to the `IntSequence` class

Adding an iterator to the `IntSequence` class

Add the three iterator methods and the `iteratorIndex` variable to the `IntSequence` class, and modify the `toString` method to use the iterator methods rather than accessing `myValues` directly. Run the program to check that you modified it correctly, and test it with the JUnit test you designed for the `toString` method last lab.

6.4.4 (Display page) Modifying the iterator

Modifying the iterator

Modify the iterator so that it returns only the nonnegative values in the sequence. Maintain the invariant property already designed: between calls to the iterator methods, `iteratorIndex` should always be the position of the value next to be returned by `next`. Here is pseudocode for a JUnit test.

```java
1. Create a new `IntSequence` object named `seq` that initially holds 5 elements.
2. Add the integer -1 to the sequence. `seq.toString()` should return an empty string.
3. Add the integer 7 to the sequence. `seq.toString()` should return the string "7".
4. Add the integer -2 to the sequence twice. After each add, `seq.toString()` should return the string "7".
5. Add the integer 9 to the sequence. `seq.toString()` should return the string "7 9".
6. Remove the -1 via `seq.remove(-1).toString()` should return the string "7 9".
7. Add the integer 2 to the sequence. `seq.toString()` should return the string "7 9 2".
```

6.4.5 (Brainstorm) Iterator tests

Identify all the features of iterators that are being tested by the pseudocode just suggested (reprinted below):

```java
1. Create a new `IntSequence` object named `seq` that initially holds 5 elements.
2. Add the integer -1 to the sequence. `seq.toString()` should return an empty string.
3. Add the integer 7 to the sequence. `seq.toString()` should return the string "7".
4. Add the integer -2 to the sequence twice. After each add, `seq.toString()` should return the string "7".
5. Add the integer 9 to the sequence. `seq.toString()` should return the string "7 9".
6. Remove the -1 via `seq.remove(-1).toString()` should return the string "7 9".
7. Add the integer 2 to the sequence. `seq.toString()` should return the string "7 9 2".
```

6.4.6 (Display page) An alternative invariant

An alternative invariant

On paper, design an iterator that returns only the positive values in an `IntSequence` object, and maintain the following invariant:

- Between calls to the iterator methods, `iteratorIndex` should be the position of the most recently returned value.
Your \texttt{initIterator} method should set \texttt{iteratorIndex} to -1. Keep in mind that 0 or more than one call to \texttt{hasNext} may separate calls to \texttt{next}.

6.4.7 (Brainstorm) Comparison of the two iterator versions

Comment on the advantages and disadvantages of one of the iterator versions over the other (i.e. comparing the version in which \texttt{iteratorIndex} is the position of the \texttt{next} element to return, with the version in which \texttt{iteratorIndex} is the position of the \texttt{most recently returned item}).

6.5 Homework (1 step)

6.5.1 (Display page) Programming + reading

Programming + reading

You should complete all the activities from this lab, and finish the implementation and testing of the \texttt{Deck} class that was assigned earlier this week. Here are some additional details regarding that assignment. (The clarifications have also been added to the problem statement in the Tuesday/Wednesday homework activity.)

- The three instance variables \texttt{myCards}, \texttt{myDealtCards}, and \texttt{myDiscards} should all be private.
- Your \texttt{Deck} class should not include a \texttt{main} method.
- Your JUnit \texttt{DeckTest} file should include calls to \texttt{isOK} and \texttt{invalidate}. It should also include calls to the public \texttt{Deck} methods that test as much of the internal \texttt{Deck} behavior as possible. (For example, you are able to determine that a shuffle of the deck doesn't crash.)
- In your \texttt{readme} document, describe how you chose your tests and what potential errors you were unable to test.
- Your \texttt{readme} file should be a text file. Saying "cat readme" should produce only text.
- Put your name and lab section time in all your files.

Finally, read chapters 7 and 8 in \textit{Head First Java} and chapter 3 in \textit{Objects, Abstraction, Data Structures, and Design}. The concept of inheritance should be familiar to you from CS 61A, so we hope you can focus mainly on the Java details.

7 Inheritance; callbacks; interfaces 2008-2-12 ~ 2008-2-13 (5 activities)

7.2 Practice using inheritance (6 steps)

7.2.1 (Display page) Review of inheritance from CS 61A

Review of inheritance from CS 61A

You learned in CS 61A that a programmer can set up one class to inherit characteristics (methods and instance variables) from another. This is typically done to reuse most of an already-defined class that needs an extra method, or that requires a method to behave slightly differently. For example, some of the CS 61A activities involved an adventure game. Basic classes in the game were \texttt{Person}, \texttt{Place}, and \texttt{Thing}, and one could define a \texttt{Food} class to inherit from \texttt{Thing} and \texttt{Thief} and \texttt{Police} classes to inherit from \texttt{Person}.

7.2.2 (Display page) Terminology

Terminology

We will refer to the inheriting class as the subclass of its parent or superclass, and say that the subclass \texttt{extends} the superclass. Methods in the superclass that are redefined in the subclass are said to be \texttt{overridden}. In Java, we set up inheritance in a class's header, using the keyword \texttt{extends}. You may have noticed examples of its use already this semester in applications involving graphics, for instance in the \texttt{DotRace} class:

```java
private class MyFrame extends Frame { ...
```

Here's a simpler example. Recall the \texttt{Counter} class from earlier in the course:

```java
public class Counter {
    private int myCount;
    public Counter ( ) {
        myCount = 0;
    }
    public void increment ( ) {
        myCount++;
    }
    public void reset ( ) {
        myCount = 0;
    }
    public int value ( ) {
        return myCount;
    }
}
```

Now suppose we want to define a "mod N" counter, whose value cycles from 0 to N-1 and back. There are lots of real-life examples of mod N counters including clocks, automobile odometers, and television channel selectors. One way to set up a mod N counter would be to inherit the \texttt{increment} and \texttt{reset} operations from \texttt{Counter}, replacing only the constructor and the \texttt{value} method.

```java
public class ModNCounter extends Counter {
    private int myN;
    public ModNCounter (int n) {
        myN = n;
    }
    public int value ( ) {
        // cycles around from 0 to myN-1
        return myCount % myN;
    }
}
```

This almost works.

7.2.3 (Display page) Access to superclass information

Access to superclass information
The one flaw in the pair of classes we just set up is that the ModNCounter code does not have access to the private instance variable myCount. This restriction makes sense. A programmer defining a variable as private presumably intends that access to the variable be limited. However, if all you had to do was gain access to a private variable was to define a subclass of the class containing it, it would be easy to subvert the limited access. There are two ways to fix this problem in ModNCounter and Counter. One is for the value method in Counter to call its counterpart in Counter rather than use myCount directly. In order to avoid an infinite recursion, we have to make clear which value method we’re calling using the super keyword:

```java
public int value ( ) {
   // Get the counter’s value, then apply the mod n operation.
   int v = super.value ( );
   return v % myN;
}
```

Another is to redefine myValue in the Counter class to allow access to subclasses, but not to anyone outside the inheritance structure. We do this using the protected keyword, substituting

```java
protected int myCount;
```

```java
for
private int myCount;
```

The code for the Counter class (with private myCount) and the ModNCounter class with a main method for testing is in the ~cs61b/code directory. Experiment with these two ways of accessing the counter’s value from the ModNCounter class.

### 7.2.4 (Brainstorm) Comparison of access methods
Which solution to allowing ModNCounter access to the Counter value do you think is better, redefining myCount or protected or calling the Counter value method (using super)? Briefly explain your choice.

### 7.2.5 (Display page) Using inheritance to trace assignments to objects

#### Using inheritance to trace assignments to objects

Some classes provide “setter” methods. A useful debugging aid is to override a setter method to produce informative output at each time that the object’s state changes. We’ve already worked with the Point class in the java.awt library. Its setter method is named move; the call

```java
p.move (27, 13);
```

changes the x coordinate to 27 and the y coordinate to 13 in the Point referenced by p. Given below is the framework for a TracedPoint class intended to intercept calls to move and print information about the pre- and post-change state of the object along with doing the move. You are to complete and test the framework.

```java
import java.awt.

public class TracedPoint extends Point {
   public TracedPoint (int x, int y) {
      super (x, y);
   }
   // Your move method goes here.

   public static void main (String [ ] args) {
      TracedPoint p1 = new TracedPoint (5, 4);
      p1.move (3, 4);
      p1.move (9, 10);
      TracedPoint p2 = new TracedPoint (25, 30);
      p2.move (45, 50);
      System.out.println ("p1 is " + p1);
      System.out.println ("p2 is " + p2);
   }
}
```

Take advantage of the superclass's methods and variables as much as possible rather than reinventing the wheel. A note about the TracedPoint constructor: Java automatically supplies a call to the superclass constructor with no arguments (here, Point { }). If you want to call one of the other constructors of the superclass, you have to use the super keyword as shown above. The super keyword means the superclass of the class you’re in. super by itself means the constructor method of the superclass; super.methodName means the method in the superclass with the given name.

### 7.2.6 (Brainstorm) Not reinventing the wheel

#### How did you organize your TracedPoint class to "take advantage of the superclass’s methods and variables as much as possible"?

### 7.3 Work with polymorphism. (18 steps)

#### 7.3.1 (Display page) Another advantage of inheritance

#### Another advantage of inheritance

We saw earlier that inheritance provides a way to reuse existing classes, implementing small changes in behavior by overriding existing methods in the superclass or by adding new methods in the subclass. We defined the class TracedPoint to inherit from Point; we also recalled the use of inheritance in CS 61A to define hood to inherit from Thing and Thief and Police to inherit from Person. Inheritance also, however, makes it possible for us to design general data structures and methods using polymorphism. The word “polymorphism” comes from the Greek words for “many” and “forms”. In the context of object-oriented programming, it means that a given object can be regarded either as an instance of its own class, as an instance of its superclass, as an instance of its superclass’s superclass, and so on up the hierarchy. In particular, where a given reference type is requested for a method argument or needed for a return type, we can supply instead an instance of any subclass of that reference type. That’s because inheritance implements an “is-a” relationship: a TracedPoint is a Point with some extra properties; a Thief Of a Police is a Person with extra properties; a Food is a Thing with extra properties.

Here’s an example:

```java
public static void moveTo79 (Point p) {
   p.move (7, 9);
}
```
We can call `moveTo79` with a `TracedPoint` argument. Not only will this be accepted by the compiler, but the compiled code will use the `move` method of `TracedPoint`.

**7.3.2 (Brainstorm) A proposed variation**

Would you expect the substitution mechanism to work in reverse? For example, can we call the method

```java
    public static void moveTo79 (TracedPoint tp) {
        tp.move (7, 9);
    }
```

with

```java
    Point p = new Point (3, 4);
    moveTo79 (p);
```

Briefly explain why you would expect this to work or not to work.

**7.3.3 (Display page) Polymorphic data structures**

**Polymorphic data structures**

The `java.util` class library contains several collection classes that take advantage of polymorphism and are therefore able to store a variety of types of objects. We will examine the `ArrayList` class as an example. It represents an expandable array-like structure. (It's described in chapter 6 of Head First Java.) To declare an `ArrayList` reference, one specifies not only the `ArrayList` class name but also the class of objects that the `ArrayList` will contain in angle brackets. For example,

```java
   ArrayList<String> values;
```

declares a reference to an `ArrayList` that contains only `String` objects. Similarly, to construct an `ArrayList` object, one supplies the element class name in angle brackets, for example,

```java
   ArrayList<String> values = new ArrayList<String> ( );
```

For now, since we want to experiment with polymorphic data structures, we will work with an `ArrayList<Object>`.

`ArrayList` methods include the following.

```java
    // Add an object to the end of the ArrayList.  
    // The return value is always true (and is therefore  
    // usually ignored) since there aren't any circumstances  
    // that the add could fail. 
    boolean add (Object o);

    // Return the object at the given position in the ArrayList. 
    Object get (int index);

    // Return the number of elements that have been added  
    // to the ArrayList; similar to the myCount information in  
    // the IntSequence class. 
    int size ( );

    // Return true if the ArrayList contains the given object,  
    // and return false if not. 
    boolean contains (Object o);
```

The class `Object` is the root of the inheritance hierarchy—every class inherits from `Object`, at least indirectly—so these operations provide a data structure that can store objects of any type. Here's an example:

```java
   ArrayList<Object> a = new ArrayList<Object> ( );
   a.add (new TracedPoint (5, 6));
   a.add (new Point (10, 11));
   a.add ("abcd"); // a String object
   IntSequence seq = new IntSequence (3);
   seq.add (5);
   seq.add (4);
   seq.add (3);
   a.add (seq);
   for (int k=0; k<a.size( ); k++) {
      System.out.println (a.get (k));
   }
```

The output is

```
    TracedPoint[x=5,y=6]
    java.awt.Point[x=10,y=11]
    abcd
    5 4 3
```

showing that each object's own `toString` method was used to construct the corresponding output line.

**7.3.4 (Display page) A problem**

**A problem**

Unfortunately, elements of an `ArrayList` seem to selectively forget some of their methods. Given the following code:

```java
   Point p = new Point (3, 4);
   TracedPoint tp = new TracedPoint (5, 6);
   ArrayList<Object> a = new ArrayList<Object> ( );
   a.add (p);
   a.add (tp);
   // Move both points to (7, 9).
   for (int k=0; k<a.size( ); k++) {
      a.get(k).move (7, 9);
   }
```

the compiler claims that

```
Cannot resolve symbol
    symbol : method move (int,int)
  location: class java.lang.Object
    a.get(k).move (7, 9);
```

It appears that the Java compiler is looking not at the `Point` and `TracedPoint` classes to find the
move method, but at the Object class.

7.3.5 (Brainstorm) A proposed solution

Try replacing the line

\[ a.get(k).move (7, 9); \]

by

\[ \text{Point } p2 = a.get(k); \]
\[ p2.move (7, 9); \]

Does it fix the problem (and allow the two calls to move)? Why or why not?

7.3.6 (Display page) Polymorphic method selection

Polymorphic method selection

The Java compiler and runtime system have to resolve two questions that arise when polymorphism is used. The compiler, which wants to catch inconsistencies and other possibilities for error, asks whether a method with a given name can be invoked on a given object. Once the compiler has answered "yes" to that question, the runtime system needs to find which of the possible versions of that method is the right one to invoke.

7.3.7 (Display page) What the compiler does

What the compiler does

To determine the legality of the method call

\[ p.move (7, 9); \]

the compiler examines the type of \( p \). If the class for which \( p \) is a reference, or one of its superclasses, contains a move method that takes two integers as arguments, the statement is legal. If not, an error message results. Similarly, for a statement like

\[ a.get(3).move (7, 9); \]

the compiler examines the type of \( a.get(3) \), searching for a move method either in the corresponding class or one of its superclasses. According to the online documentation for ArrayList, the get method returns an Object. Objects don't have a move method; thus, the error message.

We just considered the alternative of

\[ \text{Point } p = a.get(3); \]
\[ p.move(7,9); \]

The statement \( p.move(7,9); \) would work fine; \( p \) is a reference of type Point, and the Point class has a move method. The compiler's complaint arises from the assignment statement; it objects to the attempt to take a reference to an Object and store it in a Point reference. Since almost all Objects are not Points, the compiler has probability on its side. We, however, are smarter than the Java compiler. We know that we have only inserted Points and TracedPoints into the vector, and we know that they have move methods. We communicate this information to the compiler by a type cast, putting a class name in parentheses immediately preceding a reference or reference expression.

Here's how we would successfully call the move methods of the vector elements.

\[
\text{for (int k=0; k<a.size(); k++) {
    ((Point) a.get(k)).move (7, 9);
}}
\]

The compiler knows that \( a.get \) returns a reference to an Object. The cast causes the reference to be interpreted as a Point, which allows the compiler to find a move method.

7.3.8 (Display page) What happens at run time

What happens at run time

Once the compiler is satisfied that a move method is available for an object, the runtime system then selects the most specialized move method to call. For a Point, it uses the Point move method. For a TracedPoint, it uses the TracedPoint move method. We just considered the code

\[
\text{ArrayList<Object> a = new ArrayList<Object> ( ) ;}
\]
\[
\text{a.add (new TracedPoint (5, 6));}
\]
\[
\text{a.add (new Point (10, 11));}
\]
\[
\text{seq.add (5);}
\]
\[
\text{seq.add (4);}
\]
\[
\text{for (int k=0; k<a.size(); k++) {
    System.out.println (a.get (k));
}}
\]

Why did it work? Because the compiler noted that a toString method is defined in the Object class, and was satisfied. A mantra to help understand this: an object always remembers how it was constructed, and chooses a method from the class of which it is an instance.

7.3.9 (Self Test) Polymorphism puzzles

Consider the class definitions below.

| public class Superclass { |
| public void print ( ) { System.out.println ("super"); } |
| } |

| public class Subclass extends Superclass { |
| public void print ( ) { System.out.println ("sub"); } |
| } |

Now determine whether the following program segment prints "super", prints "sub", results in a compile-time error, or results in a run-time error.

\[
\text{Superclass obj1 = new Subclass ( );}
\]
\[
\text{obj1.print ( );}
\]

Do the same for the following program segment.

\[
\text{Subclass obj2 = new Superclass ( );}
\]
obj2.print();

Do the same for the following.

Superclass obj3 = new Superclass();
((Subclass) obj3).print();

Do the same for the following.

Subclass obj4 = new Subclass();
((Superclass) obj4).print();

7.3.10 (Display page) Callbacks

Callbacks

It is common in Java for a method we provide to be called not by our own code but by someone else's. The example we've seen in earlier activities is the toString method. toString is rarely called directly. More likely, it is called by code that needs to produce a people-readable representation of an object, for instance, the System.out.println method. The situation where we call a method of a library class and another of our methods gets called as a result is referred to as a callback. Another frequent participant in a callback is the equals method. For example, the Vector contains method calls equals to determine if the argument to contains is an element of the vector, in a manner that recalls the implementation of Scheme's member procedure:

```
(define (member x L)
  (cond
   ((null? L) #f)
   ((equal? x (car L)) #t)
   (else (member x (cdr L)))))
```

We will experiment with the equals method in the context of the Measurement class coded in an earlier lab exercise.

7.3.11 (Display page) Measurement.equals, version 1

```
Measurement.equals(), version 1

Add the following methods to your Measurement class. (You'll also need to add an import statement for java.util.) Predict what output ought to appear. Then run the program and observe the results.

```java
public boolean equals(Measurement m) {
    return this.toString().equals(m.toString());
}
```

```
public static void main(String[] args) {
    ArrayList<Measurement> a = new ArrayList<Measurement>();
a.add(new Measurement(3, 5));
a.add(new Measurement(10, 9));
a.add(new Measurement(6, 3));
    Measurement m1 = new Measurement(2, 7);
a.add(m1);
    Measurement m2 = new Measurement(10, 9);
    if (a.contains(m1)) {
        System.out.println("a contains 2' 7");
    } else {
        System.out.println("a doesn't contain 2' 7");
    }
    if (a.contains(m2)) {
        System.out.println("a contains 10' 9");
    } else {
        System.out.println("a doesn't contain 10' 9");
    }
}
```

7.3.12 (Brainstorm) Explanation of version 1

Explain the results you just observed. (The equals method was intended to make both calls to contains succeed.)

7.3.13 (Brainstorm) Measurement.equals, version 2

The following equals assumes the existence of instance variables myFeet and myInches. Under that assumption, would it produce the intended behavior of success in both calls to contains? Explain why or why not.

```
public boolean equals(Measurement m) {
    return m.myFeet == this.myFeet && m.myInches == this.myInches;
}
```

7.3.14 (Brainstorm) Measurement.equals, version 3

What's the effect of replacing Measurement m by Object m in the equals header? Briefly explain, and indicate what other changes are necessary to make it work.

```
public boolean equals(Object m) {
    return m.myFeet == this.myFeet && m.myInches == this.myInches;
}
```

7.3.15 (Brainstorm) Measurement.equals, version 4

Now change Measurement m to Object m in the first version of equals:

```
public boolean equals(Object m) {
    return m.toString().equals(m.toString());
}
```

Explain why this version of equals produces different behavior from the version in the previous step (where Measurement m was changed to Object m in a method using myFeet and myInches, with no other modifications).

7.3.16 (Display page) How polymorphism fits in

How polymorphism fits in

We have seen that ArrayList.contains apparently requires an equals method defined as follows:

```
public boolean equals(Object obj) ...
As it happens, many other collection classes in \texttt{java.util} assume that the classes of their elements include an \texttt{equals} method defined as above. The reason is that these collection classes are set up to contain any object, not just \texttt{Measurements} and the like. We earlier noted that in Java, the object class is the root of the inheritance hierarchy. Every class extends \texttt{Object}. One of the methods provided in the \texttt{Object} class is \texttt{equals}; our redefinition of \texttt{equals} overrides \texttt{Object}'s version. In \texttt{java.util.Arrays}, there is a call to \texttt{equals}. Its argument is most likely a reference to a subclass of \texttt{Object}. The compiler verifies that an \texttt{equals} method is available. \texttt{object.equals} suffices if no overriding version has been provided.) Then, at run time, the most specific \texttt{equals} is used to make the comparison. Another of \texttt{Object}'s methods is \texttt{toString}.

\begin{verbatim}
public String toString ( )

The \texttt{toString} method we defined earlier for \texttt{Measurements} override \texttt{Object.toString}. When a method like \texttt{System.out.print} calls \texttt{toString}, it uses the most specific version.
\end{verbatim}

7.3.17 (Display page) If we don't override these methods ...

If we don't override these methods ...

As just noted, the \texttt{Object} class contains definitions for \texttt{equals} and \texttt{toString}. Unfortunately, they are not too useful. \texttt{Object.equals} compares the object's reference—basically, its location in memory—to the argument reference, performing the same comparison as does \texttt{==}. \texttt{Object.toString} returns a printable representation of this reference. Thus it is a Java convention, when defining a class, to provide definitions for \texttt{equals} and \texttt{toString}. The \texttt{equals} method should compare two objects for content. It typically starts by casting the argument in order to access relevant instance variables, then does the comparison. We just saw a version of \texttt{equals} that uses \texttt{toString}; this is fine as long as every object in the class has a unique printable representation.

7.3.18 (Self Test) One more note about \texttt{equals}

What's the difference between \texttt{eq} and \texttt{eq?} in Scheme? (Don't worry if your correct answer is diagnosed as "incorrect"; the system is doing character-by-character comparisons to check correctness.)

7.4 Learn about using Java interfaces. (3 steps)

7.4.1 (Display page) Syntax of Java interface use

Syntax of Java interface use

Java provides a construct called an interface, which is a collection of method header declarations. For example, the interface named \texttt{Iterator} in \texttt{java.util} names three method headers:

\begin{verbatim}
public boolean hasNext ( )
public Object next ( )
public void remove ( )
\end{verbatim}

A class can implement an interface by supplying definitions for all its methods. In Java, we add the phrase

\begin{verbatim}
implements interfaceName
\end{verbatim}

to the class header, for example,

\begin{verbatim}
public class IterableClass implements Iterator {

// ... implementation

// Other methods

\end{verbatim}

7.4.2 (Display page) Motivation for interfaces

Motivation for interfaces

Some of you may know that Apple has requirements for the user interface of programs that run on the Macintosh. For instance, each program that involves editing data is supposed to supply a "File" menu that includes elements "Open", "Close", and "Print", along with an "Edit" menu that includes elements "Copy", "Cut", and "Paste". They supply library routines that simplify the setting up of these menus. A program that obeys the guidelines presents a consistent interface to the user and is deemed "Mac-like". A program that doesn't obey the guidelines is "un-Mac-like", and may be harder to learn and use as a result. Java interfaces support the same kind of consistency for programmers as the Macintosh user interface guidelines do for users. By specifying method headers, they tell the programmer exactly what to name his or her methods in order to coordinate with other parts of the Java class libraries. For example, the \texttt{sort} method in \texttt{java.util.Arrays}. Sorting involves comparing array elements, so a user of the \texttt{sort} method needs to supply that method with a mechanism for doing the comparison. One does this by having the element class implement the \texttt{Comparable} interface and supply a method named \texttt{compareTo} to do comparison between elements. There is a close connection between Java interfaces and callbacks. A programmer wanting to sort an array would probably not call \texttt{compareTo} directly. Instead, he or she would call \texttt{Arrays.sort}, which in turn would call the user-provided method. \texttt{Head First Java} explains interfaces in terms of abstract classes, a feature of Java that we won't be using much in CS 61B. Whatever the explanation, however, the idea is the same: an interface specifies a contract for a class that implements the interface, when then can be relied upon to supply the relevant methods for users of the class.

7.4.3 (Display page) Sorting measurements

http://spring08.ucwise.org/builder/builderPortal.php?BUILD menu=curriculumSummary
Sorting Measurements

Modify your `Measurement` class to work with the following `main` method. (You'll also need to import `java.util`.)

```java
public static void main (String [ ] args) {
    Measurement [ ] marray = new Measurement [4];
    marray[0] = new Measurement (5, 2); // 5' 2"
    marray[1] = new Measurement (2, 9); // 2' 9"
    marray[2] = new Measurement (6, 3); // 6' 3"
    marray[3] = new Measurement (0, 11); // 11"
    Arrays.sort (marray);
    for (int k=0; k<marray.length; k++) {
        System.out.println (marray[k]);
    }
    // should print the measurements in ascending order:
    // 0' 11", 2' 9", 5' 2", and 6' 3".
}
```

7.5 Homework (2 steps)

7.5.1 (Display page) Reading + review requests

Reading + review requests

For homework, provide a contribution to the activity in the next step. T.a.s will use your input to devise review activities for the first lab next week. Also read pages 1 through 22 (the first two sections) of the “Bowling Scores” case study for your next lab period. The case study will be handed out in lab today. It is also online here, and the code is available in the directory ~cs61b/code/Scorer.java and ScorerTest.java. Finally, read chapter 11 in *Head First Java*.

7.5.2 (Brainstorm) Topics needing review for the exam

Suggest a topic or technique to be covered on the exam that you would most like your t.a. to review in lab on next Tuesday or Wednesday.

8 More on inheritance; a case study

2008-2-14 ~ 2008-2-15 (6 activities)

8.2 Work through more inheritance puzzles (with a partner) (3 steps)

8.2.1 (Display page) Partnering

Partnering

The next activities provide more practice with the details of inheritance. Working in partnership can help you not to miss or misunderstand these details. Find a partner as usual, and exchange information about your favorite cell phone features. Then work together on the next two activities.

8.2.2 (Display page) Playing more with inheritance

Playing more with inheritance

Work with a partner on these two exercises. Consider the following code (a slight variation of the program on page 192 of *Head First Java*).

```java
public class A {
    protected int ivar = 7;
    /*
    public void m1 () {
        System.out.print ("A's m1, ");
    }
    */
    public void m2 () {
        System.out.print ("A's m2, ");
    }
    public void m3 () {
        System.out.println ("A's m3, ");
    }
}

public class B extends A {
    public void m1 () {
        System.out.print ("B's m1, ");
    }
}

public class C extends B {
    public void m3 () {
        System.out.println ("C's m3, " + (ivar+6));
    }
}

public class Mixed {
    public static void main (String [ ] args) {
        A a = new A ();
        B b = new B ();
        C c = new C ();
        A a2 = new C ();
        b.m1 ();
        c.m2 ();
        a.m3 ();
        c.m1 ();
        c.m2 ();
        c.m3 ();
        a.m1 ();
        b.m2 ();
        c.m3 ();
        a2.m1 ();
        a2.m2 ();
        a2.m3 ();
    }
}
```
1. Identify what will be printed by each group of three lines in Mixed.main. Possibilities are

- A's m1, A's m2, C's m2, 6
- B's m1, A's m2, A's m3,
- A's m1, B's m2, A's m1, B's m1, A's m2, C's m3, 13
- B's m1, C's m2, A's m3,
- B's m1, A's m2, C's m3, 6
- A's m1, A's m2, C's m3, 13

The solution appears on page 195 of Head First Java.

2. Identify which line or lines in Mixed.main would cause a compiler error if the m1 method were removed from class A. (You can check your answer by first splitting the program into four pieces, one for each public class, and then either importing the program into Eclipse or copying it to UNIX files and running javac on it.)

8.2.3 (Display page) Exceptions

8.2.2 (Brainstorm) Flaws in error-handling mechanisms

8.2.1 (Display page) Error handling

8.3.1 (Display page) Using IntSequence with the Bowling Scores code

8.2.3 (Display page) Debriefing

Try to answer any of your partner’s questions about inheritance, and have him or her clear up your own confusion (if any). If you’re both confused about some aspect of inheritance, check with your t.a. or a lab assistant.

8.3 Work with the “Bowling Scores” program

8.3.1 (Display page) Using IntSequence with the Bowling Scores code

Using IntSequence with the “Bowling Scores” code

The files ~cs61b/code/Scorer.java and scorerTest.java contains the programs from Appendix D of the case study. Import them into your IntSequence project, restore the iterator methods to the version that returns all elements of the sequence, and make the following modification to the “Bowling Scores” code.

- Modify the program so that roll returns an IntSequence instead of an int array. Elements of the returned sequence should be exactly those that had been previously returned in the int array.

This modification will of course require changes to the Scorer methods. The JUnit test suite will have to be changed as well; however, rather than change each individual test... method, you need merely restrict your changes to the testMain method.

8.4 Work with Java exceptions

8.4.1 (Display page) Error handling

Error handling

So far, we have not had to worry much about error handling. In most of the CS 61B assignments so far, you were allowed to assume that the arguments to methods were formatted or structured appropriately. This is not always the case: bugs in your programs might result in inconsistent structures or erroneous method calls, and of course users can’t ever be trusted to do the right thing. We assume in the following discussion that we can detect the occurrence of an error and at least print an error message about it. A big challenge in dealing with the error is to provide information about it at the right level of detail. For instance, consider the error of running off the end of an array or list. A “reference through a null pointer” or “index out of bounds” error message might be the result of inconsistent updating of the contents of a list and the number of elements in that list. Thus one might wish to pass information about the error back to the calling method in the hope that the caller can provide more appropriate and useful information about the root cause of the error. Here are some options for handling errors in this situation.

1. Don’t try to pass back any information to the caller at all. Just print an error message and halt the program.
2. Detect the error and set some global error indicator to indicate its cause.
3. Require the method where the error occurs, along with every method that, directly or indirectly, calls it, to pass back as its value or to take an extra argument object that can be set to indicate the error.

All these options have flaws.

8.4.2 (Brainstorm) Flaws in error-handling mechanisms

Here are the three mechanisms just described for dealing with an error situation.

1. Don’t try to pass back any information to the caller at all. Just print an error message and halt the program.
2. Detect the error and set some global error indicator to indicate its cause.
3. Require the method where the error occurs, along with every method that, directly or indirectly, calls it, to pass back as its value or to take an extra argument object that can be set to indicate the error.

Identify a flaw with each mechanism.

8.4.3 (Display page) Exceptions

Exceptions

A fourth option, provided in Java and other modern languages, uses exceptions, essentially signals that an error of some sort has happened. Java allows both the signalling of an error and selective handling of that error. Methods called between the signalling method and the handling method need not be aware of the possibility of the error. Two definitions: an exception is thrown by the code that detects the exceptional situation, and caught by the code that handles the problem. There is a wide spectrum of “exceptional” situations. On the one hand, there are catastrophic situations like the computer powering off. On the other hand are situations that might not even correspond to errors. Encountering the end of an input file, for example, is a likely event in a file processing program; reaching the end of a list might naturally occur in an unsuccessful search. In between are errors that might occur as the result of bugs—dividing by 0, index out of bounds or dereference of a null pointer—and situations caused by a user error—nonnumeric input where a number is required, search failures where a command or keyword was misspelled. Java’s exception facility classifies all these
into two categories: checked exceptions that a method must explicitly handle or hand off to its caller, and unchecked exceptions that a method need not worry about.

8.4.4 (Display page) Exceptions in Java

Exceptions are objects in Java, all extending the exception class. Unchecked exceptions extend the RuntimeException class, a subclass of Exception. Here are some examples of each type of exception. Unchecked exceptions:

- ArithmeticException, e.g. dividing by 0
- EmptyStackException, thrown by methods of the Stack class
- IllegalArgumentException, thrown as the result of a consistency check
- IndexOutOfBoundsException, thrown by methods in array-based classes
- NullPointerException
- ArrayStoreException, thrown when the wrong type of value is stored in an array
- ClassCastException
- NoSuchElementException, thrown by an iterator method
- NumberFormatException, thrown by Integer.parseInt

Checked exceptions:

- FileNotFoundException, thrown by an input method that's supposed to be reading from a file

The class IOException is the superclass of EOFException and FileNotFoundException. The programmer can easily define his or her own exceptions by extending Exception or RuntimeException. Each has two constructors to override, one with no arguments, the other with a String argument in which an error message is stored. Here's an example.

```java
class Disaster extends RuntimeException {
    public Disaster ( ) {
        super ( );
    }
    public Disaster (String msg) {
        super (msg);
    }
}
```

8.4.5 (Display page) Throwing and catching exceptions

Throwing and catching exceptions

To catch an exception, surround code that might produce it with a try {} catch block as follows.

```java
try {
    code that might produce the exception
} catch (exceptionName variableName) {
    code to handle the exceptional situation
}
```

We've seen an example of this in the DateConverter program from the first week of the semester:

```java
try {
    dayOfYear = Integer.parseInt (keyboard.readLine ( ));
} catch (NumberFormatException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

This code is reading what is supposed to be a day of the year from the keyboard. Error situations to look out for include general input errors, say, resulting from a damaged keyboard, or inappropriately formatted input, say, the string "mike". In either case, the above code prints the chain of method calls that got to this point (the "stack trace"). Observe that the "tried" code can be simpler since it is coded as if nothing will go wrong. One can catch multiple exceptions, as in the code above, and handle them separately by supplying multiple catch blocks (ordered from most specific to least specific). If your code may produce a checked exception, you have two choices. One is to catch the exception. The other is to "pass the buck" by saying throws exceptionName in the method header. This puts responsibility on the calling method either to handle the exception or to pass the buck to its caller. To throw an exception, we use the throw operator and give it a newly created exception as argument. For example, if the number read above must be positive, we could have the following:

```java
dayOfYear = Integer.parseInt (keyboard.readLine ( ));
if (dayOfYear <= 0) {
    throw new IllegalArgumentException ("day of year must be positive");
}
```

In the exception-catching code, we may access the String argument to the Exception constructor via the getMessage() method.

8.4.6 (Display page) Bulletproofing time input

Bulletproofing time input

The following code is also in ~cs61b/code/Time.java. It represents a time of day in military time, storing a number of hours that's between 0 and 23 and a number of minutes between 0 and 59.

```java
public class Time {
    private int myHours;
    private int myMinutes;
    public Time (String s) {
        int colonPos = s.indexOf ("\":");
        myHours = Integer.parseInt (s.substring (0, colonPos));
        myMinutes = Integer.parseInt (s.substring (colonPos+1));
    }
    public Time (int hours, int minutes) {
    }
}
```
public boolean equals (Object obj) {
    Time t = (Time) obj;
    return myHours == t.myHours && myMinutes == t.myMinutes;
}

public String toString ( ) {
    return myHours + ":" + myMinutes;
}

Here is a method for testing the constructor, suitable for pasting into a JUnit file.

```java
public void testConstructor ( ) {
    String [ ] timeArgs = {null, "x", "x:", ":x", "x:y", 1, ":30", ":4: 35", 555.000, "11:99", "3:30", "00004:45", "4:00?", "47", "4 :09", "1:30", "11:55");
    Time [ ] correctTimes = {null, null, null, null, null, null, null, null, null, null, Time (1, 10), new Time (11, 35));
    for (int k=0; k<timeArgs.length; k++) {
        Time t = new Time (timeArgs[k]);
        assertEquals (correctTimes[k], t);
    }
}
```

Catch all exceptions that arise within the Time constructor and handle them by throwing an IllegalArgumentException with an informative message string. (Eclipse has a handy operation named "Surround with try/catch block" in the Source menu that will simplify this operation.) Four cases for illegal times must be tested separately: too many leading zeroes in the hours or minutes (e.g. "00007"), and values for hours and minutes that are out of range. You should add the tests for these cases and throw IllegalArgumentException with informative message strings. Also change the testConstructor method to catch IllegalArgumentException and print its message. Our solution produces eight different error messages for the given test cases.

**8.4.8 (Display page) Data invariants**

**Data invariants**

We have already considered a number of kinds of invariant relations:

- a loop invariant, a property that is true at the start of the loop and after every time through the loop;
- a recursion invariant, a property that is true at the start of the recursion and for every recursive call;
- the invariant property that holds after every call to an iterator method.

Another kind of invariant involves properties of and relations between the values of a class's instance variables. In a Measurement object with instance variables myFeet and myInches, for example, a programmer might require that myInches is between 0 and 11 and myFeet is nonnegative. A Fraction class might require of its numerator and denominator variables that the numerator is positive and the denominator and numerator share no factors other than 1 (i.e. the fraction is stored in lowest terms). Good defensive programming suggests the inclusion, in each class, of a method that verifies that the data invariant relations hold. (One might name this method "check", or "isOK", or some such.) The method would throw an IllegalStateException (built into Java) if the invariants were not satisfied. Each method in the class that changes the values of the instance variables should call this checking method to ensure that the variable values are still internally consistent. (If the consistency checking doesn't result in too much of a performance hit, it should be used not only while debugging but also in production versions of the code.)

**8.4.8 (Brainstorm) Line consistency**

Here is a version of the Line class from earlier this semester:

```java
public class Line {
    private Point [ ] endPoints;
    ...}
```

Supply an isOK method that throws IllegalStateException if the Line object doesn't represent a line with two end points.

**8.5 Get acquainted with java.util (3 steps)**

**8.5.1 (Display page) Java collections**

**Java collections**

The java.util library provides an interface for collections of objects, unsurprisingly named Collection, along with several classes that implement this interface. Some example Collection methods are:

<table>
<thead>
<tr>
<th>method name</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean add (Object obj)</td>
<td>ensures that this collection contains obj; returns true if this collection changes as a result of the addition.</td>
</tr>
<tr>
<td>boolean contains (Object obj)</td>
<td>returns true if this collection contains obj.</td>
</tr>
<tr>
<td>boolean equals (Object obj)</td>
<td>compares obj with this collection for equality.</td>
</tr>
<tr>
<td>boolean isEmpty ( )</td>
<td>returns true if this collection contains no elements.</td>
</tr>
<tr>
<td>boolean remove (Object obj)</td>
<td>removes a single instance of obj from this collection; returns true if this collection contained obj.</td>
</tr>
<tr>
<td>int size ( )</td>
<td>returns the number of elements in this collection.</td>
</tr>
</tbody>
</table>

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
Collections include sets, which store only one copy of a given object, and lists, which can store multiple copies. There are also interfaces named set and list that extend collection to provide more specialized operations. The list interface, for example, provides methods that refer to an element’s position in the list, for example:

<table>
<thead>
<tr>
<th>method name</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>void add (int index, Object obj)</td>
<td>inserts obj at the specified position in this list (the position of subsequent elements increases by 1).</td>
</tr>
<tr>
<td>Object remove (int index)</td>
<td>remove and return the object at the specified position in this list (the position of subsequent elements decreases by 1).</td>
</tr>
<tr>
<td>Object get (int index)</td>
<td>return the object at the specified position in this list.</td>
</tr>
<tr>
<td>Object set (int index, Object obj)</td>
<td>replace the element at the specified position in this list, and return the replaced element.</td>
</tr>
</tbody>
</table>

Classes in java.util that implement the list interface are Vector, ArrayList, and LinkedList. Vector and ArrayList are array-based; they are similar to the IntSequence class we worked with in lab. LinkedList uses a doubly-linked list (to be covered next week).

8.5.2 (Brainstorm) Vector’s evolution
The Vector class was included in the first version of java.util, then was retrofitted to extend the list interface. How can you tell? Browse the methods of the Vector class, and describe some evidence of the retrofitting.

8.5.3 (Display page) Iterators vs. Enumerations

Iterators vs. Enumerators
The java.util library provides two interfaces for iterating through collection elements. One, named Enumeration, provides the methods hasMoreElements and nextElement. An Enumeration is a read-only iteration; we don’t expect the underlying collection to be changing during the enumeration of elements. You may guess, based on your own iterator implementations, that changes to the collection might cause some elements to be skipped or returned multiple times. Another, named Iterator, allows the element most recently returned to be removed from the underlying collection. More bookkeeping is required to provide this facility. In CS 61BL, we will generally be using Enumerations rather than the more general Iterator. A possible convenience is provided by a subclass of Iterator named ListIterator, which allows elements to be enumerated either first to last or last to first from the underlying collection. The Vector class provides both elements, a method that returns an iterator, and Iterator and ListIterator, which return an iterator and a ListIterator respectively. The ArrayList class only provides iterator and listIterator methods.

8.6 Homework (4 steps)

8.6.1 (Display page) Reading + programming + discussion

There will be an exam in class on Wednesday, February 20. Information about this exam is available here. There will be a quiz next lab, but not much else. Labs will run review sessions and answer questions. Labs next week will cover chapter 4 in Objects, Abstraction, Data Structures, and Design, and the document “Notes on Linked Data Structures” in CS 61B Readings. Contribute to the following discussion topics, both with a post and with a comment or two on one of your classmates’ posts.

8.6.2 (Discussion Forum) Coping with CS 61B so far
What has been the hardest thing for you that we’ve covered in CS 61B so far? (If everything has been easy, there still is a “hardest” topic or technique.) Please be specific. In response to two of your classmates’ posts, describe strategies that you have tried or would try to understand the difficult topics they mention. Again, please be specific.

8.6.3 (Discussion Forum) Important parts of the case study
In your post, list what you believe are the three most important things to be learned from the “Bowling Scores” case study. Then respond to a labmate who chose different things to learn from the case study by commenting on the differences between your lists.

8.6.4 (Discussion Forum) Comments on the design and development of the Bowling Scores program
If you’re an EECS major, pick some aspect of the design and development of the “Bowling Scores” program that you would have done differently or that you disagree with. Describe this in a post, and defend your alternate approach or opinion. If you’re not an EECS major, pick some aspect of the design and development of the “Bowling Scores” program that you thought was particularly good or that you learned a lot from. Describe this in a post. Then argue with someone on the other side. If he or she disagrees with some aspect of the case study, defend it; if he or she found some part of the design/development noteworthy and you didn’t agree with that, provide a response that supports your opinion.

10 Linked lists and algorithm analysis

10.1 Work with linked lists

10.1.1 (Display page) What a linked list is

What a linked list is
Early in CS 61BL, we considered an Account class whose instance variables included another Account object (the account providing overdraft protection). What we were really building back then was a linked list of accounts. Any object with an instance variable of the same class creates a linked structure, with that instance variable providing the “link”. An object with a link is often referred to as a node of the linked list.

10.1.2 (Display page) A one-node circularly linked structure

A one-node circularly linked structure
Create a class with a link, then construct a node that points to itself. Use the debugger to verify that the node points to itself. (Your Account class may be useful here.)

10.1.3 (Display page) Scheme lists

8.6.1 (Display page) Reading + programming + discussion

There will be a quiz next lab, but not much else; t.a.s will run review sessions and answer questions.

There will be an exam in class on Wednesday, February 20. Information about this exam is available here. There will be a quiz next lab, but not much else; t.a.s will run review sessions and answer questions.

Labs next week will cover chapter 4 in Objects, Abstraction, Data Structures, and Design, and the document “Notes on Linked Data Structures” in CS 61B Readings. Contribute to the following discussion topics, both with a post and with a comment or two on one of your classmates’ posts.

8.6.2 (Discussion Forum) Coping with CS 61B so far
What has been the hardest thing for you that we’ve covered in CS 61B so far? (If everything has been easy, there still is a “hardest” topic or technique.) Please be specific. In response to two of your classmates’ posts, describe strategies that you have tried or would try to understand the difficult topics they mention. Again, please be specific.

8.6.3 (Discussion Forum) Important parts of the case study
In your post, list what you believe are the three most important things to be learned from the “Bowling Scores” case study. Then respond to a labmate who chose different things to learn from the case study by commenting on the differences between your lists.

8.6.4 (Discussion Forum) Comments on the design and development of the Bowling Scores program
If you’re an EECS major, pick some aspect of the design and development of the “Bowling Scores” program that you would have done differently or that you disagree with. Describe this in a post, and defend your alternate approach or opinion. If you’re not an EECS major, pick some aspect of the design and development of the “Bowling Scores” program that you thought was particularly good or that you learned a lot from. Describe this in a post. Then argue with someone on the other side. If he or she disagrees with some aspect of the case study, defend it; if he or she found some part of the design/development noteworthy and you didn’t agree with that, provide a response that supports your opinion.

10 Linked lists and algorithm analysis

10.1 Work with linked lists

10.1.1 (Display page) What a linked list is

What a linked list is
Early in CS 61BL, we considered an Account class whose instance variables included another Account object (the account providing overdraft protection). What we were really building back then was a linked list of accounts. Any object with an instance variable of the same class creates a linked structure, with that instance variable providing the “link”. An object with a link is often referred to as a node of the linked list.

10.1.2 (Display page) A one-node circularly linked structure

A one-node circularly linked structure
Create a class with a link, then construct a node that points to itself. Use the debugger to verify that the node points to itself. (Your Account class may be useful here.)

10.1.3 (Display page) Scheme lists

8.6.1 (Display page) Reading + programming + discussion

There will be a quiz next lab, but not much else; t.a.s will run review sessions and answer questions.

There will be an exam in class on Wednesday, February 20. Information about this exam is available here. There will be a quiz next lab, but not much else; t.a.s will run review sessions and answer questions.

Labs next week will cover chapter 4 in Objects, Abstraction, Data Structures, and Design, and the document “Notes on Linked Data Structures” in CS 61B Readings. Contribute to the following discussion topics, both with a post and with a comment or two on one of your classmates’ posts.

8.6.2 (Discussion Forum) Coping with CS 61B so far
What has been the hardest thing for you that we’ve covered in CS 61B so far? (If everything has been easy, there still is a “hardest” topic or technique.) Please be specific. In response to two of your classmates’ posts, describe strategies that you have tried or would try to understand the difficult topics they mention. Again, please be specific.

8.6.3 (Discussion Forum) Important parts of the case study
In your post, list what you believe are the three most important things to be learned from the “Bowling Scores” case study. Then respond to a labmate who chose different things to learn from the case study by commenting on the differences between your lists.

8.6.4 (Discussion Forum) Comments on the design and development of the Bowling Scores program
If you’re an EECS major, pick some aspect of the design and development of the “Bowling Scores” program that you would have done differently or that you disagree with. Describe this in a post, and defend your alternate approach or opinion. If you’re not an EECS major, pick some aspect of the design and development of the “Bowling Scores” program that you thought was particularly good or that you learned a lot from. Describe this in a post. Then argue with someone on the other side. If he or she disagrees with some aspect of the case study, defend it; if he or she found some part of the design/development noteworthy and you didn’t agree with that, provide a response that supports your opinion.

10 Linked lists and algorithm analysis

10.1 Work with linked lists

10.1.1 (Display page) What a linked list is

What a linked list is
Early in CS 61BL, we considered an Account class whose instance variables included another Account object (the account providing overdraft protection). What we were really building back then was a linked list of accounts. Any object with an instance variable of the same class creates a linked structure, with that instance variable providing the “link”. An object with a link is often referred to as a node of the linked list.

10.1.2 (Display page) A one-node circularly linked structure

A one-node circularly linked structure
Create a class with a link, then construct a node that points to itself. Use the debugger to verify that the node points to itself. (Your Account class may be useful here.)

10.1.3 (Display page) Scheme lists

http://spring08.ucwise.org/builder/builderPortal.php?8UILDER_menu=curriculumSummary
Scheme lists

Lists in Scheme are linked. Each cons pair is a node; the cdr is the link. Here's an implementation.

```java
class ListNode {
    private Object myFirst;
    private ListNode myRest;

    // cons in Scheme.
    public ListNode (Object first, ListNode rest) {
        myFirst = first;
        myRest = rest;
    }

    public ListNode (Object first) {
        this (first, null);
    }

    // car in Scheme.
    public Object first () {
        return myFirst;
    }

    // cdr in Scheme.
    public ListNode rest () {
        return myRest;
    }
}
```

10.1.4 (Display page) Testing for an empty list

Testing for an empty list

A CS 61B student suggests that the `isEmpty` predicate (`null?` in Scheme) could be implemented as follows:

```java
class ListNode {
    ...
    public boolean isEmpty () {
        return (this == null);
    }
}
```

Implement this suggestion. Then write and run a `main` method that tests it with an empty list and a nonempty list.

10.1.5 (Brainstorm) Evaluating the `isEmpty` implementation

Comment on the student's implementation of `isEmpty`. Is it a good idea or a bad idea, and why?

10.1.6 (Display page) A solution to the problem

A solution to the problem

The problem is that we can't call methods with a null reference. Thus we must ensure that any list, even an empty list, is represented by a `ListNode` object. One idea is to have a trailer node at the end of each list. This node would not contain any information; it would only be used to satisfy the requirement that each list contains at least one node. Things to think about:

- Why is a trailer node needed for all lists and not just the empty list?
- What should the trailer node's `myFirst` instance variable contain?

10.1.7 (Display page) A better solution

A better solution

A trailer node is really a special kind of `ListNode`, which suggests the use of inheritance. We'll use two new classes, one named `EmptyListNode` and the other named `NonemptyListNode`, both extending the class `ListNode`. The table below displays the differences between the two.

<table>
<thead>
<tr>
<th>EmptyListNode</th>
<th>NonemptyListNode</th>
</tr>
</thead>
<tbody>
<tr>
<td>only the 0-argument constructor makes sense</td>
<td>only the 1- and 2-argument constructors make sense</td>
</tr>
<tr>
<td>isEmpty always returns true</td>
<td>isEmpty always returns false</td>
</tr>
<tr>
<td>first and rest don't make sense</td>
<td>first and rest are quite meaningful</td>
</tr>
</tbody>
</table>

What then is in the `ListNode` class? We would like all the list-processing methods to be there (eventually). That means that definitions for `isEmpty`, `first`, and `rest` should be there, since almost every list-processing method will use those. Should `myFirst` and `myLast` be there? No, because they are only relevant for a `NonemptyListNode`. That means that the actual definition for `first` and `rest` must appear in a `NonemptyListNode`, and the definition in `ListNode` is a "dummy" version whose purpose is only to satisfy the Java compiler. Java provides a mechanism for declaring such "dummy" methods, namely the `abstract` class and `abstract` methods. We use this feature both for `first` and `rest` as just explained, as well as `isEmpty`. As noted above, only `EmptyListNode` will have a 0-argument constructor (which is supplied by default), and only `NonemptyListNode` will have 1- and 2-argument constructors. The 1-argument constructor will set `myRest` to an `EmptyListNode`. The 2-argument constructor will do the same if its second argument is `null`. One last detail: both `EmptyListNode` and `NonemptyListNode` have package visibility. That is, any class in the same directory as `EmptyListNode` and `NonemptyListNode` can use the public methods and public instance variables of those two classes. Private methods and instance variables in `EmptyListNode` and `NonemptyListNode` are inaccessible from other classes as usual. Thus we have the following code (also in `~cs61b/code/ListNode.java`):

```java
abstract public class ListNode {
    abstract public Object first ( );
    abstract public ListNode rest ( );
    abstract public boolean isEmpty ( );

    // Every other list-processing method goes here.
}
class NonemptyListNode extends ListNode {
    ```
private Object myFirst;
private ListNode myRest;

// cons in Scheme.
public NonemptyListNode (Object first, ListNode rest) {
    myFirst = first;
    if (rest == null)
        myRest = new EmptyListNode ( );
    else {
        myRest = rest;
    }
}

public NonemptyListNode (Object first) {
    this (first, new EmptyListNode ( ));
}

// car in Scheme.
public Object first ( ) {
    return myFirst;
}

// cdr in Scheme.
public ListNode rest ( ) {
    return myRest;
}

public boolean isEmpty ( ) {
    return false;
}
}

class EmptyListNode extends ListNode {
    public EmptyListNode ( ) {
    }
    public Object first ( ) {
        throw new IllegalArgumentException ("EmptyListNode constructor takes no arguments.");
    }
    public ListNode rest ( ) {
        throw new IllegalArgumentException ("EmptyListNode constructor takes no arguments.");
    }
    public boolean isEmpty ( ) {
        return true;
    }
}

Incidentally, Objects, Abstraction, Data Structures, and Design present yet another way to organize a linked list, using a "wrapper" class. We will work next week with lists organized in this way.

10.1.8 (Display page) The size method

The size method

Write a JUnit test method to test the constructors for EmptyListNode, NonemptyListNode, and the ListNode.size method (which works the same as length in Scheme). Then write the ListNode.size method.

10.1.9 (Display page) The get method

The get method

The ListNode.get method (list-ref in Scheme) takes an int position as argument, and returns the list element at the given position in the list. If the position is out of range, get should throw IllegalArgumentException with an appropriate error message. Add the get method to your ListNode class, and provide and run JUnit tests to provide evidence of its correctness.

10.1.10 (Display page) The toString method

The toString method

The ListNode.toString method returns the String representation of this list, namely:

1. a left parenthesis, followed by a blank,
2. followed by the first element, followed by a blank,
3. followed by the second element, followed by a blank,
4. ...
5. followed by a right parenthesis.

The empty list is represented by the string "( )". The list containing the Integer objects 1, 3, and 5 is represented by the string "( 1 3 5 )". Write the ListNode.toString method, and provide and run JUnit tests to provide evidence of its correctness.

10.1.11 (Display page) The equals method

The equals method

The ListNode.equals method, given a ListNode as argument, returns true if this list and the argument list are the same length with equal elements in corresponding positions (determined by using the elements’ equals method). Write the ListNode.equals method, and provide and run JUnit tests to provide evidence of its correctness.

10.2 Estimate a program’s efficiency by timing it.

10.2.1 (Display page) Time estimates

Time estimates

Much of our subsequent work in this course will involve estimating program efficiency and differentiating between "fast" algorithms and "slow" algorithms. This set of activities introduces an approach to making these estimates. At first glance, it seems that the most reasonable way to estimate the time an algorithm takes is to measure it. Each computer has an internal "clock" that keeps track of "time" (usually the number of fractions of a second that have elapsed since a given
base date); language libraries provide access to the clock. The Java method that accesses the clock is `System.currentTimeMillis`. A Timer class is provided in `~cs61b/code/Timer.java`. Take some time now to find out

- exactly what value `System.currentTimeMillis` returns, and
- how to use the Timer class to measure the time taken by a given segment of code.

### 10.2.2 (Display page) Measuring insertion sort

#### Measuring insertion sort

The file `~cs61b/code/Sorter.java` contains a version of the insertion sort algorithm you worked with a couple of weeks ago. Its `main` method uses a command-line argument to determine how many items to sort. It fills an array of the specified size with randomly generated values, starts a timer, sorts the array, and prints the elapsed time when the sorting is finished. For example, the shell command sequence

```
javac Sorter.java
java Sorter 300
```

will time the sorting of an array of 300 randomly chosen elements. Copy `Sorter.java` to your directory, compile it, and then determine the size of the smallest array that needs 1 second (1000 milliseconds) to sort. An answer within 100 elements is fine.

### 10.2.3 (Brainstorm) Timing results

What's the smallest array size (within 100) that takes at least 1 second to sort?

### 10.2.4 (Display page) Measuring a C version of insertion sort

#### Measuring a C version of insertion sort

The file `~cs61b/code/sort.c` is a version of the insertion sort algorithm coded in the language C. Its `main` function does pretty much the same as the `main` method of `Sorter.java`, except that it prints elapsed time in seconds rather than milliseconds. First, verify the similarity of the C and Java program versions. Then copy `sort.c` to your directory and compile it with the command

```
gcc sort.c
```

This produces a file named `a.out`. You then run the program by giving the command `a.out` followed by the size of the array you want to sort, for example,

```
a.out 100
```

to sort an array of 100 values. Run the program, using the array size you just found with `Sorter.java`, to see how long the C version takes.

### 10.2.5 (Display page) Measuring an optimized C version

#### Measuring an optimized C version

The C compiler includes an optimizer, which analyzes the program in more depth to produce more efficient compiled code. You access the optimizer using the command-line option `"-O2"` (that's "minus upper-case Oh 2"):

```
gcc -O2 sort.c
```

The compiler again produces a file named `a.out` that you can use in a command, e.g.

```
a.out 100
```

to sort an array of 100 elements. Conduct the same experiment you did before, finding the smallest array size that needs at least a second to sort. Limit yourself to array sizes that are divisible by 1000.

### 10.2.6 (Brainstorm) Conclusions

What's the problem with estimating a program's efficiency by measuring the clock time it takes to execute?

#### 10.3 Analyze the running time of a program.

##### 10.3.1 (Display page) Statement counting

**Statement counting**

An alternative to timing a program segment is statement counting. The number of times a given statement in a program gets executed is independent of the computer on which the program is run and is probably the same for programs coded in closely related languages. We make some simplifying assumptions here. One is that each assignment statement and test that doesn't involve a method call counts as 1. For example, in the program segment

```
int a = 4 * values[25];
int b = 9 * (a - 24) / (values[1] - 7);
if (a > 0) ...
```

each assignment statement counts as 1 and the `if` test counts as 1. We will also generally ignore statements not directly relevant to a computation, for example, `print` statements. Some easy program segments to analyze:

- A single simple statement or test counts as 1.
- The count for a simple statements that don't involve conditionals, loops, or method calls is 1.
- The count for a method call is the count for evaluating the arguments, plus the count for the body of the method.

### 10.3.2 (Display page) Counting conditionals

**Counting conditionals**

With a conditional statement like `if`, the statement count depends on the outcome of the test. For example, in the program segment

```
if (a > b) {
    temp = a;
    a = b;
    b = temp;
}
```
there are four statements executed (the test and three assignments) if \(a > b\) and only one statement executed if \(a \leq b\). That leads us to consider two quantities: the worst case count, namely the maximum number of statements that can be executed, and the best case count, the minimum number of statements that can be executed. The worst case count for the program segment above is 4 and the best case count is 1. In general, we'll refer to the best-case statement count for a program segment \(s\) as \(\text{bestcount}(s)\) and the worst-case statement count correspondingly as \(\text{worstcount}(s)\).

10.3.3 (Self Test) if ... then ... else counting
Consider an if ... then ... else of the form
\[
\begin{array}{l}
\text{if (A) \{ B; \}} \\
\text{else \{ C; \}} \\
\end{array}
\]
where \(A\), \(B\), and \(C\) are program segments. \((A\) might be a method call\). Give an expression for the best-case statement count for the if ... then ... else in terms of \(\text{bestcount}(A), \text{bestcount}(B)\), and \(\text{bestcount}(C)\).

10.3.4 (Brainstorm) Statement counting with a loop
Consider the following program segment.
\[
\text{for (int } k=0; k<N; k++) \{
\text{sum = sum + values[k];}
\}
\]
In terms of \(N\), how many operations are executed in this loop? You should count 1 for each of the actions in the for-loop header (the initialization, the test, and the increment). Briefly explain your answer. Then correct it if necessary after you review your labmates' responses.

10.3.5 (Display page) A slightly more complicated loop
A slightly more complicated loop
Now consider the code for \(\text{IntSequence.remove}\):
\[
\text{void remove (int pos) \{}
\text{for (int } k=pos+1; k<\text{myCount;} k++) \{
\text{myValues[k-1] = myValues[k];}
\}
\text{myCount--;}
\}
\]
The counts here depend on \(\text{pos}\), and are displayed in the table below.

<table>
<thead>
<tr>
<th>category</th>
<th>pos=0</th>
<th>pos=1</th>
<th>pos=2</th>
<th>pos=myCount-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment to (k)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>loop tests</td>
<td>\text{myCount}</td>
<td>\text{myCount-1}</td>
<td>\text{myCount-2}</td>
<td>1</td>
</tr>
<tr>
<td>adds to (k)</td>
<td>\text{myCount-1}</td>
<td>\text{myCount-2}</td>
<td>\text{myCount-3}</td>
<td>0</td>
</tr>
<tr>
<td>assignments to array elems</td>
<td>\text{myCount-1}</td>
<td>\text{myCount-2}</td>
<td>\text{myCount-3}</td>
<td>0</td>
</tr>
<tr>
<td>assignment to (\text{myCount})</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

We can summarize these results as follows: a call to \(\text{remove}\) with argument \(\text{pos}\) requires
- \(1\) assignment \(k\);
- \(\text{myCount-pos}\) loop tests;
- \(\text{myCount-pos}\) increments of \(k\);
- \(\text{myCount-pos}\) assignments to \(\text{myValues}\) elements;
- \(1\) assignment to \(\text{myCount}\).

If all these operations take roughly the same amount of time, the total is \(3*(\text{myCount-pos})\).

10.3.6 (Display page) Statement counting in nested loops
Statement counting in nested loops
To count statements in nested loops, one computes inside out. As an example, we'll consider an implementation of \(\text{removeZeroes}\) from an earlier quiz:

```java
public void removeZeroes ( ) {
for (int \(k=0; k<\text{myCount}; k++\) ) {
if (\text{myValues}[k] == 0) {
\text{remove } (k);
} else {
\text{\(k = k+1;\)}
}
}
```
Here, there is a best case—no removals at all—and a worst case, removing everything. Again, we create a table:

<table>
<thead>
<tr>
<th>category</th>
<th>best case</th>
<th>worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{init}s) (k)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(\text{tests of } k)</td>
<td>\text{myCount}</td>
<td>\text{myCount+1}</td>
</tr>
<tr>
<td>array accesses</td>
<td>\text{myCount}</td>
<td>\text{myCount}</td>
</tr>
<tr>
<td>comparisons</td>
<td>\text{myCount}</td>
<td>\text{myCount}</td>
</tr>
<tr>
<td>removals</td>
<td>0</td>
<td>\text{myCount}</td>
</tr>
</tbody>
</table>

The only thing left to analyze is the total cost of the calls to \(\text{remove}\) in the worst case: the sum of the cost of \(\text{remove}(0), \text{myCount}\) times. We already approximated the cost of a call to \(\text{remove}\) for a given \(\text{pos}\) and \(\text{myCount}\) value; that's \(3*(\text{myCount-pos})-1\). In our removals, \(\text{pos}\) is always 0, and only
myCount is changing. Thus the total cost of removals is
\[3 \cdot \text{myCount} + 3 \cdot \text{myCount} - 1 + 3 \cdot \text{myCount} - 2 + \ldots + 3 \cdot 1 = 3 \cdot (\text{myCount}^2 + \text{myCount} - \text{myCount} - 2 + \ldots + 1)\]
where myCount is the original number of integers in the IntSequence. A handy formula to remember is
\[1 + 2 + \ldots + k = \frac{k(k+1)}{2}\]
This lets us simplify the cost of removals:
\[= 3 \cdot \text{myCount}^2/2\]
Whew!

10.3.7 (Display page) Abbreviated estimates

Abbreviated estimates

Producing those statement count figures for even so simple a program segment was a lot of work. Normally we don't need so exact a count, but merely an estimate of how many statements will be executed. The most commonly used estimate is that a program segment runs in time proportional to a given expression, where that expression is as simple as possible. The term "proportional to ..." means "is approximately a constant times ...". Thus \(2n^3\) is proportional to \(n\) since it's approximately 2 times \(n\), and \(3n^2 + 19n + 15\) is proportional to \(n^2\) since it's approximately 3 times \(n^2\). (The approximation is better for larger values of \(n\).) Basically what we're doing here is discarding all but the most meaningful term of the estimate and also discarding any constant factor of that term. Applying this estimating technique to the removeZeros method just analyzed results in the following estimates:

- The best case running time of removeZeros is proportional to the length of the array.
- The worst case running time of removeZeros is proportional to the square of the length of the array.

This is because the actual statement count, \(4 \cdot \text{myCount}^2 + 2 \cdot \text{myCount} + \text{myCount}^2/2\), is proportional to \(\text{myCount}^2\) in the best case, and \(3 \cdot \text{myCount}^2 + 2 \cdot \text{myCount} + \text{myCount}^2/2\) is proportional to the square of \(\text{myCount}\) in the worst case. A notation often used to provide even more concise estimates is "big-Omega" (\(\Omega\)). Given two functions \(f(n)\) and \(g(n)\) that, given nonnegative integers, return real values, we have
\[f(n) = \Omega(g(n))\]
if and only if \(f(n)\) is proportional to \(g(n)\) for all large enough values of \(n\). There is an analogous notation called "big-Oh". In CS 61B, however, we will try to stick to the "proportional to" notation.

10.3.8 (Display page) The variables in the "proportional to" expression

The variables in the "proportional to" expression

You may have observed, in our analysis of remove, that we were careful to make clear what the running time estimate depended on, namely the value of \(\text{myCount}\) and the position of the removal. Unfortunately, students are sometimes careless about specifying the quantity on which an estimate depends. Don't just use "\(n\)" without making clear what "\(n\)" means.

10.3.9 (Brainstorm) Choosing the proportionality variable

Complete the following sentence. Insertion of an integer at position \(k\) of an IntSequence containing \(n\) items, with capacity greater than \(n\), is proportional to \(\ldots\). Correct your answer after reviewing the submissions of your labmates.

10.3.10 (Display page) Commonly occurring estimates

Commonly occurring estimates

Some commonly occurring estimates and situations in which they arise are the following:

- constant time: a single statement or a constant number of elementary operations;
- "proportional to (suitably defined) \(N\)" applying a constant-time process to each of a collection of \(N\) items;
- "proportional to \(N^2\)" applying a constant-time process to all pairs of items in a collection of \(N\) items.

10.3.11 (Brainstorm) Analyzing insertion sort

Give best-case and worst-case estimates of the running time of insertion sort. (Assume that \(N\) is the length of the array being sorted.) Briefly explain your answers.

10.3.12 (Display page) Logarithmic algorithms

One more proportionality expression

We will shortly encounter algorithms that run in time proportional to \(\log x\) for some suitable defined \(x\). Recall from algebra that the base-10 logarithm of a value is the exponent to which 10 must be raised to produce the value. It is usually abbreviated as \(\log_{10}\). Thus
\[\log_{10} 1000 = 3\]
\[\log_{10} 90 \text{ is slightly less than } 2\]
\[\log_{10} 1 \text{ is } 0\]

In algorithms, we commonly deal with the base-2 logarithm, defined similarly.
\[\log_2 1024 = 10\]
\[\log_2 9 \text{ is slightly more than } 3\]
\[\log_2 1 \text{ is } 0\]

Algorithms for which the running time is logarithmic are those where processing discards a large quantity of values in each iterations. The binary search algorithm you encountered a few weeks back (in the "guess a number" game) is an example. In each iteration, the algorithm discards half the possible values for the searched-for number. We will see further applications of logarithmic behavior when we work with trees in subsequent activities.
Discussions + project work

Provide a post and a response to each of the three discussion activities that follow. Also, make some progress on project 1, by completing at least two of the lab's methods addAll, removeAll, containsAll, and isOK, along with a JUnit test suite for those methods. Your lab t.a. will verify your progress and award up to 5 homework points for your efforts.

10.4.2 (Discussion Forum) ListNode confusion?
In a post (by Sunday night), supply a question or misconception that a CS 61BL student might have about the representation of linked lists described in today's lab (involving the classes EmptyListNode and NonemptyListNode). Then, provide a response by answering one of your labmates' questions or by significantly elaborating on an answer already provided.

10.4.3 (Discussion Forum) Project 1 confusion?
In a post (by Sunday night), make a guess about what aspect of project 1 will be most difficult, and provide a brief explanation. Then, in response to one of your labmates, suggest a way to deal with or simplify the difficult aspect of the project that he or she described.

11.2.2 (Display page) Project progress by Thursday or Friday

Your lab t.a. will expect to see more progress on the project by Thursday or Friday. (You will have time in lab to work on the project.) He or she will award up to 5 homework points for your progress, according to the following guidelines:

- 1 – You have some project-related code.
- 2 – You have one of addAll, removeAll, containsAll, and isOK ready to test but no JUnit tests, or you have a thorough JUnit test suite for one of these methods.
- 3 – You have one of addAll, removeAll, containsAll, and isOK almost completely coded, together with a thorough JUnit test suite for one of those methods.
- 4 – You have the above, plus an almost-complete version of one of the other methods or a thorough JUnit test suite for one of the other methods.
- 5 – You have almost-complete versions (possibly with a bug or two) of at least two of addAll, removeAll, containsAll, and isOK, together with almost-complete JUnit test suites for two of these methods.

11.2.1 (Display page) Project progress by the start of lab

Project progress by the start of lab

You should have done some work on project 1 over the weekend. Your lab t.a. will award up to 5 homework points for your progress, according to the following guidelines:

- 1 – You have some project-related code.
- 2 – You have one of addAll, removeAll, containsAll, and isOK ready to test but no JUnit tests, or you have a thorough JUnit test suite for one of these methods.
- 3 – You have one of addAll, removeAll, containsAll, and isOK almost completely coded, together with a thorough JUnit test suite for one of these methods.
- 4 – You have the above, plus an almost-complete version of one of the other methods or a thorough JUnit test suite for one of the other methods.
- 5 – You have almost-complete versions (possibly with a bug or two) of at least two of addAll, removeAll, containsAll, and isOK, together with almost-complete JUnit test suites for two of these methods.

11.2.3 (Display page) Project progress by Thursday or Friday

Project progress by Thursday or Friday

Your lab t.a. will also expect to see even more progress on the project by the end of lab on Thursday or Friday. (You will have time in lab to work on the project.) He or she will award up to 5 more homework points for your progress, according to the following scale:

- 1 – You have one of addAll, removeAll, containsAll, and isOK coded and tested.
- 2 – You have two of addAll, removeAll, containsAll, and isOK coded, plus thorough JUnit test suites for one of these methods, or one of the methods just listed plus two JUnit test suites.
- 3 – You satisfy the requirement for last Tuesday/Wednesday (two of addAll, removeAll, containsAll, and isOK coded and debugged, plus thorough JUnit test suites for two of these methods).
- 4 – You have the above, plus either a third almost-complete method in the above group or a third JUnit test suite.
- 5 – You have at least three of addAll, removeAll, containsAll, and isOK coded and debugged, plus thorough JUnit test suites for three of these methods.

You should have time to do some work on project 1 after the scheduled exercises in lab today.

11.3 Explore the use of static list processing methods

11.3.1 (Display page) A smallest-element method

Aaa smallest-element method

In the previous lab, you coded several useful methods for dealing with lists. For lists whose items implement the Comparable interface (mentioned in "Learn about Java interfaces" on February 12 and 13, methods that return the largest and smallest list item would also be useful. In Scheme, one way to find the smallest item in a list uses a "helper" function:

```scheme
(define (smallest L)
  (define helper (car L))
  (define (helper L minSoFar)
    (if (null? L) minSoFar
      (helper (cdr L) (min (car L) minSoFar)) ))
  (define (smallest-helper L smallestSoFar)
    (if (isEmpty L) smallestSoFar
      (smallest-helper (cdr L) (min (car L) smallestSoFar))))
  (smallest-helper L 1))
```

In the previous lab, you coded several useful methods for dealing with lists. For lists whose items implement the Comparable interface (mentioned in "Learn about Java interfaces" on February 12 and 13, methods that return the largest and smallest list item would also be useful. In Scheme, one way to find the smallest item in a list uses a "helper" function:

```scheme
(define (smallest L)
  (define helper (car L))
  (define (helper L minSoFar)
    (if (null? L) minSoFar
      (helper (cdr L) (min (car L) minSoFar)) ))
  (define (smallest-helper L smallestSoFar)
    (if (isEmpty L) smallestSoFar
      (smallest-helper (cdr L) (min (car L) smallestSoFar))))
  (smallest-helper L 1))
```

A smallest-element method

Implement the smallest method with parameters that tests for two of these methods.

```java
public Comparable smallest ()
{
    if (isEmpty () )
    {
        throw new NoSuchElementException( "can't find smallest in empty list" );
    }
    return ______ ;
}
```

In the previous lab, you coded several useful methods for dealing with lists. For lists whose items implement the Comparable interface (mentioned in "Learn about Java interfaces" on February 12 and 13, methods that return the largest and smallest list item would also be useful. In Scheme, one way to find the smallest item in a list uses a "helper" function:

```scheme
(define (smallest L)
  (define helper (car L))
  (define (helper L minSoFar)
    (if (null? L) minSoFar
      (helper (cdr L) (min (car L) minSoFar)) ))
  (define (smallest-helper L smallestSoFar)
    (if (isEmpty L) smallestSoFar
      (smallest-helper (cdr L) (min (car L) smallestSoFar))))
  (smallest-helper L 1))
```

A smallest-element method

In the previous lab, you coded several useful methods for dealing with lists. For lists whose items implement the Comparable interface (mentioned in "Learn about Java interfaces" on February 12 and 13, methods that return the largest and smallest list item would also be useful. In Scheme, one way to find the smallest item in a list uses a "helper" function:

```scheme
(define (smallest L)
  (define helper (car L))
  (define (helper L minSoFar)
    (if (null? L) minSoFar
      (helper (cdr L) (min (car L) minSoFar)) ))
  (define (smallest-helper L smallestSoFar)
    (if (isEmpty L) smallestSoFar
      (smallest-helper (cdr L) (min (car L) smallestSoFar))))
  (smallest-helper L 1))
```

A smallest-element method

In the previous lab, you coded several useful methods for dealing with lists. For lists whose items implement the Comparable interface (mentioned in "Learn about Java interfaces" on February 12 and 13, methods that return the largest and smallest list item would also be useful. In Scheme, one way to find the smallest item in a list uses a "helper" function:

```scheme
(define (smallest L)
  (define helper (car L))
  (define (helper L minSoFar)
    (if (null? L) minSoFar
      (helper (cdr L) (min (car L) minSoFar)) ))
  (define (smallest-helper L smallestSoFar)
    (if (isEmpty L) smallestSoFar
      (smallest-helper (cdr L) (min (car L) smallestSoFar))))
  (smallest-helper L 1))
```

A smallest-element method

In the previous lab, you coded several useful methods for dealing with lists. For lists whose items implement the Comparable interface (mentioned in "Learn about Java interfaces" on February 12 and 13, methods that return the largest and smallest list item would also be useful. In Scheme, one way to find the smallest item in a list uses a "helper" function:

```scheme
(define (smallest L)
  (define helper (car L))
  (define (helper L minSoFar)
    (if (null? L) minSoFar
      (helper (cdr L) (min (car L) minSoFar)) ))
  (define (smallest-helper L smallestSoFar)
    (if (isEmpty L) smallestSoFar
      (smallest-helper (cdr L) (min (car L) smallestSoFar))))
  (smallest-helper L 1))
```

A smallest-element method

In the previous lab, you coded several useful methods for dealing with lists. For lists whose items implement the Comparable interface (mentioned in "Learn about Java interfaces" on February 12 and 13, methods that return the largest and smallest list item would also be useful. In Scheme, one way to find the smallest item in a list uses a "helper" function:

```scheme
(define (smallest L)
  (define helper (car L))
  (define (helper L minSoFar)
    (if (null? L) minSoFar
      (helper (cdr L) (min (car L) minSoFar)) ))
  (define (smallest-helper L smallestSoFar)
    (if (isEmpty L) smallestSoFar
      (smallest-helper (cdr L) (min (car L) smallestSoFar))))
  (smallest-helper L 1))
```
public static Comparable min (Comparable c1, Comparable c2) {
    if (c1.compareTo (c2) < 0) {
        return c1;
    } else {
        return c2;
    }
}

11.3.2 (Brainstorm) Why static min?
The code for min is repeated below. Why is min a static method?

public static Comparable min (Comparable c1, Comparable c2) {
    if (c1.compareTo (c2) < 0) {
        return c1;
    } else {
        return c2;
    }
}

11.3.3 (Display page) Helpful static helper methods

Helpful static helper methods
It is convenient to declare the minHelper method as static:

public Comparable smallest ( ) {
    if (isEmpty ( )) {
        throw new NoSuchElementException ("can't find smallest in empty list");
    }
    return minHelper (rest ( ), first ( ));
}

public static Comparable minHelper (ListNode list, Comparable smallestSoFar) {
    if (list.isEmpty ( )) {
        return smallestSoFar;
    } else {
        return minHelper (list.rest ( ), min (smallestSoFar, list.first ( )));
    }
}

public static Comparable min (Comparable c1, Comparable c2) {
    if (c1.compareTo (c2) < 0) {
        return c1;
    } else {
        return c2;
    }
}

11.3.4 (Brainstorm) Benefit from a static helper
What's the advantage of making minHelper a static method?

11.4 Practice coding more ListNode methods.

11.4.1 (Display page) An add method
An add method
Write an ListNode.add method that, given an object x as argument, returns a new list with the following properties.

- If this list has n elements, the returned list should have n+1. The first n elements should be the same as the first n elements of this list; the last element in the returned list should be x.

Here's a method to add to your JUnit tests from last lab.

```java
public void testAdd ( ) {
    ListNode l1 = new EmptyListNode ( );
    ListNode l2 = l1.add ("a");
    assertEquals ("( a )", l2.toString());
    ListNode l3 = l2.add("b");
    assertEquals ("( a b )", l3.toString());
}
```

Here for your reference is a Scheme version of add:

```scheme
(define (add L x)
    (if (null? L) '(x)
        (cons (car L) (add (cdr L) x)) )
)
```

11.4.2 (Display page) An append method
An append method
Write and test a ListNode.append method that, given a ListNode as argument, returns the result of concatenating a copy of the list represented by this ListNode and the list represented by the argument. For example, appending the lists ( 1 2 3 ) and ( 4 5 6 ) should return the list ( 1 2 3 4 5 6 ). Your JUnit method should test at least four cases:

- this list is empty;
- the argument is empty;
- neither are empty;
- the argument and this list are the same.

Here for your reference is a Scheme version of append:

```scheme
(define (append L1 L2)
    (if (null? L1) L2
        (cons (car L1) (append (cdr L1) L2)) )
)
```

11.4.3 (Brainstorm) Recursion vs. iteration
So far you've coded seven methods for the ListNode class:

- size
- get
- toString
- equals
- contains
- add
- append

For each method, indicate whether you coded it recursively or iteratively (i.e. with no recursive call).
11.4.4 (Display page) Recursive list processing

Recursive list processing

Recursive list processing is generally more convenient than its iterative counterpart, especially in applications that require building new list structure. (Don’t forget what you learned about recursion in CS 61A or whatever you took as prerequisite to this course!) We believe that it’s not possible to define iterative versions of copy and append without using an explicit stack—we’ll get to stacks soon—or introducing set methods into the ListNode class.

11.4.5 (Display page) A reverse method

A reverse method

Write a ListNode.reverse method that returns a copy of this list with the elements in reverse order, and test it with JUnit. Here for your reference is a Scheme version of reverse:

```scheme
(define (reverse L)
 (reverse-helper L '( ))
)
(define (reverse-helper L so-far)
 (if (null? L) so-far
     (reverse-helper (cdr L) (cons (car L) so-far)) )
)
```

11.5 Work on project 1. (1 step)

11.5.1 (Display page) Project 1 work

Project 1 work

Once you finish the other activities for today, you may work on your project. A reminder: this project is an individual effort. You may not show any of your code to any classmate, or view any of your classmates’ code.

12 Destructive list manipulation; a wrapper class 2008-2-28 ~ 2008-2-29 (4 activities)

12.2 Add destructive list operations to the ListNode class. (10 steps)

12.2.1 (Display page) A source of inefficiency

A source of inefficiency

Three of the methods coded last lab, add, append, and reverse, all made calls to new in the process of producing their answers. However, these calls can be wasteful when we don’t mind modifying a data structure in place. Even though Java reclaims unused storage, the process of doing so takes time, and under some circumstances the effects of needlessly generating and reclaiming nodes can be noticeable and even an efficiency bottleneck. In the next few steps, we’ll explore and evaluate the option of destructive list modification, that is, changing pointers without generating any new nodes.

12.2.2 (Self Test) Nodes generated for append

Here is code for the ListNode.append method.

```java
public ListNode append (ListNode list) {
    return appendHelper (this, list);
}

private static ListNode appendHelper (ListNode list1, ListNode list2) {
    if (list1.isEmpty()) {
        return list2;
    } else {
        return new NonemptyListNode (list1.first(), appendHelper (list1.rest(), list2));
    }
}
```

How many calls to new are made as a result of the call `list1.append(list2)`?

12.2.3 (Display page) Desired behavior of append

Desired behavior of append

The call `list3 = list1.append(list2)` using the version of append coded in the last lab behaved as shown below. Before the call

```
Before the call

list1 = 1 —- 2 —- EMPTY
```

```
After the call

list1 = 1 —- 2 —- 3 —- 4 —- EMPTY
```

We want

```
After the call

list1 = 1 —- 2 —- 3 —- 4 —- EMPTY
```

the new append to produce the following: After the call

```
list1 = 1 —- 2 —- 3 —- 4 —- EMPTY
```
The code from last lab did not allow the `myFirst` and `myRest` instance variables to be changed once initialized; every assignment to either variable came during the construction of a new `ListNode`. With the addition of two `mutator` methods, which we'll name `setFirst` and `setRest`, we will be able to replace the `myFirst` variable with a reference to any object and replace `myRest` with a reference to any `ListNode`.

### 12.2.5 (Brainstorm) Declarations of setFirst and setRest?

Should `setFirst` and `setRest` be declared in the `ListNode` class, the `EmptyListNode` class, the `NonemptyListNode` class, two of the classes, or all three of the classes? Briefly explain your answer.

### 12.2.6 (Display page) The appendInPlace method

Write the `ListNode.appendInPlace` method. It is given a `ListNode` reference `list2`. There are two cases:

- If this list is empty, `list2` is returned.
- If this list isn't empty, the rest in its last `NonemptyListNode` is replaced by `list2`.

Here are some straightforward test cases.

```java
public void testStraightforward ( ) {
    ListNode empty1 = new EmptyListNode ( );
    ListNode empty2 = new EmptyListNode ( );
    empty1 = empty1.appendInPlace (empty2);
    assertEquals ("( )", empty1.toString ( ));
    assertEquals ("( )", empty2.toString ( ));
    ListNode a = new NonemptyListNode ("a");
    a = a.appendInPlace (empty1);
    assertEquals ("( a )", a.toString ( ));
    assertEquals ("( )", empty1.toString ( ));
    ListNode b = new NonemptyListNode ("b");
    ListNode c = new NonemptyListNode ("c");
    b = b.appendInPlace (c);
    assertEquals ("( b c )", b.toString ( ));
    assertEquals ("( c )", c.toString ( ));
}
```

### 12.2.9 (Brainstorm) Test results?

What happened when you ran the test in the preceding step? Provide appropriate details.

### 12.2.10 (Brainstorm) Reason for appendInPlace behavior?

Describe the box-and-arrow diagram that results from the program segment

```java
ListNode list1 = new EmptyListNode ( );
ListNode list2 = new NonemptyListNode ("a");
list1 = list1.appendInPlace(list2);
```

### 12.3 Work with a List wrapper class.

(10 steps)

12.3.1 (Display page) A wrapper class that protects list integrity

A wrapper class that protects list integrity

It's clear, we hope, that allowing any method to call `setRest` to change the links in a list invites disaster. The solution is to wrap the `ListNode` class inside another class, which we'll call `List`. The `List` class will have a `ListNode` instance variable—we'll call it `myHead`—that refers to the first node in the linked list. Here's an outline of the class:

```java
public class List {
    private ListNode myHead;

    public List ( ) { }
    private static class ListNode {
        private Object myFirst;
        private ListNode myRest;
    }
}
```
The List class will provide most of the methods previously provided by the ListNode class, but access to myFirst and myRest will not be allowed outside the List class. Normally, one must have an instance of the outer class in order to instantiate a nested class. ListNode class is declared as static because there are occasions where (briefly) we will want to create a ListNode object that's not connected to any List object.

12.3.2 (Display page) An extra bonus

An extra bonus

We had previously encountered a problem because we didn't have an object to represent an empty list. The wrapper class List has now solved this problem; the empty list can be represented by a List object whose myHead instance variable contains null.

12.3.3 (Display page) Some code for the List class

Some code for the List class

Some code for the List class, also in ~cs61b/code/List.java, appears below. The file ~cs61b/code/ListTest.java contains a JUnit test suite. You may observe that more of the methods are iterative as opposed to recursive, and in particular that they fit into a pattern:

```
for (ListNode p = myHead; p != null; p = p.myRest) ...
```

Here's the code.

```java
public class List {
    private ListNode myHead;

    public List () {
        myHead = null;
    }

    public boolean isEmpty () {
        return myHead == null;
    }

    private static class ListNode {
        private Object myFirst;
        private ListNode myRest;

        private ListNode (Object first, ListNode rest) {
            myFirst = first;
            myRest = rest;
        }

        private ListNode (Object first) {
            myFirst = first;
            myRest = null;
        }
    }

    public String toString () {
        String rtn = "[");
        for (ListNode p = myHead; p != null; p = p.myRest) {
            rtn = rtn + p.myFirst + " ");
        }
        return rtn + "]");
    }

    // Return the number of items in this list ("length" in Scheme).
    public int size () {
        int rtn = 0;
        for (ListNode p = myHead; p != null; p = p.myRest) {
            rtn++;
        }
        return rtn;
    }

    // Return a reference to the sub-list of this list that begins
    // with the given element: returns the empty list if this list
    // doesn't contain the given element.
    public boolean contains (Object obj) {
        for (ListNode p = myHead; p != null; p = p.myRest) {
            if (obj.equals (p.myFirst)) {
                return true;
            }
        }
        return false;
    }

    // Returns the element at the given position in this list.
    public Object get (int pos) {
        if (pos < 0) {
            throw new IllegalArgumentException("Argument to get must be at least 0. ");
        } else if (pos >= size()) {
            throw new IllegalArgumentException("Argument to get is too large.");
        }
        int k = 0;
        for (ListNode p = myHead; p != null; p = p.myRest) {
            if (k == pos) {
                return p.myFirst;
            }
            k++;
        }
        return null;
    }

    public void addToFront (Object obj) {
        myHead = new ListNode (obj, myHead);
    }
}
```

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
12.3.4 (Brainstorm) Box-and-arrow representation
Which of the following box-and-arrow diagrams most accurately represents the one-element list referred to by the variable $L_1$? Briefly explain your answer.

12.3.5 (Display page) The equals method
The equals method
Write the List.equals method, which determines if this list contains the same elements in the same sequence as the argument list. The JUnit test suite in ~cs61b/code/ListTest.java contains a test method.

12.3.6 (Display page) The add and appendInPlace methods
The add and appendInPlace methods
Write the List.add method, which adds its argument as the last element of this list. For instance, if this list contains the elements "a", "b", and "c", then the result of adding "d" is the list containing the elements "a", "b", "c", and "d", in that order. Then write the List.appendInPlace method (it will be similar in structure to List.add). Don’t create any new nodes to do the appending. The JUnit test suite in ~cs61b/code/ListTest.java contains tests for both these methods.

12.3.7 (Display page) Iterator methods
Iterator methods
Complete the methods for the List iterator. Framework code is below; it resembles the iterator implementation that you will have to provide for project 1. The ListTest.java file contains a test method.

12.3.8 (Display page) Opportunities for efficiency
Opportunities for efficiency
A principle of organizing data for efficient access is the following:

If you need to access information quickly, store it (or a reference to it) somewhere.

The List class provides an opportunity to apply this principle. In particular, both add and appendInPlace currently search for the last node in the list, and size does essentially the same thing to count the nodes in the list. To save time, we can provide two more instance variables; one, an int named mySize, will store the length of this list, and the other, a ListNode named myTail, will store a reference to the last node in this list. (Both myHead and myTail will contain null if this list is empty.) Modify the size, add, addToFront, and appendInPlace methods to accommodate these two new variables. The ListTest.java tests should be appropriate tests for your modifications.
12.3.9 (Display page) A consistency checker

A consistency checker

Write a List method named isOK that checks this list for consistency. In particular, it should check that

- the value stored in mySize is the number of nodes in this list;
- all myFirst objects in this list are nonnull;
- either both myHead and myTail are null or myTail is a reference to the last node in the list whose first node myHead refers to.

(These are data invariants.) The isOK method does the clear-box testing that's not possible with JUnit.

12.3.10 (Display page) A remove method

A remove method

Write and test List method named remove that, given an Object as argument, removes elements that are equal to that object from the list.

12.4 Homework (1 step)

12.4.1 (Display page) Reading + project

Reading + project

We'll be covering trees next week, sections 8.1 through 8.3 in Objects, Abstraction, Data Structures, and Design. Project 1 is due this coming Monday night (March 3). You have a total of 72 "slip hours" to apply toward late project submission. (You probably shouldn't use them all for this project.)

13 Finishing up linked lists 2008-3-04 ~ 2008-3-05 (4 activities)

13.2 Work through more mutation exercises (with a partner) (7 steps)

13.2.1 (Display page) "Doubling" each node in a linked list

"Doubling" each node in a linked list

Two heads will be better than one at working on today's activities. We encourage you to work today with a partner. Here is an exercise that previous 61B students have found quite difficult. You encountered it on a quiz in another context last week; it's interesting to compare the difficulty of the two exercises. Fill in the body of the loop in the List method doubleInPlace whose framework is given below. Each node in the list should be copied in place, resulting in a list that's twice as long. For example, if the list initially contains (a b c), it should contain (a a b b c c) after the call to doubleInPlace. Don't call any methods other than the ListNode constructor, and don't call it more than once per node in the original list. Here's the framework.

```java
public void doubleInPlace() {
    for (ListNode p=myHead; p!=null; p=p.myRest) {
        ________________________________ ;
        ________________________________ ;
    }
}
```

Work on this exercise, and report in the next steps what your answer is and what makes this exercise difficult.

13.2.2 (Brainstorm) "Doubling" solutions

What goes in the blanks?

```java
public void doubleInPlace() {
    for (ListNode p=myHead; p!=null; p=p.myRest) {
        ________________________________ ;
        ________________________________ ;
    }
}
```

13.2.3 (Brainstorm) Difficulty of doubling

Why is this exercise difficult?

13.2.4 (Display page) Reversing a list in place

Reversing a list in place

Now consider the problem of reversing a list. In Scheme, this was done with an auxiliary procedure:

```
(define (reverse L)
    (helper L '() )
)
```

```scheme
(define (helper L soFar)
    (if (null? L) soFar
        (helper (cdr L) (cons (car L) soFar)) )
)
```

We will assume the additional restriction that the reversal is to be done in place, that is, without creating any new ListNode. As noted in an earlier lab session, a static method will be helpful. We must also, however, pay close attention to the pointer processing necessary to reverse all the links. 13.2.5 (Brainstorm) Identifying an invariant

The Scheme version of reverse just given maintains a recursion invariant that relates the values of L and soFar at each call. Here is a trace of the helper function that results from the call

```
(reverse '(A B C D))
```

```scheme
(helper '(A B C D) '() )
```

```scheme
(helper 'B C D) 'A()
```

```scheme
(helper 'C D) '(B A)
```

```scheme
(helper 'D) 'C B A) ;
```

Identify the invariant relation between the original argument to reverse (you may name that Lorig) and the helper arguments L and soFar that is true at each call to helper.

13.2.6 (Display page) Completing the reverse method

Completing the reverse method

Whenever helper is called, L is the list of elements not yet reversed, and soFar is the reversed list
of the elements already processed. (Thus \( \text{append} \ (\text{reverse} \ \text{soFar}) \ L) = L_{\text{orig}} \). To design the procedure from this relation, we position ourselves in the middle of the recursion, say, with \( L = \text{(C D)} \) and \( \text{soFar} = \text{(B A)} \). What do we do with \( C \), the next element in \( L \), so that \( \text{soFar} \) is the reverse of \( \text{(A B C)} \)? The answer is to \text{cons} it to the current value of \( \text{soFar} \). We apply the same reasoning for the Java \text{reverseHelper} method. Here’s its header:

```java
private static ListNode reverseHelper (ListNode L, ListNode soFar) {
    if (L == null) {
        return soFar;
    } else {
        ...}
```

Again we position ourselves in the middle of the recursion. For design in Java, it helps to draw box-and-pointer diagrams; here’s the situation with \( L \) pointing to the list \( \text{(C D)} \) and \( \text{soFar} \) pointing to the list \( \text{(A B)} \), along with the structure that should be passed to the recursive call. The boxes outlined in bold are the boxes whose contents should change.

There are three changes to be made. All are interdependent, though, so we need a temporary variable:

```java
ListNode temp = L.myRest;
L.myRest = soFar;
return reverseHelper ( ____ , ____ );
```

The copying of the arguments in the recursive call will provide the other two changes to box values. Fill in the arguments of the recursive call to complete the method. Then try it out with a JUnit test.

### 13.2.7 (Display page) An iterative reverse

An iterative reverse

Fill in the framework for \text{reverseHelper} given below so that instead of making a recursive call, it uses a loop to change the relevant links. (You’ll also need to change the call to \text{reverseHelper}.) Each iteration of your loop should maintain the invariant described for the recursive version. Your JUnit test from the previous step should work for this version as well. Here’s the framework.

```java
private static ListNode reverseHelper (ListNode head) {
    ListNode p, soFar;
    // p plays the role of L in the previous version.
    for (p=head, soFar=null; p!=null; ) {
        ListNode temp = p;
        ...
    }
    return soFar;
}
```

### 13.3 Examine variations on linked lists (with a partner).

(8 steps)

#### 13.3.1 (Display page) Header nodes

Header nodes

In coding some of the \text{List} methods, you may have noticed a bit of awkwardness arising from the need to handle a modification to the start of the list differently from a modification anywhere else:

```java
if (myHead == null) {
    // update myHead
} else {
    // update some myRest field
}
```

This awkwardness can be eliminated if we guarantee that there will always be at least one node in the list. We did this in our first list representation; there, every list ended with an \text{EmptyListNode} as the first element in the list. In most circumstances, its \text{myFirst} value is ignored. (When the list elements are to be maintained in sorted order, the \text{myFirst} value would be the smallest or largest representable value.) Iterator code would have to be revised to skip the dummy node, so we trade off one area of clumsiness for another.

#### 13.3.2 (Display page) Doubly linked lists

Doubly linked lists

Another bit of awkwardness in list processing arises when inserting or deleting a list element. In these situations, the \text{myRest} variable in the \text{ListNode} object immediately prior to the node being inserted or deleted must be changed. A way to deal with this is to use a \text{doubly linked list}. Each node in the list contains not only a reference to the next node but also a reference to the immediately preceding node, as in the following diagram.
Occasionally we find it useful to have the last node in a list point back to the first node, thus creating a **circular** structure. (You created such a structure in an earlier activity.) One obviously needs to worry about infinite loops with such a structure, so processing involves saving a reference to a node in the list and then comparing it with an iteration reference to make sure we don't go past the end of the list.

### 13.3.4 Applications of circular lists

Describe a collection of data for which a circular list would be an appropriate implementation.

### 13.3.5 Managing a circular doubly-linked list

The code here, also in ~cs61b/code/Sequence.java, is an incomplete class that represents a sequence as a circular doubly-linked list. You are to supply the code for the `delete` method and the `Enumeration` methods. Comment out the existing body of `toString` and comment in the alternative implementation to test the `Enumeration` code. Hint: You will probably need additional state information in the `SequenceEnumeration` class beyond a `DListNode` variable, so that your `hasMoreElements` method will perform correctly. We suggest a counter or a `boolean` `returnedFirstElement` variable.

### 13.3.6 Timing experiments with java.util classes

You already know about the `ArrayList` class. The `java.util` library also contains a `LinkedList` class. In the next couple of steps, you'll perform some timing experiments that investigate properties of these two classes. Each experiment should involve only one or two timings (to find an `n` big enough to show a difference), along with some big-Oh analysis. You may use the `Timer` class used earlier with sorting methods (it's in ~cs61b/code/Timer.java). You may not use the `getClass` method or the `instanceof` operator.

**Experiment 1**

Suppose you know that a given `Collection` class is represented either as an `ArrayList` or as a `LinkedList`. Design an experiment to discover which of the two is used. In your experiment, time some operation on an `ArrayList` and the same operation on a `LinkedList`; the timings should differ enough to make it clear how to identify a "mystery" collection.

**Experiment 2**

The online documentation for a `LinkedList` claims that it's implemented as a doubly-linked list. Design an experiment that supports this claim. In particular, your experiment should show that the underlying implementation is not a singly-linked list with a head and tail pointer.

### 13.3.7 Experiment 1 results

Describe your experiment 1 and its results, and explain why they distinguish between an `ArrayList` and a `LinkedList`.

### 13.3.8 Experiment 2 results

Describe your experiment 2 and its results, and explain why they distinguish between a doubly linked list and a singly linked list with head and tail pointers.

### 13.4 Homework

#### 13.4.1 Reading

Trees will be covered in the next couple of labs. Read *Objects, Abstraction, Data Structures and Design*, sections 8.1 through 8.3.

**14 Trees**

2008-3-06 ~ 2008-3-07 (5 activities)

**14.2 Meet some trees**

(6 steps)

**14.2.1 Introduction to trees**

Almost all the collections of data we've dealt with so far have been sequential, or one-dimensional. If we loosen this requirement and allow elements of a collection to be collections themselves, we get a two-dimensional structure, as diagrammed below.
This collection-of-collections can be generalized to collections-of-collections-of-collections and so on. This results in a structure that for now we’ll call a **generalized collection** that can be defined recursively:

- a generalized collection is either empty, or
- it consists of N elements, each of which is a generalized collection.

We can draw this structure in a tree-like fashion—visualize the diagram above turned 45 degrees clockwise—and in fact usually refer to a generalized collection as a tree. A tree either is empty, or it has a root R with zero or more trees whose roots are the children of R. The root is the parent of each of its children, and a child has only one parent; two children of the same parent are siblings. The root's children, grandchildren, etc. are its descendants, and the root is an ancestor of its descendants. A parent together with its descendants form a subtree of the tree of which they are elements. A root without children is referred to as a leaf of the tree. A forest is a collection of disjoint trees. A Java representation of a tree typically involves two classes, one for the overall tree and one for the tree's nodes. This recalls the `List` and `ListNode` classes we have used to represent sequences.

14.2.2 (Brainstorm) Uses of trees

Name at least two applications that can be modelled as trees.

14.2.3 (Display page) Operations on trees

**Operations on trees**

Typical operations on trees are those we've seen with sequential structures: adding and deleting elements, search, and traversal or iteration. The recursive structure of a tree suggests recursive algorithms for these operations, for instance, search:

1. If the tree is empty, the search fails;
2. If the tree's root is the element we're looking for, the search succeeds;
3. Otherwise we search the subtrees.

Traversal—visiting each element of the tree—may also be done recursively. Two traversal variations differ in the order in which the root is visited vis-a-vis the descendants. A **preorder** traversal visits the root, then its descendants; a **postorder** traversal visits the root after its descendants. Traversal as just described is an all-at-once kind of operation. An alternative is to use a one-by-one iterator as we did with lists. An iterator for sequences need merely keep track of where in the sequence the next element of the iteration is; with trees, however, we need more complicated state-saving information. Exercises in upcoming labs will familiarize you with tree iterators.

A recursive search is **depth first** since it searches all descendants of a given root, no matter how many there are, before moving on to search through siblings of the root. Another kind of search is **breadth first**, where first the root is checked, then the root's children, then its grandchildren, and so on, searching tree elements in order of their distance from the root.

14.2.4 (Display page) Binary trees and general trees in Java

**Binary trees and general trees in Java**

In Java, as noted earlier, we implement trees with a `Tree` class and an inner `TreeNode` class.

```java
public class Tree {
    private TreeNode myRoot; // the root of this tree

    private static class TreeNode {
        private Object myItem; // Objects, Abstraction, Data Structures, and Design uses "data"
        private Collection myChildren; // The collection contains TreeNodes.
        // It could be a List or one of // the collection classes in java.util.
        ...
    }

    // Tree methods would go here.
}
```

Many computing applications involve **binary trees**, a special case where each node has at most two children. A `TreeNode` in a binary tree contains two references to children (the "left child" and "right child"):  

```java
private class TreeNode {
    private Object myItem;
    private TreeNode myLeft;
    private TreeNode myRight;
    ...
}
```

14.2.5 (Display page) Amoeba family trees in Java

**Amoeba family trees**
An amoeba family tree is simpler than a normal family tree because amoebas do not get married. An amoeba has one parent, perhaps a million siblings (brothers and sisters), and billions and billions of children. An amoeba also has a name. Amoebas (or amoebae) live dull lives. All they do is reproduce. So all we need to keep track of them are the following methods:

```java
// A constructor that starts an Amoeba family with an amoeba
// with the given name.
public AmoebaFamily (String name);

// Add a new amoeba named childName as the youngest child
// of the amoeba named parentName.
// Precondition: the amoeba family contains an amoeba named parentName.
public void addChild (String parentName, String childName);

// Print the amoeba family tree.
public void print ( );
```

The code for the `AmoebaFamily` class is in `~cs61b/code/AmoebaFamily.java`. Its main program is intended to construct the family shown below.

![Amoeba Family Tree Diagram]

### 14.2.6 (Display page) Adding an amoeba child

#### Add an amoeba child

Supply the `addChild` method that, given the names of a parent amoeba and a child amoeba, locates the parent and links the child into the amoeba family. Use the `ArrayList<Amoeba> iterator` method to go through the children of a given amoeba during your search.

### 14.3 Work with amoeba family trees (8 steps)

#### 14.3.1 (Display page) Printing an amoeba family tree

**Printing an amoeba family tree**

Supply the code for the `print` method. It should print the root amoeba's name and the names of all its descendants. One name should be printed per line. The names of the children of each descendant amoeba should be printed on the lines immediately following; they should be indented four spaces more than the name of their parent. Thus the output should appear in the form

```
amoeba-name
  child-name
    grandchild-name
      great-grandchild-name
      great-great-grandchild-name
      grandchild-name
    child-name
      grandchild-name
      child-name
```

You will probably need a helper method whose argument keeps track of the indenting level.

#### 14.3.2 (Display page) Amoeba counting

**Amoeba counting**

Write and test an `AmoebaFamily` method named `size` that returns the number of amoebas in this family. (There are thirteen amoebas in the example family shown earlier.)

#### 14.3.3 (Display page) Finding the "busiest" amoeba

**Finding the "busiest" amoeba**

Consider an `AmoebaFamily` method named `busiest`, which returns the name of the amoeba with the most children. One might code it to use a helper method as follows:

```java
public String busiest ( ) {  
  if (myRoot == null) {  
    return null;  // failure case  
  } else {  
    Amoeba b = busiestAmoeba (myRoot);  
    return b.myName;  
  }
}
```

Here is the header of the helper method. Complete the method.

```java
// Return the amoeba among a and its descendents
// that has the most children.
private static Amoeba busiestAmoeba (Amoeba a) ...
```

If there are more than one amoeba with the same maximum number of children, you may return the name of any of them. If the tree is empty, return `null`. For example, in the amoeba family tree you worked with earlier, the two busiest amoebas are named "me" and "mom/dad"; both have three children. Hint: the `count` method you just completed is a good model for `busiestAmoeba`. It initialized an accumulation variable, updated that variable in a loop, and then returned the result after all the updating.

#### 14.3.4 (Display page) Computing the height of a family tree

**Computing the height of an amoeba family tree**

The height of a node in a tree is the maximum length of a path from that node to a leaf. We'll define the height of a leaf as 1, and the height of a null tree as 0. The next two exercises involve the following code for computing the height of a node in an amoeba tree.

```java
public int height (Amoeba a) {  
  if (a == null) {  
    return 0;  
  } else {  
    return 1 + max (height (a.left), height (a.right));
  }
}
```
private static int height (Amoeba x) {
    if (x.myChildren.isEmpty ( ) ) {
        return 1;
    } else {  
        int bestSoFar = 1;
        Iterator<Amoeba> iter = x.myChildren.iterator ( );
        while (iter.hasNext ( ) ) {
            Amoeba child = iter.next ( );
            bestSoFar = 1 + Math.max (height(child), bestSoFar);
        }
        return bestSoFar;
    }
}

14.3.5 (Brainstorm) Evaluating the height computation, part 1
The height computation is incorrect. How would one notice that?

14.3.6 (Brainstorm) Evaluating the height computation, part 2
Describe, as completely as you can, amoebas for which the height computation produces the correct result.

14.3.7 (Brainstorm) Evaluating the height computation, part 3
How did you determine your answer to the preceding step? If someone suggested there were more amoeba trees for which the code worked correctly than the ones you listed, how would you go about finding them?

14.3.8 (Brainstorm) Replacing the height computation
Fix the bug in the height computation. Here's the original code.

private static int height (Amoeba x) {
    if (x.myChildren.isEmpty ( ) ) {
        return 1;
    } else {  
        int bestSoFar = 1;
        Iterator<Amoeba> iter = x.myChildren.iterator ( );
        while (iter.hasNext ( ) ) {
            Amoeba child = iter.next ( );
            bestSoFar = 1 + Math.max (height(child), bestSoFar);
        }
        return bestSoFar;
    }
}

14.4 Work with binary trees

14.4.1 (Display page) Working with binary trees

Working with binary trees

As noted earlier, a common special case is a binary tree, one in which each node has at most two children. Rather than store the children in an ArrayList as done in Amoeba nodes, one normally just has two separate variables myLeft and myRight for the left and right children. Three processing sequences for nodes in a binary tree occur commonly enough to have names:

- preorder: process the root, process the left subtree (in preorder), process the right subtree (in preorder)
- postorder: process the left subtree (in postorder), process the right subtree (in postorder), process the root
- inorder: process the left subtree (in inorder), process the root, process the right subtree (in inorder)

14.4.2 (Self Test) Identifying processing sequences
Consider the binary tree

```
    A
   / \  
  B   C
 / \ /  
D  E
```
If the values are printed in preorder with one space between values, what's the output?

If the values in the above tree are printed in inorder, what's the output?

If the values in the above tree are printed in postorder, what's the output?

14.4.3 (Display page) A BinaryTree class

A BinaryTree class

The file ~cs61b/code/BinaryTree.java defines a BinaryTree class and a TreeNode class. First, read the code, and predict what output will be produced by the statement

```java
print (t, "sample tree 2");
```
in the main method. Run the program to verify your answer. Then provide a fillSampleTree

```
    A
   / \  
  B   C
 / \ /  
D  E
```
method that initializes the BinaryTree version of the tree pictured below. Run the program to verify your answer.

14.4.4 (Display page) Height and isBalanced methods

height and isBalanced methods

Add a height method to the BinaryTree class, similar to the AmoebaFamily height method you
worked with earlier. The height of an empty tree is 0; the height of a one-node tree is 1; the height of any other tree is 1 + the greater of the heights of the two children. Test your method on the various sample trees. Then provide an isCompletelyBalanced method for the BinaryTree class. An empty tree and a tree containing just one node are both completely balanced; any other tree is completely balanced if and only if the height of its left child is equal to the height of its right child, and its left and right children are also completely balanced. Test your method on the various sample trees, plus a completely balanced tree of height 3.

14.5 Homework (1 step)
14.5.1 (Display page) Programming
Programming

The last lab activity involved writing a BinaryTree method to check if this tree is completely balanced. However, some trees can't be completely balanced, for instance, trees of size 4, 5, or 6. In general, a completely balanced tree must have $2^k - 1$ nodes for some $k$. For this homework assignment, write a BinaryTree method named isMaximallyBalanced and an associated set of JUnit tests. The isMaximallyBalanced method returns true either when the tree is completely balanced, or when all the leaves are on level $k$ and level $k+1$ for some $k$, level $k$ is full, and the leaves at level $k+1$ are as far left as possible, and returns false otherwise. (The level of a leaf is its distance from the root.) Two trees appear below as examples; the left tree is maximally balanced, the right tree is not.

Submit your BinaryTree.java files as homework day14. Your solution is due prior to the start of your lab section.

15 Mid-semester survey
2008-3-06 ~ 2008-3-07 (1 activity)
15.1 Survey questions (2 steps)
15.1.1 (Display page) Some background about filling out the survey questions
Some background about filling out the survey questions

Some of the questions on the survey in the next step involve assigning numeric ratings to a list of items we provide. We would like you to input the ratings one per line, without extraneous text, in the same order as the items you're rating. Here's a sample question of this type.

Rate each of the following on a scale from 4 (highly valuable) to 1 (useless) according to how valuable they have been for improving your understanding of CS 61BL topics:

a. lab exercises
b. homework exercises
c. "brainstorm" questions
d. face-to-face collaboration with other students in lab

If you have no opinion about one of the activities above, rate it as 0.

Suppose that you thought lab exercises were really valuable, homework wasn't very useful, brainstormers were somewhat valuable, and interacting with your labmates was useless. You would then type the following into the text box:

4
2
3
1

We chose this more compressed question format (instead of a separate question for each activity) in order to make the survey easier for you to take.

16 More on trees
2008-3-11 ~ 2008-3-12 (4 activities)
16.2 Work with tree iterators (7 steps)
16.2.1 (Display page) A brief digression: stacks and queues
A brief digression: stacks and queues

Two linear data structures that represent objects in the real world are a stack and a queue. A stack data structure models a stack of papers, or plates in a restaurant, or boxes in a garage or closet. A new item is placed on the top of the stack, and only the top item on the stack can be accessed at any particular time. Stack operations include the following:

- **pushing** an item onto the stack;
- accessing the top item on the stack;
- **popping** the top item off the stack;
- checking if the stack is empty.

A computing application where you may have recognized the use of a stack is keeping track of where to return from a procedure call. Suppose procedure 1 calls procedure 2, which calls procedure 3, which calls procedure 4. Each procedure call puts the place in the program to return to on a stack maintained by the operating system. Each procedure return pops the stack to find out where to return to. The table below displays this behavior.

<table>
<thead>
<tr>
<th>action</th>
<th>effect on system stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>procedure 1 calls procedure 2</td>
<td>place to return to in procedure 1</td>
</tr>
<tr>
<td></td>
<td>place to return to in caller of procedure 1</td>
</tr>
</tbody>
</table>
A queue is a waiting line. (The term is more popular in England than it is here.) As with stacks, access to a queue's elements is restricted. Queue operations include

- adding an item to the back of the queue;
- accessing the item at the front of the queue;
- removing the front item;
- checking if the queue is empty.

Both these data structures will be used as auxiliary structures to implement the one-by-one enumeration of items in a tree.

16.2.2 (Self Test) Working with stacks and queues

Suppose that the following sequence of operations is executed using an initially empty stack. What ends up in the stack?

push A
push B
pop
push C
push D
pop
push E
pop

Suppose that the following sequence of operations is executed using an initially empty queue. What ends up in the queue?

add A
add B
remove
add C
add D
remove
add E
remove

16.2.3 (Display page) General design concerns for a tree iterator

General design concerns for a tree iterator

We now consider the problem of returning the elements of a tree one by one, using an iterator. To do this, we will implement the interface `java.util.Enumeration`, which prescribes two methods:

```java
// Are there more tree elements yet to be returned?
boolean hasMoreElements ( );

// Return the next element.
// Precondition: hasMoreElements ( );
//   throws NoSuchElementException when
//   that precondition is not met.
Object nextElement ( );
```

We will also use a nested enumeration class to hide the details of the enumeration. As with previous iterators, we need to maintain state saving information that lets us find the next tree element to return, and we now must determine what that information might include. To help work this out, we'll consider the sample tree below, with elements to be returned depth first as indicated by the numbers.

The first element to be returned is the one labeled "1". Once that's done, we need to somewhere keep track of the fact that we have to return at some point to element "5". Similarly, once we return element "2", we have to remember that element "4" is yet to return, as in the diagram below.

That means that our state-saving information must include not just a single pointer of what to return next, but a whole collection of "bookmarks" to nodes we've passed along the way. More generally, we will maintain a collection that we'll call fringe or frontier of all the nodes in the tree that are candidates for returning next. The `nextElement` method will choose one of the elements of the fringe as the one to return, then add its children to the fringe as candidates for the next element to return.
hasMoreElements is true when the fringe isn’t empty. The iteration sequence will then depend on
the order we take nodes out of the fringe. Depth-first iteration, for example, results from storing the
fringe elements in a stack, a last-in first-out structure. The java.util; class library conveniently
contains a Stack class with push and pop methods. We illustrate this process on a binary tree
in which tree nodes have 0, 1, or 2 children named myLeft and myRight. Here is the code.

```java
public class DepthFirstIterator implements Enumeration {
    private Stack fringe = new Stack ( );
    public DepthFirstIterator ( ) {
        if (myRoot != null) {
            fringe.push (myRoot);
        }
    }
    public boolean hasMoreElements ( ) {
        return !fringe.empty ( );
    }
    public Object nextElement ( ) {
        if (!hasMoreElements ( )) {
            throw new NoSuchElementException ("tree ran out of elements");
        }
        TreeNode node = (TreeNode) fringe.pop ( );
        if (node.myRight != null) {
            fringe.push (node.myRight);
        }
        if (node.myLeft != null) {
            fringe.push (node.myLeft);
        }
        return node;
    }
}
```

16.2.4 (Brainstorm) Stack contents during depth-first iteration

What’s on the stack when element 4 in the tree below has just been returned by nextElement?
What about element 6? Explain how you determined your answers.

![Tree diagram](image)

16.2.5 (Brainstorm) Effect of pushing left child before right

Suppose the nextElement code pushes the left child before the right:

```java
if (node.myLeft != null) {
    fringe.push (node.myLeft);
}
if (node.myRight != null) {
    fringe.push (node.myRight);
}
```

In what order are the elements of the tree below returned? Briefly explain your answer.

![Tree diagram](image)

16.2.6 (Display page) A depth-first Amoeba iterator

A depth-first Amoeba iterator

Complete the definition of the AmoebaEnumeration class. It should successively return names of
amoebas from the family in preorder, with children amoebas returned oldest first. Thus, for the family
set up in the AmoebaFamily main method, the name “Amos McCoy” should be returned by the first
call to nextElement; the second call to nextElement should return the name “auntie”; and so on. Do
not change any of the framework code. Organizing your code as described in the previous step will
result in a process that takes time proportional to the number of amoebas in the tree to return them
all. Moreover, the constructor and hasMoreElements both run in constant time, while nextElement
runs in time proportional to the number of children of the element being returned. Uncomment the
code at the end of the AmoebaFamily main method to test your solution. It should print names in the
same order as the call to family.print, though without indenting.

16.2.7 (Display page) A breadth-first Amoeba iterator

A breadth-first Amoeba iterator

Now rewrite the AmoebaEnumeration class to use a queue (first in, first out) instead of a stack. This
will result in amoeba names being returned breadth first. That is, the name of the root of the family
will be returned first, then the names of its children (oldest first), then the names of all their children,
and so on. For the family constructed in the AmoebaFamily main method, your modification will result
in the following enumeration sequence:

```
Amos McCoy
auntie
mom/dad
me
Fred
Wilma
Mike
 Homer
Narge
Sart
Lisa
Bill
Hilary
```

There isn’t a built-in queue class, so you’ll have to simulate one using a list (either your own List or
16.3 Build and check a tree (6 steps)
16.3.1 (Display page) Printing a tree

Printing a tree

We now return to binary trees, in which each node has either 0, 1, or 2 children. Last lab, we worked with an implementation of binary trees using a BinaryTree class with a nested TreeNode class, as shown below.

```java
public class BinaryTree {
    private TreeNode myRoot;
    private static class TreeNode {
        public Object myItem;
        public TreeNode myLeft;
        public TreeNode myRight;
        public TreeNode (Object obj) {
            myItem = obj;
            myLeft = myRight = null;
        }
        public TreeNode (Object obj, TreeNode left, TreeNode right) {
            myItem = obj;
            myLeft = left;
            myRight = right;
        }
    }
    ...}
```

The framework is available online in `~cs61b/code/BinaryTree.java`. Fill in the blanks in the following code to print a tree so as to see its structure.

```java
public void print ( ) {
    if (myRoot != null) {
        printHelper (myRoot, 0);
    }
}
```

```java
private static void println (Object obj, int indent) {
    for (int k=0; k<indent; k++) {
        System.out.print (indent1);
    }
    System.out.println (obj);
}
```

The `print` method should print the tree in such a way that if you turned it 90 degrees counterclockwise, you see the tree. Here's an example:

<table>
<thead>
<tr>
<th>Tree</th>
<th>Printed version</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
</tr>
</tbody>
</table>

16.3.2 (Display page) Checking a BinaryTree object for validity

Checking a BinaryTree object for validity

A legal binary tree has the property that, when the tree is traversed, no node appears more than once in the traversal. A careful programmer might then include a method named, say, isOK to check that property:

```java
public boolean isOK ( ) {
    alreadySeen = new ArrayList ( );
    try {
        check (myRoot);
        return true;
    } catch (IllegalStateException e) {
        return false;
    }
}
```

// Contains nodes already seen in the traversal.
private ArrayList alreadySeen;

(IllegalStateException is provided in Java.) Write and test the `check` method, using variations of the sample trees you constructed for earlier exercises as test trees. Here's the header:

```java
private void check (TreeNode t) throws IllegalStateException
```

You may use any traversal method you like. You should pattern your code on earlier tree methods, for example, `count` and `busiest`. Incidentally, this exercise illustrates a handy use of exceptions to return from deep in a recursion.

16.3.3 (Brainstorm) Analyzing isOK's running time

Complete the following sentence.

The `isOK` method, in the worst case, runs in time proportional to ___ , where N is
Briefly explain your answer. After reviewing your labmates’ answers, correct your own if necessary.

16.3.4 (Display page) Building a Fibonacci tree

Building a Fibonacci tree

This exercise deals with “Fibonacci trees”, trees that represent the recursive call structure of the Fibonacci computation. (The Fibonacci sequence is defined as follows: \( F_0 = 0, F_1 = 1 \), and each subsequent number in the sequence is the sum of the previous two.) The root of a Fibonacci tree should contain the value of the \( n \)th Fibonacci number the left subtree should be the tree representing the computation of the \( n-1 \)st Fibonacci number, and the right subtree should be the tree representing the computation of the \( n-2 \)nd Fibonacci number. The first few Fibonacci trees appear below.

```
  fibtree(0)  fibtree(1)  fibtree(2)  fibtree(3)  fibtree(4)  fibtree(5)
```

Supply the `fibTreeHelper` method to go with the `BinaryTree` method below.

```java
public static BinaryTree fibTree (int n) {
    BinaryTree result = new BinaryTree ();
    result.myRoot = result.fibTreeHelper (n);
    return result;
}

private TreeNode fibTreeHelper (int n) ...
```

Note: primitive types like `int` are not objects. Java provides wrapper classes that allow primitive values to be stored as objects. In some situations, Java will automatically convert one to the other. However, for this exercise, you should use the `Integer` wrapper class. To create an `Integer` object, evaluate

```
new Integer (n)
```

where \( n \) is the integer you want to store. To access the stored integer value, use the `intValue` method.

16.3.5 (Display page) Building an expression tree

Building an expression tree

Compilers and interpreters convert string representations of structured data into tree data structures. For instance, they would contain a method that, given a `String` representation of an expression, returns a tree representing that expression:

```java
public static BinaryTree exprTree (String s) {
    BinaryTree result = new BinaryTree ();
    result.myRoot = result.exprTreeHelper (s);
    return result;
}
```

Complete and test the following helper method for `exprTree`.

```java
// Return the tree corresponding to the given arithmetic expression.
// The expression is legal, fully parenthesized, contains no blanks,
// and involves only the operations + and *.
private TreeNode exprTreeHelper (String expr) {
    if (expr.charAt (0) != '(') {
        ____; // you fill this in
    } else {
        // expr is a parenthesized expression.
        // Strip off the beginning and ending parentheses,
        // find the main operator (an occurrence of + or * not nested
        // in parentheses, and construct the two subtrees.
        int nesting = 0;
        int opPos = 0;
        for (int k=1; k<expr.length()-1; k++) {
            ____ ; // you supply the missing code
            String opnd1 = expr.substring (1, opPos);
            String opnd2 = expr.substring (opPos+1, expr.length()-1);
            String op = expr.substring (opPos, opPos+1);
            System.out.println ("expression = " + expr);
            System.out.println ("operand 1  = " + opnd1);
            System.out.println ("operator   = " + op);
            System.out.println ("operand 2  = " + opnd2);
        }
    }
    ____; // you fill this in
}
```

Given the expression `((a+(5*(a+b)))+(6*5))`, your method should produce a tree that, when printed using the `print` method you just designed, would look like

```
  5
  * 6
   * b
     * a
     * 5
   * a
```

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
16.3.6 (Display page) Optimizing an expression tree

Optimizing an expression tree

Given a tree returned by the exprTree method, write and test a method named optimize that replaces all occurrences of an expression involving only integers with the computed value. Here's the header.

```java
public void optimize ( )
```

It will call a helper method as did BinaryTree methods in earlier exercises.

```java
your optimize method should produce the tree corresponding to the expression

```java
((a+50)+30)
```

Don't create any new TreeNode; merely relink those already in the tree.

16.4 Homework (4 steps)

16.4.1 (Display page) Project 2 progress

Project 2 progress

The activities for lab on Thursday and Friday should leave you some time to make more progress on your project. Please come prepared. For homework, provide posts and responses to the next two discussions, and prepare a plan for project 2 to show to your t.a. on Thursday or Friday.

16.4.2 (Discussion Forum) Project 2 questions

In a post (by tomorrow morning), supply a question that you or a labmate have about project 2. Then, prior to your lab on Thursday or Friday, provide a response by answering one of your labmates' questions or by significantly elaborating on an answer already provided.

16.4.3 (Discussion Forum) Project tough spots?

In a post (by tomorrow morning), make a guess about what aspect of project 2 will be more difficult, and provide a brief explanation. Then, prior to lab on Thursday or Friday in response to one of your labmates, suggest a way to deal with or simplify the difficult aspect of the project that he or she described.

16.4.4 (Display page) Project 2 plan

Project 2 plan

Prepare a plan for proceeding on project 2 to show your lab t.a. on Thursday or Friday. This checkpoint will be worth up to 2 homework points.

17 Binary search trees

2008-3-13 ~ 2008-3-14

(5 activities)

17.2 Show your t.a. that you've thought about project 2. (1 step)

17.2.1 (Display page) Project 2 thoughts

Project 2 thoughts

You were to prepare a plan for proceeding on project 2 to show your lab t.a. today. This checkpoint will be worth up to 2 homework points:

- 2 points if you are ready to write some code;
- 1 point if you have a plan for the structure of the program, but it's too vague to see how you might start implementing it;
- 0 points if you apparently haven't yet thought about the project.

17.3 Review search with arrays and linked lists. (5 steps)

17.3.1 (Display page) Search in a collection

Search in a collection

An important operation provided by collection classes is finding whether a given item is an element of the collection. You've encountered this operation in several contexts already, for example, the AmoebaFamily activity involving locating an amoeba with a given name.

17.3.2 (Self Test) Search estimates

Suppose that \(n\) integers are stored in an IntSequence object. How many comparisons are necessary in the worst case to determine if a given integer \(x\) occurs in the sequence?

Suppose that \(n\) comparable objects are stored in a List object. How many comparisons are necessary in the worst case to determine if a given object \(x\) occurs in the list?

Suppose that \(n\) integers are stored in increasing order in an IntSequence object. How many comparisons are necessary in the worst case to determine if a given integer \(x\) occurs in the sequence?

Suppose that \(n\) comparable objects are stored in increasing order in a List object. How many comparisons are necessary in the worst case to determine if a given object \(x\) occurs in the list?

17.3.3 (Display page) Binary search

Binary search

We encountered a variation of the binary search algorithm in a guessing game in a lab exercise earlier this semester. Used with an array, it assumes that the elements of the array are in order (say, increasing order). A version that returns the position in the array of the value being looked for works as follows:

1. Set low to 0 and high to the length of the array, minus 1. The value we're looking for—we'll call it key—will be somewhere between position low and position high if it's in the array.
2. While low \(\leq\) high, do the following:
   a. Compute mid, the middle of the range \([low, high]\), and see if that's key. If so, return mid.
   b. Otherwise, we can cut the range of possible positions for key in half, by setting high to
mid-1 or by setting low to mid+1, depending on the result of the comparison.

3. If the loop terminates with low > high, we know that key is not in the array, so we return
some indication of failure.

The diagrams below portray a search for the key 25. Elements removed from consideration at each
iteration are greyed out.

<table>
<thead>
<tr>
<th>low</th>
<th>high</th>
<th>mid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Since (roughly) half the elements are removed from consideration at each step, the worst-case
running time is proportional of the log base 2 of N, where N is the number of elements in the array.

17.3.4 (Brainstorm) Binary search with linked lists?
Consider applying the binary search algorithm to a sorted doubly linked list. The variables low, high,
and mid would then be references to nodes in the list. Which of the steps in the binary search
algorithm would execute significantly more slowly for a collection stored in a sorted doubly linked list
than for a collection stored in a sorted array?

17.3.5 (Brainstorm) Post-search operations
As you just observed, binary search with a sorted array is significantly faster than search with a
sorted doubly linked list. Now consider what might happen after a successful search. Binary search
would return the position of the key in the array; search in a linked list—again, we'll assume it's
doubly linked—would return a reference to the corresponding node in the list. Describe an operation
that, when applied to the search result, would be significantly faster with the doubly linked list than
with the array.

17.4 Work with binary search trees.

Binary search trees
One might naturally look for a linked structure that combines fast search with fast link manipulation of
the found node. The tree of choices in a binary search algorithm suggests a way to organize keys in
an explicitly linked tree, as indicated in the diagram below.

The tree that results is called a binary
search tree (BST).

• Let the root value (one of the keys to be stored) be k.
• Put all the keys that are smaller than k into a binary search tree, and let that tree be k's left
subtree.
• Put all the keys that are larger than k into a binary search tree, and let that tree be k's right
subtree.

(This organization assumes that there are no duplicate keys among those to be stored.) Given below
are an example of a binary search tree and two examples of trees that are not binary search trees.

17.4.2 (Brainstorm) Identifying trees that aren't binary search trees
Given below are two trees that aren't binary search trees. How does each tree violate the
requirements for a binary search tree?

17.4.3 (Display page) The "contains" method
The contains method
Determining if a given value key is in the tree takes advantage of the recursive structure. In
pseudocode, here is an outline of the helper method

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
private static boolean contains (TreeNode t, Object key)

1. Nothing is in an empty tree, so if \( t \) is null, return false.
2. If \( \text{key} \) is equal to \( t\text{.myItem} \), return true.
3. If \( \text{key} < t\text{.myItem} \), \( \text{key} \) must be in the left subtree if it's in the structure at all, so return the result of searching for it in the left subtree.
4. Otherwise return the result of search for \( \text{key} \) in the right subtree.

This algorithm can go all the way down to a leaf to determine its answer. Thus in the worst case it makes \( O(d) \) comparisons, where \( d \) is the depth of the tree. Objects, Abstraction, Data Structures, and Design has code for the contains method—the authors name it find—in listing 8.3.

### 17.4.4 (Display page) Use of Comparable objects

#### Use of comparable objects

As just noted, finding a value in the tree will require "less than", "greater than", and "equals" comparisons. Since the operators < and > don't work with objects, we have to use method calls for comparisons. The Java convention for this situation is to have the values stored in the tree be objects that implement the Comparable interface. This interface prescribes just one method, int compareTo (Object). For Comparable objects \( \text{obj1} \) and \( \text{obj2} \), \( \text{obj1}\text{.compareTo} \) is

- negative if \( \text{obj1} \) is less than \( \text{obj2} \),
- positive if \( \text{obj1} \) is greater than \( \text{obj2} \), and
- zero if the two objects have equal values.

### 17.4.5 (Display page) Balance and imbalance

#### Balance and imbalance

Unfortunately, use of a binary search tree does not guarantee efficient search. For example, the tree

```
    7
   /|
  6 /|
 / | | /
4 / | | |
/  | |   /
3  / |   |
/   |   |
2   |   |
   /   |
  1
```

is a binary search tree in which search proceeds the same as in a linked list. We thus are forced to consider the balance of a binary search tree. Informally, a balanced tree has subtrees that are roughly equal in size and depth. In a few weeks, we will encounter specific algorithms for maintaining balance in a binary search tree. Until then, we will work under the possibly unwarranted assumption that we don't need to worry much about balance. One can prove, incidentally, that search in a BST of \( N \) keys will require only about \( 2 \log N \) comparisons (i.e. the "natural log" of \( N \)) if the keys are inserted in random order. Well-balanced trees are common, and degenerate trees are rare.

### 17.4.6 (Brainstorm) Count of maximally imbalanced trees

There are 14 different binary search trees with 4 nodes. How many of the 14 are as deep as possible (i.e. they cause the search algorithm to make 4 comparisons in the worst case)? Briefly explain how you got your answer. (Luckily these bad trees get rarer when \( N \) gets larger.)

### 17.4.7 (Display page) Insertion into a BST

#### Insertion into a binary search tree

As you may have noticed in the preceding step, there are a large number of binary search trees that contain a given \( N \) keys. Correspondingly, there are typically a bunch of places in a BST that a key to be inserted might go, anywhere from the root down to a leaf. Given below are two trees; the tree on the right shows one possible way to insert the key 41 into the tree on the left.

To minimize restructuring of the tree and the creation of internal nodes with only one child, we choose to insert a new key only as a new leaf. The following pair of methods does this.

```java
public void add (Comparable key) {
    myRoot = add (myRoot, key);
}

private static TreeNode add (TreeNode t, Comparable key) {
    if (t == null) {
        return new TreeNode (key); // return the new leaf
    } else if (key.compareTo (t.myItem) < 0) {
        t.myLeft = add (t.myLeft, key);
        return t;
    } else {
        t.myRight = add (t.myRight, key);
        return t;
    }
}
```

A common pattern for methods that modify trees is the following:

Make a recursive call to modify a subtree—

```
http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
```
this call returns the root of the modified subtree.
Store the reference to the modified subtree into
the appropriate TreeNode field (myLeft or myRight),
and return the root of the current tree.

(You may have encountered this in methods involving ListNode.) In the code above, this is done in
the statements

\[
t.\text{myLeft} = \text{add}(t.\text{myLeft}, \text{key});
\text{return } t;
\]

and

\[
t.\text{myRight} = \text{add}(t.\text{myRight}, \text{key});
\text{return } t;
\]

The corresponding method in Objects, Abstraction, Data Structures, and Design appears in listing
8.4.

17.4.8 (Display page) A BinarySearchTree class

A BinarySearchTree class

Since binary search trees share many of the characteristics of regular binary trees, we can define
the BinarySearchTree class using inheritance. Do this as follows.

1. First, redefine relevant private methods and instance variables in BinaryTree.java as
protected.
2. Next, create a file BinarySearchTree.java with the class definition

\[
\text{public class BinarySearchTree extends BinaryTree} \ldots
\]

(The BinarySearchTree class shouldn't have any instance variables. Why not?)
3. Finally, supply two methods for the BinarySearchTree class.

\[
\text{public boolean contains (Comparable)}
\]
takes a Comparable object as argument and checks whether the tree contains it. (Recall that Comparable objects provide an int compareTo method that returns a
negative integer if this object is less than the argument, a positive integer if this is
greater than the argument, and 0 if the two objects have equal values.)

\[
\text{public void add (Comparable)}
\]
takes a Comparable object as argument and adds it to the tree only if it isn't already
there.

The trees you create with the add method will thus not contain any duplicate elements.
4. Test your code. In particular, using only calls to add inside JUnit or a main method, create the
tree shown below.

```
    C
   /|
  B  E
 / | /
A  F D
```

17.4.9 (Brainstorm) Comparison counting

You may work on this with a partner if you want. How many comparisons between keys were
necessary to produce the sample tree in the previous step? Explain how you got your answer. Then,
after reviewing your labmates' answers, correct your own answer if appropriate.

17.5 Homework (4 steps)

17.5.1 (Display page) Project + programming exercises + discussion

Project + reading + discussion

For next lab, work on project 2. In lab on Tuesday and Wednesday, t.a.s will ask to see significant
progress. A quiz and project work will be the only activities on Tuesday and Wednesday. Also
provide solutions for the two programming exercises in the next step. Finally, contribute two posts
due tomorrow morning) and two responses (due by next lab) to the following discussion.

17.5.2 (Display page) Programming exercises

Programming

Complete the following two exercises, and submit your solutions in the file binaryTree.java by the
start of lab on Tuesday, March 18 or Wednesday, March 19. Submit your file as day16.

1. A textbook used in a previous offering of CS 61BL presented code for an inorder iterator.
This code is in the file ~/cs61b/code/InorderIterator.java (it should go inside the
BinaryTree class). The iterator is not optimally coded; in particular, the invariant relation
between current node and nodeStack maintained by the next method isn't stated anywhere,
and anyway this invariant is more complicated than it should be. Fix the code so that, after
each call to the constructor and the next method, the next item to return will be at the top of
nodeStack. The corresponding change to hasNext will be

\[
\text{public boolean hasNext( )} \quad \{ \\
\text{return } !\text{nodeStack.isEmpty( );}
\}
\]

2. Write and test a constructor for the BinarySearchTree class that, given two ArrayLists of Objects,
constructs the corresponding binary tree. The first ArrayList contains the objects in the
order they would be enumerated in a preorder traversal; the second ArrayList contains the
objects in the order they would be enumerated in an inorder traversal. For example, given
ArrayLists ['A', 'B', 'C', 'D', 'E', 'F'] (preorder) and ['B', 'A', 'E', 'D', 'F', 'C'] (inorder),
you should initialize the tree as follows:

17.5.3 (Discussion Forum) Project 1 post-mortem
Supply one post that describes the biggest mistake you made in designing, coding, or testing project 1. Then supply another that describes something you did that was a big help in designing, coding, or testing project 1. Do this by tomorrow morning. Then, before your next lab, comment on a mistake of one of your labmates by describing how you avoided the error he or she described. Also comment on a positive thing that one of your classmates did (and you didn't do) by noting how well that positive action would have worked for you. Our not-so-hidden agenda: try not to repeat mistakes.

17.5.4 (Display page) Project 2 progress
Project 2 progress
Your lab t.a. will expect to see that you've made some progress on project 2 in the next lab, namely code to handle a change in the directory structure, along with code to handle the ls, lstree, or cat method to show that it works.

18 Activities for March 18 and 19
2008-3-18 ~ 2008-3-19 (2 activities)
18.2 Work on project 2 (1 step)
18.2.1 (Display page) Project 2 progress
Project 2 progress
During lab, your t.a. will be evaluating your progress on project 2. He will expect to see code to handle a change in the directory structure, along with code to handle the ls, lstree, or cat method to show that it works. The rest of the lab section will be devoted to working on the project. Reminder: Your work on this project should be essentially your own. Don't show any of your code to any of your classmates, and acknowledge any significant help you received (either from staff or from fellow students).

19 More on binary search trees; maps
2008-3-20 ~ 2008-3-21 (5 activities)
19.2 Delete a key from a binary search tree (8 steps)
19.2.1 (Display page) Deletion from a BST
Deletion from a binary search tree
When inserting a key, we were allowed to choose where in the tree it should go. The deletion operation doesn't appear to allow us that flexibility: deletion of some keys will be easy (leaves or keys with only one child), but our deletion method has to be able to delete any key from the tree. Here are a bunch of binary search trees that might result from deleting the root of the tree

Think about which ones are reasonable. (The next step asks for your opinion about this.)

19.2.2 (Brainstorm) Good choices for deletion result
Which of the twelve trees in the previous step make the most sense for the result of deleting the root of the original tree? Briefly explain your answer.

19.2.3 (Display page) A good way to delete a key
A good way to delete a key
The following algorithm for deletion has the advantage of minimizing restructuring and unbalancing of the tree. The method returns the tree that results from the removal.

1. Find the node to be removed. We'll call it remNode. If it's a leaf, remove it immediately by returning null. If it has only one child, remove it by returning the other child node.

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
2. Otherwise, remove the inorder successor of `remNode`, copy its `myItem` into `remNode`, and return `remNode`.

An example is diagrammed below. The node to remove is the node containing 4. It has two children, so it's not an easy node to remove. We locate the node with 4's inorder successor, namely 5. It has only one child, so it's easily to delete. We copy the 5 into the node that formerly contained 4, and delete the node that originally contained 5.

<table>
<thead>
<tr>
<th>Before deletion</th>
<th>After deletion</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
</tbody>
</table>

19.2.4 (Brainstorm) A feature of the inorder successor
Suppose `node` is a node in a BST with both a left child and a right child. Why must `node`'s inorder successor node have a null left child? After answering, review your labmates' responses and correct your own response if necessary.

19.2.5 (Brainstorm) A general property of the inorder successor
Where in a BST is the inorder successor of a node whose right child is null? After answering, review your labmates' responses and correct your answer if necessary.

19.2.6 (Brainstorm) Complicated aspects of BST algorithms
What aspects of insertion into and deletion from a binary search tree are hardest to understand?

19.2.7 (Display page) Animations of BST algorithms
Experiment with this [applet](http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary), which displays the result of insertions and deletions in a binary search tree in animated form. Try inserting a key that ends up as a right child and another key that ends up as a left child. Add some more nodes to the tree, then delete some of them: a node with no children, a node with one child, and a node with two children. Keep track of what you learn from your experiments. You may work with a partner on this if you want.

19.2.8 (Brainstorm) Usefulness of animations?
How, if at all, did the animations clarify the complex aspects of BST insertion and deletion that you mentioned previously? (If the animations didn't help, please say so with a brief explanation.)

19.3 Get acquainted with maps.
(7 steps)

19.3.1 (Display page) Tables
In CS 61A and possibly in CS 3, you occasionally worked with tables that stored collections of associations between symbols and other objects. For example, a function to translates a Roman numeral to decimal might use a table that stores the following associations:

<table>
<thead>
<tr>
<th>Roman digit</th>
<th>corresponding decimal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>V</td>
<td>5</td>
</tr>
<tr>
<td>X</td>
<td>10</td>
</tr>
<tr>
<td>L</td>
<td>50</td>
</tr>
<tr>
<td>C</td>
<td>100</td>
</tr>
<tr>
<td>D</td>
<td>500</td>
</tr>
<tr>
<td>M</td>
<td>1000</td>
</tr>
</tbody>
</table>

In CS 61A, you first encountered tables in the context of data-directed programming. Other applications of tables in that course included memoization of recursive processes (we'll revisit that application shortly), keeping track of attributes of characters in the Adventure game, and working with environments in the metacircular evaluator. More recently, you encountered tables as sets of associations in the "Medical Diagnosis" case study. A table has two important features.

- It's a collection of pairs of information. Each pair consists of a key and an associated value.
- Access to table elements is generally by key: we ask for the value that's associated with a given key. There can't be any more than one value associated with a given key. For example, if "mike" was associated with the value 27, it would not also be associated with the value 45.

19.3.2 (Display page) The Map interface
Java's name for an association table is a `Map`. `Map` is an interface in `java.util`, not an actual class; an actual class that stores associations would implement `Map` and provide definitions for the methods it prescribes. Among these methods, the most interesting are

- `Object put (Object key, Object value)` which adds the association between key and value to the table, and
- `Object get (Object key)` which retrieves the value associated with key

The `Object` returned by `put` is the value previously associated with the given key, or `null` if the key had no previously associated value. (Recall that tables contain at most one associated value for each key.) The example below, in which `table` is assumed to be an instance of a class

```java
Map<String, Integer> table = new HashMap<>();
table.put("mike", 27);
table.put("mike", 45); // Error: Only one value associated with a key
```

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
implementing Map, demonstrates how put works. (The table stores associations between people and blood types.)

<table>
<thead>
<tr>
<th>statement</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>value = table.put (&quot;mike&quot;, &quot;a-&quot;);</td>
<td>value contains null;</td>
</tr>
<tr>
<td></td>
<td>the set of associations represented by table is {(&quot;mike&quot;, &quot;a-&quot;), (&quot;sara&quot;, &quot;a+&quot;)}.</td>
</tr>
<tr>
<td>value = table.put (&quot;sara&quot;, &quot;a+&quot;);</td>
<td>value contains null;</td>
</tr>
<tr>
<td></td>
<td>table represents a set of two associations, {(&quot;mike&quot;, &quot;a-&quot;), (&quot;sara&quot;, &quot;a+&quot;), (&quot;mike&quot;, &quot;a-&quot;), (&quot;sara&quot;, &quot;a+&quot;)}.</td>
</tr>
<tr>
<td>value = table.put (&quot;mike&quot;, &quot;ab-&quot;);</td>
<td>value contains &quot;a-&quot; since that was the value originally associated with &quot;mike&quot;;</td>
</tr>
<tr>
<td></td>
<td>the set of associations represented by table is {(&quot;mike&quot;, &quot;ab-&quot;), (&quot;sara&quot;, &quot;a+&quot;)}.</td>
</tr>
<tr>
<td>value = table.put (&quot;david&quot;, &quot;o-&quot;);</td>
<td>value contains null;</td>
</tr>
<tr>
<td></td>
<td>table represents a set of three associations, {(&quot;mike&quot;, &quot;ab-&quot;), (&quot;sara&quot;, &quot;a+&quot;), (&quot;david&quot;, &quot;o-&quot;), (&quot;mike&quot;, &quot;a-&quot;), (&quot;sara&quot;, &quot;a+&quot;)}.</td>
</tr>
</tbody>
</table>

19.3.3 (Brainstorm) A class that implements Map

Describe the private instance variable(s) of a class that implements the Map interface, and explain how the get and put operations would work with them. (There are several reasonable answers.)

19.3.4 (Brainstorm) A different kind of association

Suppose you already have a class named Table that implements Map, and you want to associate each key with a collection of values. For instance, you might want to associate each blood type with the set of people who have that blood type. You decide to do this with a class named KeyCollectionTable that inherits from Table:

```java
public class KeyCollectionTable extends Table ...
```

KeyCollectionTable.put acts somewhat differently from Table.put; given a key and a value, KeyCollectionTable.put adds the value to the collection associated with the key. Here's an example:

```java
KeyCollectionTable table = new KeyCollectionTable ( );

// table contains the association {"o+", "mike"}

table.put ("ab-", "elaine");
// table contains two associations:
// {"o+", "mike"} and {"ab-", "elaine"}

table.put ("o+", "angel");
// table contains two associations:
// {"o+", "mike"}, {"o+", "angel"} and {"ab-", "elaine"}
```

Provide a definition for the KeyCollectionTable.put method. Once you have reviewed your labmates' responses, correct your own response if appropriate.

19.3.5 (Display page) java.util classes that implement Map

Several classes in java.util implement the Map interface. Two involve hash tables, which we'll study next week. Another, TreeMap, is based on a balanced binary search tree.

19.3.6 (Self Test) TreeMap organization

Which of the following trees represents a TreeMap storing the four associations ("Mike", "Clancy"), ("Paul", "Hale"), ("Leonard", "Wei"), and ("Ryan", "Waliany")? (There may be more than one correct answer.)

19.3.7 (Brainstorm) Callbacks

Add an appropriate compareTo method to the Measurement class (it should already have an appropriately defined equals method). Then determine via experiment whether a call to TreeMap.put produces a callback to Measurement.equals, Measurement.compareTo, or to both. Report the results of your experiment.

19.4 (Display page) Trading memory for speed

Trading memory for speed

Consider the problem of finding the s'th largest key in a binary search tree. An obvious way of doing this is to use the inorder enumeration from a recent homework; k calls to nextElement get us the desired key. If k is near the number of keys in the tree, however, that algorithm's running time is proportional to \(N\). We'd like to do better. Storing in each node the size of the tree rooted at that node allows the design of an algorithm that runs in time proportional to \(d\), where \(d\) is the depth of the tree. Modify the TreeNode class and the BinarySearchTree add method to maintain this information in each TreeNode. Then write and test a method

```java
Comparable kthLargest (int k);
```

that returns the s'th largest item in the tree, where \(k > 0\) and \(k < N\) is the number of keys in the tree.

19.4.2 (Brainstorm) Analyzing memory/speed tradeoffs (1)

Recall the algorithm to delete a node:

1. Find the node to be deleted.
2. If it's easy to delete, delete it; otherwise, find its inorder successor, copy the successor's value into the node to be deleted, and delete the successor.

If each node stored a pointer to its inorder successor, would that significantly improve the worst-case performance of the deletion algorithm (expressed as a "runs in time proportional to ..." estimate)? Explain why or why not. Once you have a chance to check your labmates' answers, correct your own if necessary.

19.4.3 (Brainstorm) Analyzing memory/speed tradeoffs (2)
Here's a outline of a solution to the inorder enumeration problem from a recent homework.

Constructor:

```java
t = myRoot;
myFringe = new Stack(); myFringe.push(t);
while (t.myLeft != null) {
    myFringe.push(t.myLeft);
    t = t.myLeft;
}
```

```java
nextElement:

t = myFringe.pop();
t = toReturn.myRight;
if (t != null) {
    myFringe.push(t);
    while (t.myLeft != null) {
        myFringe.push(t.myLeft);
        t = t.myLeft;
    }
}
return toReturn.myItem;
```

If we stored in each TreeNode a pointer to the node's inorder successor, how if at all would the performance of this algorithm (expressed as a "runs in time proportional to ..." estimate) improve? Be explicit, and briefly explain your answer.

19.5 Homework (1 step)

19.5.1 (Display page) Reading

Readings

Reminder: project 2 is due the day you return from spring break (Monday, March 31). As usual, you have a total of 72 "late hours" to apply toward late project submissions. Lab activities on April 1 and 2 will cover the topic of hashing, described in chapter 9 in Objects, Abstraction, Data Structures, and Design.

20 Hashing 2008-4-01 ~ 2008-4-02 (4 activities)

20.2 Work with another kind of Map.(16 steps)

20.2.1 (Display page) Review of TreeMap

Review of TreeMap

Recall the TreeMap class from last lab. Like any class that implements the Map interface, it stores entries, each consisting of a key and a value. It is implemented as a balanced binary search tree, organized by keys. Its put method, given objects key and value, calls key.compareTo to find out where in the tree the new entry should go.

20.2.2 (Self Test) TreeMap insertion

Suppose that the TreeMap object named table contains three entries as shown below. In each entry, the key is the first name and the value is the last name.

```
("Mike", "Clancy")
("Ryan", "Waliany")
("Paul", "Hale")
```

What tree results from the call table.put ("Leonard", "Wei")?

20.2.3 (Display page) The quest for a faster map

The quest for a faster map

The number of comparisons to do an insertion or search in a TreeMap that contains N entries is proportional to log N, since the underlying binary search tree is balanced. Can we do better? Here are two reasons for thinking we can.

- The binary search tree contains a lot of structure. With a simple inorder traversal, we can access the elements in ascending order by key. If the key we're looking for isn't in the tree, we can find the closest key that is, again with around log N comparisons. Maybe this structure is more than we need to do simple "is this key in the table?" queries.
- If the keys are integers, we can store the values in an array and access them immediately using array indexing. For example, if table were an array of 100 elements that stores the entries (5, "Clancy"), (17, "Wei"), (42, "Waliany"), and (83, "Hale"), the value associated with 5 is just table[5], and we can access that value in constant time.

Hashing is a technique for extending the constant-time access to non-integer keys.

20.2.4 (Display page) Hashing: a solution

Hashing: a solution

The idea underlying hashing is to add a step to the search process, namely the transformation of the key into an int that can be used to index an array. If this transformation is fast and different keys get transformed into different index values, then we can approximate the direct access that an array provides. The transforming function is called a hash function, and the int it returns is the hash value. (The reason for the term "hash" will appear shortly.) Here's an example, using the entries ("Mike", "Clancy"), ("Leonard", "Wei"), ("Ryan", "Waliany"), and ("Paul", "Hale"). Coincidentally, all the keys start with different first letters ("M", "L", "R", and "P"). A reasonable hash function for this set of keys is the ASCII code of the first letter of the key: ASCII codes for the upper-case letters range from 65 (A) to 90 (Z), so applying this hash function to the four keys gives us the following results:

<table>
<thead>
<tr>
<th>key</th>
<th>hash value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Mike&quot;</td>
<td>77</td>
</tr>
<tr>
<td>&quot;Leonard&quot;</td>
<td>76</td>
</tr>
<tr>
<td>&quot;Ryan&quot;</td>
<td>82</td>
</tr>
<tr>
<td>&quot;Paul&quot;</td>
<td>80</td>
</tr>
</tbody>
</table>

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
Collisions: a problem

In the previous example, all four keys had different hash values. That will not be the case in general, and any hashing scheme has to have a way to handle collisions (keys with the same hash value). One solution is to store the colliding keys elsewhere in the array. (This strategy is called "linear probing"; we'll encounter it shortly.) A second, more common (and easier) solution is to store all the keys with a given hash value together in a collection of their own; this collection is often called a "bucket". Here are the example entries from the previous step, hashed into a 10-element table of linked lists using the function \( \text{hash(key)} = \text{key.length} \).

```
HashMap
```

We would like to have the collections for the various hash values be as close in size as possible, since this produces the best average running time. (To get some insight justifying this claim, look at the worst case, where one hash collection has all the keys and the rest have none. This basically boils down to linear search, which we want to avoid.) We thus want the hash function to scatter the keys throughout the table, to minimize clumps produced by keys that are similar in some way to one another. This is why it's called a "hash" function. One definition of "hash" is "to muddle or confuse", and a good hash function will "muddle" whatever ordering exists in a set of keys. Note also that the performance of hashing depends very much on average case behavior. No matter what the hash function, we can always produce a set of keys that make it perform as badly as possible.

20.2.6 (Display page) java.util.HashMap

The class `java.util.HashMap` is implemented as a hash table, that is, an array of collections as just diagrammed. (The elements of this array are called "buckets".) The `put` method adds an entry to the `HashMap`:

```
Object put (Object key, Object value);
```

As mentioned in the discussion of `TreeMap`, `put` returns either the value previously associated with `key`, or `null` if there was no previous association with `key`. The `put` method requires that keys support two operations:

- `public boolean equals (Object obj)`
- `public int hashCode ( )`
- `public boolean containsKey (Object obj)`

Here's how it uses them.

1. First, `put` calls `key.hashCode ( )` automatically to find out which collection to search for `key`. The programmer does not call `hashCode` him/herself. The relevant array index in the table maintained by the `HashMap` class is `key.hashCode ( ) % 10`, the length of the bucket array.
2. Then, it searches the "bucket"—the collection of entries whose keys all have the same hash value—using `equals`. If it finds an entry with the given key, it replaces the associated value; otherwise it adds the key/value entry to the collection.

The `get` method, which returns the value associated with a given key, works similarly. It (note: not the programmer) calls the key's `hashCode` method to find out which collection to search, then compares the key with the keys in the collection using `equals`. A possible ambiguity arises when `get` returns `null`: either the key is not in the table, or it is in the table but is associated with the value `null`. We resolve the ambiguity with the `containsKey` method, which returns `true` if its key argument is associated with something and returns `false` otherwise.

20.2.7 (Display page) More details for java.util.HashMap

The `HashMap` constructor has two optional arguments. One is the initial size of the bucket array. Normally, the array is expanded (doubling its size, similar to `Vector` and `ArrayList`) when the table gets too full. If we know approximately how many entries the table will contain, however, we can avoid the intermediate expansions. The other argument is the maximum load factor, the number of entries in the table divided by the array size. This specifies when the array should be expanded. As we have seen in other Java collection classes, we are allowed to restrict the keys and values in the table to certain types. To declare a `HashMap` whose keys are all `Strings` and whose associated values are all `Integers`, we provide the relevant types in brackets:

```
HashMap <String, Integer> table = new HashMap <String, Integer> ( );
```

20.2.8 (Display page) An animation of hashing into linked lists

An animation of hashing into linked lists

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
This applet illustrates the hashing process, using the `hashCode` method of the `String` class. Collisions are stored in a linked list as just described. After loading the applet, type words separated by blanks (with no duplicated) into the text field. The words "e h g a o r a d k w c t" yield interesting behavior. Then click "Step" repeatedly. Continuing to click "Step" after all the words are entered into the table results in some calls to `get`.

20.2.9 (Display page) Experiments with HashMap

Experiments with HashMap

The next few steps involve experimenting with a `HashMap`. You should work with a partner on these exercises.

20.2.10 (Display page) Experiment 1

Experiment 1

To prepare for this sequence of experiments, copy the contents of the directory

```
~cs61b/code/hash.experiments
```
to your directory or import it into your Eclipse workspace. The directory contains several files:

- `TestString.java` defines a class whose `hashCode` method you'll be experimenting with.
- `InstrumentedHashMap.java` defines a subclass of `HashMap` that uses a bucket array of size 1001 and a maximum load factor of 30, and provides information about distribution of `TestString` objects within the table.
- `InstrumentedHashMap2.java` is similar, and will be used in later experiments.
- `small.data` contains thirteen different words (three are duplicated).
- `words` contains the dictionary for the UNIX `spell` program.
- `short.words` contains the words in the dictionary that have five letters or fewer.
- `song.titles` contains names of song titles, including those in Mike Clancy's (vinyl) record collection.

Start with the `TestString` class. Take a few minutes to read the code. It is intended to put some entries—words plus their lengths—into a hash table. If you run it with the command-line argument `small.data`, you will notice incorrect behavior: the `equals` method has a bug. Fix the bug.

20.2.11 (Brainstorm) Results of experiment 1

What was the bug in the `equals` method, and how did you figure it out?

20.2.12 (Display page) Linear probing

Linear probing

Hashing into "buckets" is not the only method for dealing with collisions. In the "linear probing" strategy of handling collisions, keys are stored in the array rather than in a separate collection. A key's hash value is found, and then the array is searched linearly (wrapping around to the beginning of the array when it reaches the end) for the key. The search stops either when the table is full, when an empty space is found (meaning that the key is not yet in the table), or when the key is found. This applet demonstrates the linear probing algorithm. Click "New" to get it started, then click "Ins" to add some keys.

20.2.13 (Display page) Experiment 2

Experiment 2

Supply, in the main method of the `TestString` class, a sequence of calls that will determine if the `HashMap` class uses linear probing to locate a key.

20.2.14 (Brainstorm) Results of experiment 2

Describe your experiment, and explain how its result confirmed that linear probing is or isn't used by the `HashMap` class.

20.2.15 (Display page) Experiment 3

Experiment 3

Add statements to the main method of the `TestString` class that show whether or not all the keys with a given `hashCode` value are stored together in a collection. (If this is the case, adding an entry whose key has a given hash value will involve comparing the key with all the other keys with that hash value.)

20.2.16 (Brainstorm) Results from experiment 3

Describe your experiment, and explain how its result confirms that keys with identical hash values are or are not stored together in the `HashMap` class.

20.3 Experiment with hash functions.(17 steps)

20.3.1 (Display page) Experiment 4

Experiment 4

The next several experiments use the `InstrumentedHashMap` class, which simulates some aspects of the `HashMap` class in order to produce statistics about the behavior of the hashing process. The main method of `InstrumentedHashMap` takes two command-line arguments. The first is the name of a file of words to hash; the second is the name of a file in which to put counts of the various buckets. The information written to the file is most usefully displayed in a graph. The `gnuplot` program can be used to produce the graph. Run `gnuplot`, then give the command

```
plot "filename"
```

where `filename` is the name of the file written by `InstrumentedHashMap.main`. (You need the double-quote characters around the file name.) As before, we suggest that you work with a partner on these exercises. Here's the next experiment. The `HashMap` class's method supplied in the `TextString` class is a terrible one. Run the `InstrumentedHashMap` program using the `words` file and plot the graph from the produced information to verify this.

20.3.2 (Brainstorm) Results of experiment 4

Explain the problem with the `TestString` `hashCode` method, and how the graph shows that the hash function is bad.

20.3.3 (Display page) Experiment 5

Experiment 5
Rename the existing `TestString.hashCode` method and rename `hashCode2` to `hashCode`. Run the `InstrumentedHashMap` program on the `words` file and on the `song.titles` file, and compare the results.

20.3.4 (Brainstorm) Results of experiment 5
On which set of strings does the given hash function (the former `hashCode2`) perform better? What explains the difference in performance? In particular, what explains the "lumps" in the distribution in one of the graphs? (Hint: the ASCII values of the upper-case letters range from 65 to 90; the ASCII values for lower-case letters range from 97 to 122.)

20.3.5 (Display page) Experiment 6

Experiment 6
Now rename the `hashCode` method from the previous step and fix your class to use the `String.hashCode` method. Test the updated code with the three files `words`, `song.titles`, and `short.words`. Compare the graphs.

20.3.6 (Brainstorm) Results of experiment 6
How well does `String.hashCode` work on the various sets of words? In particular, how well does it work for short strings?

20.3.7 (Brainstorm) Results of experiment 6
How well does `String.hashCode` work on the various sets of words? In particular, how well does it work for short strings?

20.3.8 (Brainstorm) Thinking about inventing a hash function
Inventing a hash function
Here are eleven keys.

    10 100 32 45 58 126 1 29 200 400 15

Find a hash function that, when used with an empty chained table of size 11, produces three or fewer collisions when the eleven keys are added to the table.

20.3.9 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.10 (Brainstorm) Thinking about inventing a hash function
Inventing a hash function
Inventing a hash function
Here are eleven keys.

    10 100 32 45 58 126 1 29 200 400 15

Find a hash function that, when used with an empty chained table of size 11, produces three or fewer collisions when the eleven keys are added to the table.

20.3.11 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.12 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.13 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.14 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.15 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.16 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.17 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.18 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.19 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.20 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.21 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.22 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.23 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.24 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.25 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.26 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.27 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.28 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.29 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.30 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.31 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.32 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.33 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.34 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.35 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.36 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.37 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?

20.3.38 (Brainstorm) Thinking about inventing a hash function
Some students found the problem in the preceding step difficult. What was hard about it?
Think about why for the next question.

20.3.13 (Brainstorm) Question about hash table contents
In the program of the previous step, three calls to add were made. However, two of the calls apparently had no effect. Explain what’s going on here.

20.3.14 (Self Test) Relation of equals and hashCode
True or false: If for two objects a and b, the expression a.equals(b) is true, then it must be the case that a.hashCode() == b.hashCode() is also true.

True or false: If for two objects a and b, the expression a.hashCode() == b.hashCode() is true, then it must be the case that a.equals(b) is also true.

20.3.15 (Display page) Some real-world hash functions

Some real-world hash functions

Information presented here is taken from the article “Selecting a Hashing Algorithm”, B.J. McKenzie et al., Software Practice & Experience, vol. 20, no. 2, February 1990. The authors experimented with eight hash functions implemented in a variety of widely distributed software.

Hashing algorithms

All of the algorithms compute a hash value h for a string of length n whose characters are c1, c2, ..., cn. The hash value is determined from successive partial results h0, h1, h2, ..., hn, with each h

computed from h

as given in the formulas below. The hash table size is the value used in the mod operation at the end of each algorithm.

1. Amsterdam Compiler Kit (ACK)
   There is a "mask" for characters, built as follows:
   \[ m_1 = 171; m_2 = \text{rightmost 8 bits of } \text{7m}_{n-1}+153 \]
   The hash value h is then the last 8 bits of h

, where h

= 0 and h

= h

+ XOR(c

, m

).

2. Eidgenossische Technische Hochschule Modula-2 Cross Compiler (ETH)
   \[ h_0 = 1; h_k = c_k \cdot (h_{k-1} \text{ mod } 257)+1); H = h_k \text{ mod } 1699 \]

3. GNU C preprocessor (GNU-cpp)
   \[ h_0 = 0; h_k = 4h_{k-1}+c_k; H = \text{last 31 bits of } h_n \text{ mod } 1403 \]

4. GNU compiler front end (GNU-cc1)
   \[ h_0 = n; h_k = 613h_{k-1}+c_k; H = \text{last 30 bits of } h_n \text{ mod } 1008 \]

5. Portable C Compiler front end (PCC)
   \[ h_0 = 0; h_k = 2h_{k-1}+c_k; H = \text{last 15 bits of } h_n \text{ mod } 1013 \]

6. Unix 4.3 BSD C preprocessor (CPP)
   \[ h_0 = 0; h_k = 2h_{k-1}+c_k; H = h_k \text{ mod } 2050 \]

7. AT&T C++ compiler (C++)
   \[ h_0 = 0; h_k = 2h_{k-1}+c_k; H = h_k \text{ mod } 257 \]

8. Icon translator (Icon)
   \[ h_0 = 0; h_k = h_{k-1}+c_k; E = h_k \text{ mod } 128 \]

Performance

Algorithms were tested on 36,376 identifiers from a large bunch of C programs, and 24,473 words from a UNIX dictionary. ACK is a loser (U-shaped distribution), Icon, C++, GNU-cc1, and GNU-cpp seem to distribute the words well. Theoretical results suggest that an algorithm of the form h

= A*h

with A a power of 2 for speed and \( n \) chosen appropriately. The authors note:

"[s] and \( n \) need to be selected with care. Although it may seem unlikely that anyone would choose one of the really bad combinations, the facts ... indicate that far-from-optimal choices are made and persisted with. The experiments have shown that very small variations in \( n \) can produce large variations in the efficiency of the hash table lookup, and that the popular view, that choice of a prime number will automatically ensure a good result, is not well founded."

20.3.16 (Display page) Experimenting with three hash functions for strings

Experimenting with three hash functions for strings

Three classes, in files named TestString1.java, TestString2.java, and TestString3.java, extend the TestString Class and override its hashCode method. TestString1 uses a slight variation of the ETH hashing algorithm. TestString2 uses a slight variation of the GNU-cpp hashing algorithm. TestString3 uses a slight variation of the GNU-cc1 algorithm. A class that tests the three hashing methods is declared in InstrumentedHashSet2.java. Its main method takes three command-line arguments: the name of the file of words to be hashed, the size of the underlying array, and the name of the file in which to put information for gnuplot. (The information is placed in three files, one for each hashing method: name.1, name.2, and name.3, where name is the command-line argument you give.) Find out, by testing each of the three hash functions on hash table sizes between 1040 and 1050, which of the three performs best on the strings in the words file. In particular, find out what table sizes the functions work well for and what table sizes lead to poor performance. In general, the lower the average number of comparisons, the better, and for tables with averages approximately equal, the lower the standard deviation, the better. You can produce results for all the table sizes between 1040 and 1050 by using a loop in the C shell command language:

```
for (tableSize = 1040; tableSize <= 1050; tableSize++) {
    // Your code here...
}
```

---

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
foreach $i (1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050)
    echo "*** table size = $i"
    java InstrumentedHashSet2 words $i $i
    echo ""
end

This will create files named 1040.1, 1040.2, 1040.3, 1041.1, etc. that you can plot with gnuplot.
Recall that you plot a set of data in a file named data by giving the gnuplot to the UNIX shell, then
giving the command
gp -p plot "data"
to gnuplot.

20.3.17 (Brainstorm) Results
Report the results you noticed in the previous step. In particular, identify which of the three
algorithms perform best and worst, for what table sizes between 1040 and 1050 they perform
especially well, and for what table sizes they perform especially badly.

20.4 Homework (3 steps)
20.4.1 (Display page) Exam review requests + project post-mortem
Exam review requests plus project 2 post-mortem
For homework, provide a contribution to the activity in the next step. T.a.s will use your input to
devise review activities for the first lab next week. Also for homework, make two posts to the
following discussion by tonight (April 1 or 2): the topic is lessons learned from project 2. Then make
two responses to posts of your lab mates by your lab section on Thursday, April 3 or Friday, April 4.
There will be an exam in class (5-7pm) on Wednesday, April 9. It will cover material through the
quiz on April 8-9 (hashing will be tested on this exam). The t.a.s will run review activities on April 8
and 9. A good way to prepare, we think, is to review lab activities, especially those that gave you
trouble the first time around.

20.4.2 (Brainstorm) Topics needing review for the exam
Suggest a topic or technique to be covered on the exam that you would most like your t.a. to review
in lab on next Tuesday or Wednesday.

20.4.3 (Discussion Forum) Post mortem on project 2
Supply one post that describes the biggest mistake you made in designing, coding, or testing project 2.
Then supply another that describes something you did that was a big help in designing, coding, or
testing project 2. Do this by Saturday night, March 24. Then, before your next lab, make two
responses to posts of your labmates:

• comment on a mistake of one of your labmates by describing how you avoided the error he or
  she described;
• also comment on a positive thing that one of your classmates did (and you didn’t do) by
  noting how well that positive action would have worked for you.

Again, we’re hoping to help you avoid repeating your mistakes.

21 More on hashing
2008-4-03 ~ 2008-4-04 (4 activities)
21.2 Hash things that aren’t strings (work with a partner). (8 steps)

21.2.1 (Display page) Hashing tic-tac-toe boards
Hashing Tic-tac-toe boards
The code in ~cs61b/code/TTTHashTest.java stores all possible Tic-tac-toe boards into a
java.util.HashSet object. Add methods to the TTTBoard class that test two ways to hash Tic-tac-
toe boards. In one, you should convert the board to a String and then use the String hashCode
function. In the other, you should interpret the Tic-tac-toe board as a nine-digit base 3 numeral and
return the corresponding integer as the hash value. Compare the performance of these two
implementations.

21.2.2 (Brainstorm) Explaining the results
Which hash function performed better? Suggest why.

21.2.3 (Brainstorm) Qualities of a good hashCode method
List all the qualities of a good hashCode method.

21.2.4 (Brainstorm) Evaluating a binary tree hash function
Evaluate the following hashCode method for the BinaryTree class according to the qualities that you
and your labmates listed in the previous step. In the BinaryTree class:

```java
public int hashCode ( ) {
    if (myRoot == null) {
        return 17;
    } else {
        return myRoot.hashCode ( );
    }
}
```

In the TreeNode class:

```java
public int hashCode ( ) {
    int returnValue = myItem.hashCode ( );
    if (myLeft != null) {
        returnValue += myLeft.myItem.hashCode ( );
    }
    if (myRight != null) {
        returnValue += myRight.myItem.hashCode ( );
    }
    return returnValue;
}
```

21.2.5 (Brainstorm) Evaluating another binary tree hash function
Do the same for the following hashCode method.

In the BinaryTree class:

```java
public int hashCode ( ) {
    // same as in previous step
}
```

In the TreeNode class:

```java
public int hashCode ( ) {
    int returnValue = myItem.hashCode ( );
```
21.2.6 (Brainstorm) Evaluating a third binary tree hash function

Do the same for the following `hashCode` method. (This one is a bit tricky.) In the `BinaryTree` class:

```java
public int hashCode() {
    ... // same as in previous step
}
```

In the `TreeNode` class:

```java
public int hashCode() {
    int returnValue = super.hashCode(); // the machine address of this TreeNode
    if (myLeft != null) {
        returnValue += myLeft.hashCode();
    }
    if (myRight != null) {
        returnValue += myRight.hashCode();
    }
    return returnValue;
}
```

21.2.7 (Display page) Implementing memoization using hash tables

Implementing memoization using hash tables

You may recall the technique of memoization from CS 61A. It allows redundant recursive calls to be eliminated by saving the result of each new call and then looking the result instead of recomputing it. In pseudocode, here's how this would be done with the Fibonacci computation:

```java
int Fib(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else if (Fib(n) has already been computed) {
        return the saved computed value;
    } else {
        int temp = Fib(n-1) + Fib(n-2);
        save the pair n and temp;
        return temp;
    }
}
```

Here, the structure for saving `Fib` values could be an array of `N` values, where `N` is the largest value for which you would want to compute `Fib(N)`. In general, though, the recursive computation is more complicated; a `HashMap` object that uses the function argument as the key and associates it in the table with the result that the function is to return is generally wonderful for this application. Take for example a game-playing program that makes moves. It will search the tree of moves to find winning configurations:

- Search for a winning move.
- If the configuration has been seen before, return its value.
- If it's an immediately losing configuration, store the configuration and "loss" in the table and return "loss".
- If it's an immediately winning configuration, store the configuration and "win" in the table and return "win".

Otherwise, for each possible move, do the following:

- make the move;
- make a recursive call on Search;
- store the configuration and the value in the table;
- if the value is "win", return it;
- otherwise unmake the move and try another.

If all moves have been analyzed and none are winners, return "loss".

21.2.8 (Display page) The jug problem

The jug problem

Two CS 61B students have a jug filled with eight liters of Jolt Cola that they wish to divide evenly between them. They have two empty jugs with capacities of five and three liters respectively. These jugs are unmarked and no other measuring device is available. How can they accomplish the equal division? The file `/cs61b/code/JugSolver.fw.java` contains part of a program to solve this problem. The program takes six command-line arguments: the capacities of the three jugs, the contents of jugs 0 and 1, and the desired amount. It recursively searches to find a sequence of steps (each step pours one jug into another) that produces the desired amount. The program has a flaw, namely that it doesn't keep track of configurations that it has seen before; infinite recursion results. Without changing any of the existing code (except perhaps to change the value of the `DEBUGGING` variable), add statements that fix the program. In particular, you are to use a `HashMap` object to keep track of configurations already seen. Test your completed program on the arguments

```
8 5 3 8 0 4
```

which correspond to the problem stated above.

21.3 Compare hash tables and binary search trees (work with a partner)

(4 steps)

21.3.1 (Display page) Code for a binary search

Code for a binary search

Suppose that 31 distinct integers are arranged in ascending order in an array named `values`. Suppose also that the following code is used in a method to locate an integer named `key` in the array:

```java
int leftIndex = 0;
int rightIndex = values.length() - 1;
while (leftIndex <= rightIndex) {
    int middleIndex = leftIndex + (rightIndex - leftIndex) / 2;
    if (values[middleIndex] == key) {
        return middleIndex;
    } else if (key < values[middleIndex]) {
        rightIndex = middleIndex - 1;
    } else {
        leftIndex = middleIndex + 1;
    }
}
```

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary

Page 78 of 104
```java
int middleIndex = (leftIndex + rightIndex) / 2;
if (values[middleIndex] == key) {
    return true;
} else if (values[middleIndex] < key) {
    leftIndex = middleIndex + 1;
} else {
    rightIndex = middleIndex - 1;
}
return false;
```

The next two questions ask you to count the comparisons needed to find all 31 integers, and then to do the same thing with the values stored in a hash table.

21.3.2 (Brainstorm) Counting binary search comparisons

Compute the total number of comparisons necessary to locate all 31 values in the array using the above loop. The code is repeated below for your convenience; the lines containing comparisons to count appear in bold face.

```java
int leftIndex = 0;
int rightIndex = values.length() - 1;
while (leftIndex <= rightIndex) {
    int middleIndex = (leftIndex + rightIndex) / 2;
    if (values[middleIndex] == key) {
        return true;
    } else if (values[middleIndex] < key) {
        leftIndex = middleIndex + 1;
    } else {
        rightIndex = middleIndex - 1;
    }
}
return false;
```

21.3.3 (Brainstorm) Counting hash table comparisons

Now suppose that the 31 integers are stored in a chained hash table with \( k \) chains, in which the hash function distributes the integers to chains as evenly as possible. What is the smallest value of \( k \) for which the total number of comparisons to find all 31 values in the hash table is less than the answer you computed for part a? (Recall that \( 1 + 2 + \ldots + n = n(n+1)/2 \).) Briefly explain your answer.

21.3.4 (Brainstorm) Comparing hash tables with binary search trees

Invent a question about the design of a collection of items whose answer is one of the following:

- A hash table would be a better implementation for the collection
- A (balanced) binary search tree would be a better implementation for the collection
- Either a hash table or a (balanced) BST would be good implementations for the collection

You'll need to specify operations to be optimized for your collection. Here's an example.

Suppose you want to represent a large set of items so as to optimize the operation of determining if one set is a subset of another. How should you represent the set?

21.4 Homework (1 step)

21.4.1 (Display page) Exam preparation

Exam preparation

The second exam will be Wednesday, April 9, in class. Lab on April 8 and 9 will involve a quiz but no other structured activities; t.a.s may have review sessions planned, and will certainly be available for questions. All exam questions will be based on lab activities. More details about the exam are available here. Old exams and solutions are available online, accessible from the Resources link on the CS 61BL course portal.

22 Activities for April 8 and 9 2008-4-08 ~ 2008-4-09 (2 activities)

22.2 Activities (1 step)

22.2.1 (Display page) Exam preparation, plus a lookahead

Exam preparation, plus a look ahead

There are no other formal activities for today's lab other than the quiz. T.a.s will be available to answer questions, and may run more organized review sessions. Lab on Thursday and Friday will deal with the topics of priority queues and binary heaps. This material is covered in section 8.5 of Objects, Abstraction, Data Structures, and Design.

23 Priority queues and binary heaps 2008-4-10 ~ 2008-4-11 (5 activities)

23.2 Think about implementations of queues and priority queues (7 steps)

23.2.1 (Display page) Overall road map

Overall road map

This set of activities involves exploration of priority queues (queues in which each element also has a priority that might bring it to the head of the queue more quickly) and their implementation. A data structure called a max heap turns out to be particularly suitable for implementing a priority queue. Activities include analysis of the abstract definition of a max heap, as well as debugging code that attempts to implement the abstract definition. Finally, you'll consider an application of a heap. Priority queues are an integral component of many algorithms for graph processing (which we'll cover next week). For example, the first four weeks of CS 170 all involve graph algorithms that use priority queues.

23.2.2 (Display page) Queue operations

Queue operations

A queue (pronounced "cue") is a data type that's like a waiting line. The queue has a "front", the head of the line, and a "back". Its operations include the following:

- construct an empty queue;
- (enqueue, pronounced "en-cue") add an element to the back of the queue;
- (dequeue, pronounced "de-cue") remove the front element of the queue;
- see if the queue is empty;
- access the front of the queue.

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
We've used a queue already to enumerate elements of a tree in breadth-first order. Queues also appear in any application that uses a waiting line, e.g. processes waiting their turn to run on the computer, simulations of service organizations, etc.

23.2.3 (Brainstorm) Queue implementations

What data structure makes the operations of enqueueing, dequeueing, checking for empty, and accessing the front of the queue most efficient? Briefly explain your answer. Correct your answer if necessary after reviewing the responses of your labmates.

23.2.4 (Display page) Priority queues

Priority queues

A priority queue is a queue in which every member has an associated priority (basically, just a number). Operations are

- construct an empty priority queue;
- add an element to the queue;
- remove the element with the highest priority (or one of the elements with the highest priority if there are more than one);
- see if the queue is empty;
- access a queue element whose priority is highest.

23.2.5 (Brainstorm) Linked list implementation of a priority queue

Explain how to use a linked list to implement a priority queue, and give running time estimates for the enqueue and dequeue operations. Correct your answer if necessary after reviewing the responses of your labmates.

23.2.6 (Brainstorm) BST implementation of a priority queue

Explain how to use a binary search tree to implement a priority queue, and give execution-time estimates for the enqueue and dequeue operations. Correct your answer if necessary after reviewing the responses of your labmates.

23.2.7 (Brainstorm) More or less work to balance the tree?

Your answer to the previous question probably assumed that the binary search tree was balanced. In the implementation you described, would it be less difficult or more difficult to keep the tree balanced compared to a situation where elements were being randomly added to and deleted from a binary search tree? Briefly explain your answer. Correct your answer if necessary after reviewing the responses of your labmates.

23.3 Work with binary heaps.

23.3.1 (Display page) Binary max heaps

Binary max heaps

An even better choice for implementing a priority queue is a special kind of binary tree, named a binary max heap. A binary max heap has the following characteristics.

- The element at the root—the "top of the heap"—is the largest element, or one of the largest elements, if there is a tie. (A "min heap" has the smallest item at the top.)
- All the subtrees of a max heap are themselves max heaps. (Recall a similar recursive definition for binary search trees.)
- Finally, a max heap is a complete tree, that is, completely filled except maybe for the bottom level. The bottom level is filled from left to right, with no missing nodes. ("Complete" is used here, as in section 8.5 of Objects, Abstraction, Data Structures, and Design, to mean what we called "maximally balanced" in an earlier homework.)

The table below lists some complete and incomplete trees.

<table>
<thead>
<tr>
<th>complete trees</th>
<th>incomplete trees</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Complete Tree" /></td>
<td><img src="image2" alt="Incomplete Tree" /></td>
</tr>
</tbody>
</table>

In a complete tree of \( n \) nodes, the height is at most \( 1 + \log_2 n \). This is because a complete tree of height \( h \) has between \( 2^{h-1} \) and \( 2^h - 1 \) nodes. That means that we can expect logarithmic worst-case behavior if we restrict changes in the structure to one path from the root to a leaf.

23.3.2 (Brainstorm) Heaps and BSTs

Describe as many different nonempty binary trees as you can that are both max heaps and binary search trees. ("Different" means different in shape as opposed to key values.) Assume that the largest value is at the top of the heap and at the rightmost position in the binary search tree.

23.3.3 (Brainstorm) Trying to verify that a binary tree is complete

You may have discovered in the day14 homework that it's not obvious how to verify that a linked binary tree is complete (what we called "maximally balanced" in that homework). A CS 61B student suggests the following recursive algorithm:

- A one-node tree is complete.
- A tree with two or more nodes is complete if its left subtree is complete and has depth \( k \) for some \( \geq 2 \), and its right subtree is complete and has depth \( k \) or \( k-1 \).

Find a counterexample to this claim. Correct your answer if necessary after reviewing the responses of your labmates.
23.3.4 (Brainstorm) Another try
Another CS 61B student claims that a binary tree is complete if all the leaves in the tree come at the end of a breadth-first traversal of the tree. Give a counterexample to this claim. Correct your answer if necessary after reviewing the responses of your labmates.

23.3.5 (Display page) Using an array for heap elements

Using an array for heap elements

A binary tree can be represented in an array. The root of the tree would be array element 0; the two children of the root would be array elements 1 and 2; and so on, with array elements appearing breadth-first in the array. For example, the tree can be represented as the array [X, E, C, D, G, F, A, B, H, I].

This representation of course will waste memory if the tree isn’t very “bushy”. A max heap, however, is a complete binary tree, so we can store its elements in an array without wasting any space. Tree elements appear breadth-first in the array; element 0 is the top of the heap, elements 1 and 2 are its two children, and so on, with the children of element $k$ being elements $2k+1$ and $2k+2$. Thus if we have the position of a node in the tree, we can access its children in constant time. We can also access its parent in constant time; the parent of the element in position $k$ is at position $(k-1)/2$ (using integer division).

23.3.6 (Brainstorm) Accessing the third biggest item in a heap

Which elements of the underlying array might contain the third largest element of a max heap? Briefly explain your answer. Assume for simplicity that the heap does not contain any duplicate elements. Correct your answer if necessary after reviewing the responses of your labmates.

23.3.7 (Display page) Inserting and deleting an element

Inserting and deleting an element

Insertion and deletion both require attention to preserving the heap properties:

- the heap is a complete tree;
- the top of the heap is the largest element in the heap; and
- all the subtrees are heaps.

(We assume that we’re working with max heaps here.) Insertion works as follows. The element to be inserted is added to the end of the heap array, thereby extending its size by 1. It is not necessarily in its proper place in the heap, however, so it is “bubbled up” the tree until it’s less than its parent item. Deletion proceeds by replacing the top of the heap by the last element and reducing the size by 1. Then the top of the heap is “bubbled down” to its proper position by exchanging it with its larger child. A complete example of how elements are inserted and deleted appears here.

23.3.8 (Display page) Experimenting with an animation

An animation of heap insertion/deletion

The web site http://www.student.seas.gwu.edu/~idsv/idsv.html provides an animation of the heap insertion and deletion operations. We suggest you work with a partner to experiment with the animation. Go to the web site, type your name, and click on “Priority queue” in the sidebar. The button labeled “Start Visualization” at the bottom right of the window starts the animation applet. Click “Single Step” and “I’ll Pick” to gain full control of the animation. Enqueue 14, 15, and 50 to create a small heap. (It only accepts one- and two-digit integers.) Then try enqueuing a value that’s larger than 50, another between 50 and 15, and a third that’s less than 14. The “Step” button takes you through each step of the insertion. Then dequeue three elements. The “I’ll Try” button provides a self-check facility for how the heap gets restored at each enqueue and dequeue operation. When you’re done experimenting, click the “Stop Visualization” button in the browser window.

23.3.9 (Display page) Debugging a Heap class

Debugging a heap class

The file /cs61b/code/Heap.java partially implements a max heap of integers. The main method initializes a heap from a file whose name is provided as a command-line argument. (Integers appear one per line in the file.) It then checks that the values are in heap order, and repeatedly reads and inserts values into the heap. There is a bug in the main method and another in the insert method. Find and fix the bugs, changing the existing code as little as possible.

23.3.10 (Display page) Queue and PriorityQueue in java.util

The java.util library contains a Queue interface and a PriorityQueue class that implements Queue. Queue methods include the following:

- offer (element), which adds the element to the queue;
- peek (), which returns but does not remove the head of the queue;
- poll (), which retrieves and removes the head of the queue (returning null if the queue is empty).

The LinkedList class also implements Queue. PriorityQueue is implemented using binary min heaps. One might guess that from the online documentation:

"If multiple elements are tied for least value, the head is one of those elements—ties are broken arbitrarily. ... The Iterator provided in method iterator () is not guaranteed to traverse the elements of the PriorityQueue in any particular order... Implementation note: this implementation provides $O(\log(n))$ time for the insertion methods (offer, poll, remove () and add) methods; linear time for the remove(Object) and contains(Object) methods; and constant time for the retrieval..."
Key Concepts

- **Directed vs. undirected graphs**
- **Graph nodes**
- **Graph edges**

Overview of graphs

Graphs are a representation used to show relationships between elements of a collection. First, a few definitions. An element in a graph is called a **vertex**. Vertices typically represent the objects in our graph, namely the things that have the relationships such as people, places, or things. A connection between two vertices is called an **edge**. Basically, an edge represents some kind of relationship between two vertices in a given graph. Quite a few examples of graphs exist in the everyday world:

- **Road maps** are a great example of graphs. Each city is a vertex, and the edges that connect these cities are the roads and freeways. An abstract example of what such a graph would look like can be found [here](http://www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heap.html). For a more technical example, a computer network is also a graph. In this case, computers and other network machinery (like routers and switches) are the vertices, and the edges are the network cables. Or for a wireless network, an edge is an established connection between a single computer and a single wireless router.

- **Online social networks**, such as MySpace, Orkut, and Friendster are a recent phenomenon. Each person that participates is a vertex, and an edge is established when two people claim to be friends or associates of each other.

- For a more technical example, a computer network is also a graph. In this case, computers and other network machinery (like routers and switches) are the vertices, and the edges are the network cables. Or for a wireless network, an edge is an established connection between a single computer and a single wireless router.

In more formal mathematical notation, a vertex is written as a variable, such as $v_0, v_1, v_2$, etc. An edge is written as a pair of vertices, such as $(v_0, v_1), (v_2, v_0)$.

24.2.2 (Display page) Directed vs. undirected graphs

Directed vs. undirected graphs

Once again, an element in a graph is called a **vertex**, and a connection between two vertices is called an **edge**. If all edges in a graph are showing a relationship that works in either direction, then it is called an undirected graph. A picture of an undirected graph looks like this:

The **heap sort** algorithm to sort array values into ascending order has two phases. First, it creates a max heap out of the values. Then, it repeatedly removes the top element from the heap and places it at the front of the sorted sequence. There is a way to do this in place in the array that's explained in Objects, Abstractions, Data Structures, and Design, section 10.8. The web site accessible through the "Creating a heap all at once" step —[http://www.cse.litk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heap.html](http://www.cse.litk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heap.html)—also provides an animation of the heap sort algorithm.

24.2.4 (Brainstorm) A way to sort with heaps?

A CS 61B student suggests the following method for printing the elements of an array in ascending order. First, make the array into a heap (whose maximum element is the top element). Then, simply traverse the heap in postorder (traversing the left subtree before the right), printing each node value as it is encountered in the traversal. Unfortunately, the student is confused. Give an example for which the algorithm doesn't work.

A better way to sort with heaps

The heap sort algorithm to sort array values into ascending order has two phases. First, it creates a max heap out of the values. Then, it repeatedly removes the top element from the heap and places it at the front of the sorted sequence. There is a way to do this in place in the array that's explained in Objects, Abstractions, Data Structures, and Design, section 10.8. The web site accessible through the "Creating a heap all at once" step —[http://www.cse.litk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heap.html](http://www.cse.litk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heap.html)—also provides an animation of the heap sort algorithm.

24.2.4 (Brainstorm) Isolating the smallest K out of N elements

Describe an algorithm that uses a heap of size $K+1$ to collect the $K$ smallest elements in a list of length $N$ in time proportional to $N \log K$. Your description should be in detail sufficient for another CS 61B student to recognize how your solution should be implemented in Java.

23.5 Homework (3 steps)

23.5.1 (Display page) A start on project 3

A start on project 3

Your t.a. will ask you on next Tuesday or Wednesday about who you'll be working in partnership, and how you intend to proceed on the project. Your answers to these questions will earn you up to 2 homework points. You should also supply a post and a response to each of the two discussions that follow.

23.5.2 (Discussion Forum) Questions about project 3

In a post (by Sunday night), supply a question that you or a labmate have about project 3. Then, provide a response by answering one of your labmates' questions or by significantly elaborating on an answer already provided.

23.5.3 (Discussion Forum) Likely challenges posed by project 3

In a post (by Sunday night), make a guess about what aspect of project 3 will be most difficult, and how you intend to proceed on the project. Your answers to these questions will earn you up to 2 homework points.

23.4.4 (Brainstorm) Isolating the smallest K out of N elements

Describe an algorithm that uses a heap of size $K+1$ to collect the $K$ smallest elements in a list of length $N$ in time proportional to $N \log K$. Your description should be in detail sufficient for another CS 61B student to recognize how your solution should be implemented in Java.

23.4.3 (Display page) A better way to sort with heaps

A better way to sort with heaps

The heap sort algorithm to sort array values into ascending order has two phases. First, it creates a max heap out of the values. Then, it repeatedly removes the top element from the heap and places it at the front of the sorted sequence. There is a way to do this in place in the array that's explained in Objects, Abstractions, Data Structures, and Design, section 10.8. The web site accessible through the "Creating a heap all at once" step —[http://www.cse.litk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heap.html](http://www.cse.litk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heap.html)—also provides an animation of the heap sort algorithm.

23.4.2 (Brainstorm) A better way to sort with heaps?

A CS 61B student suggests the following method for printing the elements of an array in ascending order. First, make the array into a heap (whose maximum element is the top element). Then, simply traverse the heap in postorder (traversing the left subtree before the right), printing each node value as it is encountered in the traversal. Unfortunately, the student is confused. Give an example for which the algorithm doesn't work.

A better way to sort with heaps

The heap sort algorithm to sort array values into ascending order has two phases. First, it creates a max heap out of the values. Then, it repeatedly removes the top element from the heap and places it at the front of the sorted sequence. There is a way to do this in place in the array that's explained in Objects, Abstractions, Data Structures, and Design, section 10.8. The web site accessible through the "Creating a heap all at once" step —[http://www.cse.litk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heap.html](http://www.cse.litk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heap.html)—also provides an animation of the heap sort algorithm.

23.4.1 (Display page) Creating a heap all at once

Creating a heap all at once

Suppose you have an array of $N$ values, and you want to turn them into a max heap. A straightforward approach is to insert them one by one into a heap that starts out empty. Since each insertion takes proportional to $\log N$ comparisons once the heap gets large enough, the whole process takes proportional to $N \log N$ time. However, there is a faster way that runs in time proportional to $N$, involving the conversion of the collection of values to a max heap all at once. We start from the end of the array, building first tiny heaps, then combining them into larger heaps. (This is the first phase of a sorting algorithm named “Heapsort” that we’ll revisit shortly.) Objects, Abstractions, Data Structures, and Design doesn’t mention it, but Wikipedia has the code and accompanying discussion. The process is animated [here](http://www.cse.litk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heap.html).

23 Homework (5 activities)

23.2 Work with the abstract concept of a graph.

24.2 (Display page) Directed vs. undirected graphs

Directed vs. undirected graphs

Once again, an element in a graph is called a **vertex**, and a connection between two vertices is called an **edge**. If all edges in a graph are showing a relationship that works in either direction, then it is called an undirected graph. A picture of an undirected graph looks like this:
But not all edges in graphs are the same. Sometimes the relationships between two vertices sometimes only go in one direction. Such a relationship is called a directed graph. An example of this could be a city map, where the edges are sometimes one-way streets between locations. A two-way street would actually have to be represented as two edges, one of them going from location A to location B, and the other from location B to location A. In terms of a visual representation of a graph, an undirected graph does not have arrows on its edges (because the edge connects the vertices in both directions), whereas each edge in a directed graph does have an arrow that points in the direction the edge is going. An example directed graph appears below.

More formally, an edge from a vertex \( v_0 \) to a vertex \( v_1 \) is printed as the pair \( (v_0, v_1) \). In a directed graph, the pair is ordered; thus \( (v_0, v_1) \) might be an edge, but \( (v_1, v_0) \) might not. In an undirected graph, the pair isn’t ordered, so the edge \( (v_0, v_1) \) is the same as the edge \( (v_1, v_0) \).

24.2.3 (Brainstorm) More situations that can be represented by graphs

Give another situation that can be modeled with graphs. Describe what the vertices are, and define the conditions under which two vertices are connected by an edge. Can your example be represented as an undirected graph, or does it have to be a directed graph?

24.2.4 (Brainstorm) Graphs vs. trees

An Amoeba family tree may be interpreted as a graph. What are the graph's vertices, and what defines when two vertices \( v \) and \( w \) are adjacent?

24.2.5 (Display page) A few more definitions

Now that we have the basics of what a graph is, here are a few more definitions that apply to graphs that will come in handy while discussing graphs. When an edge \( (v_0, v_1) \) exists between vertices \( v_0 \) and \( v_1 \), then \( v_1 \) is said to be adjacent to or connected to \( v_0 \), and a neighbor of \( v_0 \). Basically, “adjacent” is the official term for saying that a vertex is pointed to by another vertex, but we also refer to the “set of neighbors” or the “neighborhood” of a vertex. In addition, in the above example, the vertices \( v_0 \) and \( v_1 \) are said to be incident with edge \( (v_0, v_1) \). So when an edge is touching a vertex, that vertex is incident with that edge. A path between two vertices is a sequence of edges that can be followed from one vertex to another. A special kind of path is called a cycle, which is a path that ends at the same vertex where it originally started. The existence of cycles are one reason why graphs are more complicated than trees. Finally, a graph is connected when for every pair of vertices \( u \) and \( v \), there is a path from \( u \) to \( v \).

24.2.6 (Self Test) Edge count vs. vertex count

Suppose that \( G \) is an directed graph with \( n \) vertices. What’s the maximum number of edges that \( G \) can have? Assume that a vertex cannot have an edge pointing to itself, and that for each vertex \( u \) and vertex \( v \), there is at most one edge \( (u,v) \).

Now suppose the same graph \( G \) in the above question is an undirected graph. Again assume that no vertex is adjacent to itself, and at most one edge connects any pair of vertices. What’s the maximum number of edges that \( G \) can have?

What's the minimum number of edges that a connected undirected graph with \( n \) vertices can have?

24.2.7 (Display page) Data structures for graphs

Data structures for graphs

Now that we know how to draw a graph on paper and understand the basic concepts and definitions, we can consider how a graph should be represented inside of a computer. We need quick answers for the following questions about a graph:

- Are vertices \( v \) and \( w \) adjacent?
- Is vertex \( v \) incident to edge \( e \)?
- What vertices are adjacent to \( v \)?
- What edges are incident to \( v \)?

Imagine that we want to represent a graph that looks like this:
Probably the easiest-to-understand representation is called an adjacency list. In such a data structure, an array is created that has the same size as the number of vertices in the graph. Then, each location in the array points to a list that contains indexes to other vertices. It looks like this:

```
0     1     2     3     4
0  false  true  true false false
1  false false false  true false
2   true false false false  true
3  false false  true false false
4  false false  true false false
```

Another way to represent the edges in a graph is to use an adjacency matrix. In this data structure, it works the same way as an adjacency list, but each linked list is replaced with another array of the same length as the number of vertices in the graph. This matrix just contains boolean values, true when there is an edge between the two given vertices, false when no edge exists. Thus, each vertex has a row and a column in the matrix, and the value in that table says true or false whether or not that edge exists. Such a representation looks like this:

```
0   1   2   3   4
0  false true true false false
1  false false false  true false
2   true false false false  true
3  false false  true false false
4  false false  true false false
```

24.2.8 (Display page) Visualize graph data structures
Here is a useful applet we've created to help visualize a graph, given an adjacency matrix or adjacency lists. For the matrix, a checkbox that is turned off represents a false value in the matrix, and a checkbox turned on represents a true value. In the lists, type the numbers of the vertices, separated by spaces. Experiment with it to get a better understanding of what a corresponding graph looks like with a given adjacency matrix or a given set of adjacency lists. If you want, you can open up the applet in a separate window by clicking here.

24.2.9 (Self Test) Time estimates for accessing graph information
Using an adjacency matrix, how long in the worst case does it take to determine if vertex v is
adjacent to vertex v? (Assume vertices are represented by integers.)

Using an array of adjacency lists, how long in the worst case does it take to determine if vertex v is adjacent to vertex w? (Assume vertices are represented by integers.)

### 24.2.10 (Brainstorm) Memory use

Suppose we are representing a graph with v vertices and e edges. The memory required to store an adjacency matrix is $v^2$ times the memory required to store a boolean value. How much memory is required to represent the graph as an array of adjacency lists? Assume that references and integers each use 1 unit of memory. Briefly explain your answer.

### 24.2.11 (Brainstorm) A time estimate for finding a common neighbor

A common neighbor c between two vertices v and w is a vertex that has edges $(v,c)$ and $(w,c)$. In other words, both v and w both have an edge that points to c. In a graph with v vertices and e edges that is implemented using an adjacency matrix, how many comparisons are necessary in the worst case to determine if vertices v and w have a common neighbor? (Assume that v and w are represented as integers.) Briefly explain your answer.

### 24.2.12 (Brainstorm) Another estimate for neighbors in common

A CS 61B student claims that, in a graph with v vertices and e edges that’s implemented with an array of adjacency lists, the worst-case time to see if vertices v and w have a common neighbor is proportional to $v^2$. (Vertices are not in any particular order in an adjacency list.) This estimate is insufficiently specific. Explain why, and give a more specific estimate.

### 24.2.13 (Self Test) Adjacency lists vs. adjacency matrix

Imagine you have a graph with very few edges. Another way to put that is that each vertex almost always is adjacent to most of the other vertices in the graph. Which data structure is best for representing such a graph?

Imagine you have a graph with very few edges. Some vertices may not even be connected to other vertices, and most vertices are adjacent to only one or two other vertices. (Another term for this kind of graph is a sparse graph.) Which data structure is best for representing such a graph?

### 24.3 Work with programs that process graphs. (4 steps)

#### 24.3.1 (Display page) Graph traversal

**Graph traversal**

Earlier in the course, we used the general traversal algorithm to process all elements of a tree:

```java
Stack fringe = new Stack();
fringe.push(myRoot);
while (!fringe.isEmpty()) {
    TreeNode node = fringe.pop();
    // process the node
    [do whatever processing operation here...]
    if (node.myLeft != null) {
        fringe.push(node.myLeft);
    }
    if (node.myRight != null) {
        fringe.push(node.myRight);
    }
    // Note: If this was a tree with more than
    // two children, we'd push ALL of the
    // children onto the stack.
}
```

The code just given returns tree values in depth-first order. If we wanted a breadth-first traversal of a tree, we'd replace the Stack with a queue (the LinkedList class in java.util would work well here). Analogous code to process every vertex in a connected graph is:

```java
Stack stack = new Stack();
fringe.push (some vertex);
while (!fringe.isEmpty()) {
    // select a vertex from fringe
    TreeNode node = fringe.pop();
    // process the vertex
    [do whatever processing operation here...]
    // add the vertex's neighbors to fringe
    for (TreeNode next : node.myNeighbors) {
        fringe.push (next);
    }
}
```

This doesn't quite work, however. Applied to a cycle, for example, the code loops infinitely. The fix is to keep track of vertices that we've visited already, in order not to process them twice. Here is pseudocode:

```java
Stack fringe = new Stack();
fringe.push (some vertex);
while (!fringe.isEmpty()) {
    // select a vertex from fringe
    TreeNode node = fringe.pop();
    // process the vertex
    [do whatever processing operation here...]
    // add the vertex's unvisited neighbors to fringe
    for (TreeNode next : node.myNeighbors) {
        if (!fringe.contains (next)) {
            fringe.push (next);
        }
    }
}
```

As with tree traversal, we can visit vertices in depth-first or breadth-first order merely by choosing a stack or a queue to represent the fringe. Typically though, because graphs are usually interconnected, the ordering of vertices can be scrambled up quite a bit. Thus, we don't worry too much about using a depth-first or a breadth-first traversal. Instead, in the next lab session we will see applications that use a priority queue to implement best-first traversal.

#### 24.3.2 (Display page) Using an iterator to find a path, part 1

**Using an iterator to find a path, part 1**

The file `cs61b/code/Graph.java` contains an implementation of a graph as an array of adjacency lists, and includes a method for depth-first iteration. Add a method `pathExists` (int v, int w) that returns whether or not any path exists that goes from vertex v to vertex w. Remember that a path is any set of edges that exist which you can follow such that you can travel from one vertex to another. Your method should call `visitAll`. For an example of an undirected graph this should work on, try testing it with the following two graphs:

```java
addUndirectedEdge(0,3);
addUndirectedEdge(0,2);
addUndirectedEdge(1,4);
addUndirectedEdge(1,5);
```
addUndirectedEdge(2,3);
addUndirectedEdge(2,6);
addUndirectedEdge(4,5);
addEdge(0,1);
addEdge(1,2);
addEdge(2,0);
addEdge(2,3);
addEdge(4,2);

24.3.3 (Display page) Using an iterator to find a path, part 2

Using an iterator to find a path, part 2

Now you will actually find a path from one vertex to another if it exists. Write a method named `path` that, given two ints that represent a start vertex and a finish vertex, returns an `ArrayList` of `Integers` that represent vertices that lie on the path from the start to the finish. If no such path exists, you should return an empty `ArrayList`. Pattern your method on `visitAll`, with the following differences. First, add code to stop calling `next` when you encounter the finish vertex. Then, trace back from the finish vertex to the start, by first finding a visited vertex u for which (u, finish) is an edge, then a vertex v visited earlier than u for which (v, u) is an edge, and so on, finally finding a vertex w for which (start, w) is an edge. Collecting all these vertices in the correct sequence produces the desired path.

24.3.4 (Display page) Topological sort

Topological sort

A topological sort of a directed graph is a list of the vertices in such an order that if there is a directed path from vertex v to vertex w, then v precedes w in the list. (The graph must be acyclic in order for this to work. Directed acyclic graphs are common enough to be referred to by their acronym: DAGs.) Here is an example of a DAG:

```
A B C D E F G H
A C E G D F H
B E G A D C F H
```

In the above DAG, a few of the possible topological sorts could be:

```
A B C D E F G H
A C B E G D F H
B E G A D C F H
```

Notice that the topological sort for the above DAG has to start with either A or B, and must end with H. Another way to think about it is that the vertices in a DAG represent a bunch of tasks you need to complete on a to-do list. Some of those tasks cannot be completed until others are done. For example, when getting dressed in the morning, you may need to put on shoes and socks, but you can’t just do them in any order. The socks must be put on before the shoes. The edges in the DAG represent dependencies between the tasks. In this example above, that would mean that task A must be completed before tasks C and D, task B must be completed before tasks D and E, and so on. The topological sort algorithm uses an array named `currentInDegree` with one element per vertex. `currentInDegree[v]` is initialized with the in-degree of each vertex v; as each vertex is added to the result list, the `currentInDegree` values of its neighbors are reduced by 1. The fringe is initialized with all the vertices whose in-degree is 0. Once the `currentInDegree` of a vertex becomes 0, that means that all of the vertices that had edges pointing to it have already been processed. Thus, a vertex is added to the fringe when its `currentInDegree` becomes 0. Your task is to fill in the blanks in the `TopologicalIterator` class so that it successively returns vertices in topological order as described above. The `TopologicalIterator` class will resemble the `VertexIterator` class, except that it will use a `currentInDegree` array as described above, and instead of pushing unvisited vertices on the stack, it will push only vertices whose in-degree is 0.
24.4 Talk to your t.a. about project 3 (1 step)
24.4.1 (Display page) Project checkoff

Project checkoff

Your lab t.a. will want to chat with you about project 3 today. In particular, he or she will want to
know who you are working in partnership with, and how you plan to proceed. Two homework points
will be awarded for this checkoff. Regardless of your score, however, you should take this
opportunity to ask your t.a. about whatever aspects of the project are confusing. Your partner,
incidentally, may be any 61B student, not just someone in your lab section. (The only disadvantage
of a cross-lab partnership is that both t.a.s will be keeping track of your progress.) Take care when
choosing a partner; this is a substantial project.

24.5 Readings (1 step)
24.5.1 (Display page) Readings

Readings

Chapter 12 of Objects, Abstraction, Data Structures, and Objects contains material relevant to next
lab: Graph algorithms and applications

25 Graph algorithms and applications 2008-4-17 ~ 2008-4-18 (5 activities)
25.2 Continue work with traversal-based algorithms (in partnership), (14 steps)

25.2.1 (Display page) What you'll be doing

The next set of activities all involve the use of code in the Objects, Abstractions, Data Structures,
and Design textbook. This code is available online in the directory ~cs61b/code/graphs. It consists
of eight files:

- Graph.java — an interface for graph operations; different from the Graph.java you worked
  with last lab
- AbstractGraph.java — extended by ListGraph.java and MatrixGraph.java (not provided)
- ListGraph.java — implementation of a graph using an array of adjacency lists
- Edge.java — implementation of an edge
- BreadthFirstSearch.java
- DepthFirstSearch.java
- DijkstrasAlgorithm.java
- TopologicalSort.java — in case you want to compare your own implementation of this
  algorithm from last lab

We suggest you work in partnership on the remaining exercises in this section.

25.2.2 (Display page) Review of traversal-based algorithms

Review of traversal-based algorithms

Last week, you were introduced to graphs and their representations, then moved on to the following
basic graph iterator class:

Constructor:
initialize a 'fringe' set to contain one or more vertices of the graph;

Do more vertices remain?
when fringe isn't empty.

Get next vertex:
choose a vertex v from the fringe;
add its unvisited neighbors to the fringe;
return v.

A variety of algorithms for processing graphs are based on this kind of iteration. We've already seen
two of them:

- Determining if a path exists between two different vertices.
- In a topological sort of a directed acyclic graph, we wish to arrange the vertices into a
  sequence in which a vertex precedes all its neighbors. Here, we started the fringe with all the
  vertices with in-degree 0. As each vertex was visited, we conceptually removed it from the
  graph; the removal possibly resulted in more vertices with in-degree 0, and these were added
to the fringe.

Neither algorithm depended on the representation of the fringe. Either depth-first traversal (using a
stack) or breadth-first traversal (using a queue) worked for determining if a path existed. Neither of
them offered any benefit over the other. Similarly, the topological sort used a fringe that could have
been either a stack or a queue without any major difference. We're now going to investigate some
algorithms where the ordering of the fringe does matter.

25.2.3 (Display page) A first step: keeping track of extra information

A first step: keeping track of extra information

Recall the exercise from last lab where you located a path from a "start" vertex to a "stop" vertex. A
solution to this exercise involved building a traversal, then filtering the vertices that were not on the
path. Here was the suggested procedure:

... Then, trace back from the finish vertex to the start, by first finding a visited vertex u
for which (u, finish) is an edge, then an earlier visited vertex v for which (v, u) is an
edge, and so on, finally finding a vertex w for which (start, w) is an edge. Collecting all
these vertices in the correct sequence produces the desired path.

If each fringe element contains not only a vertex but also the vertex's predecessor along the
traversal path, we can make the construction of the path more efficient. Instead of search for the
previous vertex along the path, we merely follow the predecessor links. The file
~cs61b/code/graphs/BreadthFirstSearch.java contains an implementation of breadth-first search
in a graph that keeps track of each vertex's predecessor on the breadth-first path from the start
vertex. Copy this file to your own directory, along with the files ~cs61b/code/graphs/Graph.java
and ~cs61b/code/graphs/Edge.java. (This is not the Graph.java you used last lab.) Modify this
code in two ways:
First, add a main method that traverses, breadth-first, the vertices of the graph in Figure 12.16 of Objects, Abstraction, Data Structures, and Design. This graph is undirected; its edges are (0,1), (0,3), (1,4), (1,7), (2,3), (2,8), (2,9), (4,5), (4,6), (4,7), (6,7). This method will build a graph with the given edges, then call the breadthFirstSearch method to explore the graph breadth first. You should also add some code to print the contents of the queue each time through the loop.

Next, recode the breadthFirstSearch method as a method named path. It will have an additional parameter named destination. Instead of returning the array of parents, it should return an ArrayList specifying the vertices on the path from the start vertex to the destination. Also modify the code so that it stops when the destination vertex is reached. The main method should then print the contents of the returned ArrayList. Test your code by finding a path from vertex 0 to vertex 5.

25.2.4 (Display page) Associating distances with edges

Associating distances with edges

In graph applications we've dealt with already, edges were used only to connect vertices. A given pair of vertices were either connected by an edge or they were not. Other applications, however, processes edges with weights, nonnegative numeric values. In today's exercises, the weight associated with an edge will represent the distance between the vertices connected by the edge. In other algorithms, a weight might represent the cost of an edge or the time needed to process the edge. A graph with edge weights is shown below. Observe the weights on the edges (the small numbers in the diagram), and note that the weight of an edge (v,w) need not be equal to the weight of the edge (w,v).

25.2.5 (Self Test) Finding a shortest path

Given below is the graph from the previous step. Suppose that the weight of an edge represents the distance along a road between the corresponding vertices. (A distance might not be the same going, say, north to south as in the opposite direction.) The weight of a path would then be the sum of the weights on its edges. What is the shortest path—the one with the minimum distance sum—that connects vertex 0 with vertex 2? Give your answer as a sequence of vertex numbers separated by a blank. Thus the path v0 4 3 connecting vertex 0 with vertex 3 would be represented as 0 4 3.

25.2.6 (Display page) Dijkstra's shortest path algorithm

Dijkstra's shortest path algorithm

An algorithm devised by Edsger Dijkstra, a noted computer scientist, finds the shortest path between two vertices in a systematic way. Here, we will implement the algorithm using a modified version of the VertexIterator class, so that each call to next returns a vertex v for which the shortest path from the start vertex to v is known for sure. This requires keeping track, for each fringe vertex w, how far w is guessed to be from the start along with its predecessor along the shortest path from the start. Here's the algorithm.

Constructor:

1. Put the start vertex into the fringe. Its distance from the start is 0, and it doesn't have a predecessor.

hasNext

1. return !fringe.isEmpty();

next

1. Remove and save the vertex v in the fringe for which the distance from start is minimal. (One can prove that for every vertex v so chosen, the shortest path from the start vertex to v is
known for sure.

2. Then, for each unvisited neighbor $w$ of $v$, do the following:
   a. If there is not yet an entry in the fringe for $w$, create one: the predecessor is $v$, and the
distance from start is the distance from start to $v$ plus the weight of the edge $(v,w)$.
   b. If there is already a fringe entry for $w$, it might need updating; a path through $v$ to $w$
might be a shorter path than what we've seen so far. In that case, replace the distance
for $w$'s fringe entry by the distance from start to $v$ plus the weight of the edge $(v,w)$,
and replace its predecessor by $v$.

3. Return the fringe entry for $v$.

The fringe consists of one entry for each candidate vertex $v$ for which a path from start to $v$—not
necessarily the shortest path—has been found. Every time a vertex is removed from the fringe, that
vertex's shortest path has been found and it is finalized. The algorithm ends when the stop vertex is
returned by next.

25.2.7 (Display page) An animation

An animation of the Dijkstra algorithm is available online. (The "Run the animation" button is near
the bottom of the window.) It differs slightly from the version just described; try to keep track of the
differences as you work with the animation. The algorithm code refers to a "relax" procedure that
updates the distance for a fringe entry.

25.2.8 (Brainstorm) The fringe in the animation

What color are the vertices in the fringe? What color are vertices finalized by removal from the
fringe?

25.2.9 (Display page) A sample run of the Dijkstra algorithm

A sample run of the Dijkstra algorithm

Copy the remaining files from the ~cs61b/code/graph directory, and add a main program to
DijkstrasAlgorithm.java that runs the program on the graph in Figure 12.26 (shown below) to
find the shortest path from vertex 0 to vertex 4.

25.2.10 (Brainstorm) The fringe, in the book's version of Dijkstra's algorithm

Describe, as completely as possible, what vertices in the book's version of Dijkstra's algorithm
represent what we've been calling the "fringe".

25.2.11 (Brainstorm) Reason for using a HashSet

The book's version of Dijkstra's algorithm uses a HashSet to store the set $vMinusS$, namely, those
vertices whose shortest paths from the start vertex have not been finalized. Which use of $vMinusS$
gains the most efficiency from the HashSet representation, compared to (say) a linked list or an
array?

25.2.12 (Brainstorm) Use of a min heap

Use of a min heap would speed up Dijkstra's algorithm even more. How should a min heap be used
to do this?

25.2.13 (Display page) Counting distance updates

Counting distance updates

Add a parameter to the dijkstrasAlgorithm method, namely an array named distUpdates that
keeps track of how many times each vertex's dist and pred entries are updated. This array will
contain one entry per vertex and will be initialized to all zeroes. When the dist and pred entries get
updated at the end of the main loop, the corresponding distUpdates entry should be incremented.
Then devise a graph for which, when the Dijkstra algorithm is run, one of the vertices gets updated
four times.

25.2.14 (Brainstorm) A buggy shortest-path implementation

The book's implementation, each time through the main loop, finalizes the vertex in the fringe whose
distance from the start vertex is minimum. Consider the following alternative:

- Finalize the vertex in the fringe that's closest to some already finalized vertex.

Here's how the strategy would work with the graph in Figure 12.26:
1. The fringe starts with $[0]$.
2. $0$ is removed from the fringe and finalized, and vertices 1, 3, and 4 are added to the fringe.
3. Vertex 1, which is closest to a finalized vertex, is removed from the fringe. Vertex 2 is added.
4. Candidates for finalizing are vertex 2 (connected to finalized vertex 1 by an edge of distance 50), vertex 3 (connected to vertex 0 via an edge of distance 30), and vertex 4 (connected to vertex 0 via an edge of distance 100). Vertex 3 is removed from the fringe and finalized.
5. Vertex 2, connected to vertex 3 by an edge of distance 20, is next to be removed and finalized.
6. Vertex 4 is the last vertex to be finalized.

This algorithm correctly finds the shortest path from vertex 0 to vertex 4. However, it doesn’t work in general. Find a graph that shows this.

**25.3 The World-Wide Web is a graph** (8 steps)

**25.3.1 Vertices and edges of the WWW**

**Vertices and edges of the World-Wide Web**

The World-Wide Web can be viewed as a directed graph in which the vertices are static pages, and two vertices are connected by an edge if one page contains a hyperlink to the other. An example appears below.

This is a very large graph. As of 2001, there were estimated to be over four billion Web pages, growing at a rate of 7.5 million documents per day.

**25.3.2 Web crawlers and search engines**

A Web crawler is a program that traverses the graph, looking for “interesting” pages to index. A search engine takes a query and presents pages from the index, with the most “interesting” first. The index is a massive table; the keys are the content words, and the values are the associated URLs. The crawler adds pages to the table, and the search engine retrieves them. The crawler must also maintain a second table to know what pages it has visited already.
The definition of “interesting”

A Web crawler isn’t able to search the whole Web; it’s just too big. A user often finds that listings are three or four months out of date. The best indexes cover only around 25% of the accessible pages. Thus there is an incentive to maximize the “bang for the buck” by visiting only those pages that are likely to yield interesting elements of the index. The search engine faces a similar problem when answering a query. There may be an enormous number of indexed pages that are possibly relevant to the query, and the search engine has to order them so that the most interesting pages get presented to the user first. “Interesting” was traditionally defined by content. For example, a page might be evaluated according to how many “interesting” words it contained. On the search engine side, a page might be evaluated according to the position of the query words in the page (near the beginning is better), on the number of times previous users have accessed that page, or on the “freshness” or recency of the page. Soon, however, users seeking to maximize exposure to their pages uncovered ways to trick the content evaluation. For example, they would include many invisible copies of a popular keyword in their pages. The big innovation of Google’s inventors Larry Page and Sergey Brin (former Stanford grad students) was to include properties of the graph structure in the test for interestingness. Initially they gave priority to pages with higher in-degree, on the assumption that the more people that link to your page, the better your page must be. They later refined that idea: the more interesting pages link to you, the more interesting you must be. Web crawlers can use the same criterion to tell whether to continue their search through a given page.

Improving crawler/search engine performance

Improving the performance of Web crawlers and search engines is a thriving area of research. It relates to a wide variety of computer science research:

- natural language understanding;
- parallel computing algorithms;
- user interface design and evaluation;
- graph theory; and
- software engineering.

25.4.4 Learn about Java’s garbage collection mechanism. (9 steps)

25.4.4.1 Garbage collection

Suppose you create an object and then essentially throw it away, e.g.

```java
TreeNode t = new TreeNode("xyz");
...  
t = null;  // the "xyz" tree node is no longer accessible
```

You may have done this by accident in work earlier this semester. Sometimes, you may do it intentionally if you know you won’t need the object anymore. One might wonder what happens to all these inaccessible objects; do they just hang around, continuing to clutter the program’s memory, and if so, doesn’t that cause any problems? The answers are, yes, letting orphaned objects hang around can cause problems (these “memory leaks” can cause a program to run out of memory and crash), but there are remedies. One technique is called garbage collection. When available memory for building objects runs out, the system steps in to classify all the program’s memory as “in use” or “not in use.” If it then collects all the “not in use” memory—the “garbage”—in one place, thereby making it available for further uses of the JVM. Language environments that use garbage collection include Java and Scheme. With languages such as C and C++, the user must manage his or her own storage; doing this incorrectly is the cause of most of the errors in C and C++ programs.

25.4.4.2 The garbage collection algorithm

Garbage collection requires that each object have a “mark” bit. It proceeds in three steps.

1. First, all the marked bits are turned off. This involves a simple sweep through memory; each object has a header that includes its length, so we can just hop from object to object (this includes the orphaned objects).

2. We then mark all the objects that are currently in use. How do we find them? By starting with the references on the system stack (i.e. the local reference variables), going to the objects that those reference, and so on. For example, we worked early in the semester with a `Line` class that consisted of an array of two points. In the `main` method, we may have said

   ```java
   Line myLine = new Line(new Point (1,2), new Point (2,2));
   ```

   The marking process would start by marking the `Line` object, then the array object that’s an instance variable in the `Line` class, then the two `Point` references in that array, and finally the `Point` objects themselves.

3. Finally, the marked memory is compacted, leaving the unused memory all together and ready to be reused. (The bookkeeping necessary to keep references organized as they are moved elsewhere in memory is a bit tricky; you’ll study this in CS 61C.)

The marking phase is essentially a graph traversal. Each active object is a vertex in the graph; two objects are adjacent if one contains a reference to the other.

25.4.5 A problem with the marking phase

A problem with the marking phase

All the graph traversals we’ve done so far have involved an auxiliary structure like a stack or queue to hold the “fringe.” When we’re doing garbage collection, however, we have run out of memory;
setting up a new stack or queue object isn't possible. This seems like a serious problem.

### 25.4.4 (Display page) A solution

#### A solution

A solution to the inability to set up a separate structure to keep track of the traversal is to use the pointer fields in the objects being traversed to store the information that would have gone into the stack. We of course have to restore this information when we're done. Here's how this technique would work with a linked list. Our task is to go down to the end of the list, then to come back up, revisiting the nodes in reverse order. It's easy enough to go down the list with just an auxiliary pointer:

```
myHead
    1
    2
    3
    4
```

This doesn't let us go backward, though. An idea: reverse pointers as we go.

```
myHead
    1
    2
    3
    4
```

Here's how this technique would work with a linked list.

Our task is to go down to the end of the list, then to come back up, revisiting the nodes in reverse order.

It's easy enough to go down the list with just an auxiliary pointer:

```
myHead
    1
    2
    3
    4
```

This doesn't let us go backward, though. An idea: reverse pointers as we go.

```
myHead
    1
    2
    3
    4
```

25.4.5 (Brainstorm) Steps in the traversal loop

Supply the assignment statements that complete the next step of the backward traversal, that is, that get us from

```
myHead
    1
    2
    3
    4
```

to

```
myHead
    1
    2
    3
    4
```

in each node is named `myNext`.

### 25.4.6 (Display page) Tree traversal without a stack

#### Tree traversal without a stack

We now present an algorithm due to Deutsch, Schorr, and Waite that traverses a binary tree without a stack, using only a constant amount of space plus a "tag" bit and a back-pointer bit in each node. Two pointers are maintained, one named `current` and the other `prev`. Going down the tree, `current` leads the way; `prev` points to the parent. Going back up the tree, `prev` leads the way. The `myLeft` and `myRight` pointers keep track of where we've been, and the back-pointer bit indicates which field in the node contains the back pointer. Here are the possibilities.

```
contact unknown

```

Following `current`'s left pointer

```
parent pointer

tagged node
```

Following `current`'s right pointer (since the left is blocked)

```
parent pointer
```

Switching from a left branch to a
25.4.7 (Brainstorm) Reaction?
What do you think of this algorithm?

25.4.8 (Display page) Generalization

Generalization

We've now seen this pointer-reversal idea used for singly linked lists and binary trees (and graphs with out-degree 2). It can be generalized to more than two outgoing pointers per object; each outgoing pointer is reversed to temporarily hold an incoming pointer. Thus the garbage collection system is using the programmer's own data structures. Garbage collection must leave them in the same form as they started; the "going back up" process does this.

25.4.9 (Display page) Modern garbage collection techniques

Modern garbage collection techniques

In practice, most programs contain a small number of objects that live for a long time, and a large number of objects that come and go quickly. A technique called generational garbage collection handles this by segregating objects that have been around a long time into their own section of memory. These sections of older "generations" get traversed less frequently. The various steps of garbage collection all basically take over the processing while they do their work. Some real-time applications, however, can't afford to be interrupted by a process as complicated as garbage collection. Modern operating systems allow concurrent garbage collection, where the garbage collector is allowed to run for a short while, then the application program (even though the garbage collection process hasn't finished!), switching back and forth between the two.

25.5 Homework

25.5.1 (Display page) Project 3 progress

In lab on Tuesday and Wednesday, you will be expected to show some progress on your project. Some possibilities: initializing a tray from a file; finding a move to make; or making a move. Your t.a. will also want you to predict what you will have completed by the next checkpoint (Tuesday/Wednesday), and how you will split up the predicted work between you and your partner. This prediction will be used to evaluate your progress for that checkpoint.

26 Activities for April 22 and 23

26.2 Show progress on your project

Project checkoff

You may earn up to 6 homework points for today's project checkoff. 1 point is for your prediction of how far along on the project you will be at this time next week, and for your explanation of how you are splitting work between you and your partner. The other 5 points will be awarded according to the following table:

<table>
<thead>
<tr>
<th>Points</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Completion of two of the tasks listed below, with evidence that they work correctly.</td>
</tr>
<tr>
<td>4</td>
<td>Completion of one of the tasks listed below, with evidence that it works correctly, plus significant progress on one of the other tasks.</td>
</tr>
<tr>
<td>3</td>
<td>Completion of one of the tasks listed below, with evidence that it works correctly.</td>
</tr>
<tr>
<td>2</td>
<td>No working code, but with one of the tasks below almost implemented.</td>
</tr>
<tr>
<td>1</td>
<td>Some progress on one of the tasks below.</td>
</tr>
<tr>
<td>0</td>
<td>No apparent progress on the project.</td>
</tr>
</tbody>
</table>

The tasks:
- Input of blocks from a file into a tray data structure
• comparison of a tray with a goal configuration
• generating moves from a given configuration
• making a move
• implementation of a good hashing scheme

26.2.2 (Display page) Project work

Project work

Use the rest of the lab period to work on your project. Reminder: the best approach to this project is
to get something straightforward running, and only then to start optimizing.

26.3 Readings (1 step)

26.3.1 (Display page) Upcoming topics

Upcoming topics

The topic for the next lab is sorting algorithms, covered in Chapter 10 in Objects, Abstractions, Data
Structures, and Design.

27 Sorting algorithms 2008-4-24 ~ 2008-4-25 (3 activities)

27.2 Explore several sorting algorithms (23 steps)

27.2.1 (Brainstorm) Sorting by hand (1)

Explain how you would sort a hand of 13 playing cards as you are dealt the cards one by one. Your
hand should end up sorted by suit, and then by rank within a suit.

27.2.2 (Brainstorm) Sorting by hand (2)

Explain how you would sort a pile of 300 CS 61A exams by login name. If your algorithm differs from
your card-sorting algorithm of the previous step, explain why.

27.2.3 (Display page) Overview of sorting algorithms

Overview of sorting algorithms

Today we’ll explore several algorithms for sorting a collection of data. The methods we’ll be working
with will sort an array or linked list of integers, but can easily be adapted to work with a collection of
Comparable objects. There are several categories of sorting algorithms:

• selection sorts, which repeatedly remove first the largest element, then the second largest,
  and so on from a collection;
• insertion sorts, which repeatedly insert elements into a sorted collection (your card sorting
  algorithm was probably one of these);
• exchange sorts, which repeatedly exchange elements that are out of order until the collection
  is sorted;
• merge sorting, in which sorted sequences of items are merged into larger sequences;
• distribution sorting, where elements are sorted into groups based on their “digits”; and the
  groups are sorted and combined.

Activities for today will include coding and analysis. The simple algorithms generally perform better
than their more complicated counterparts on small sets of data, and we’ll be working with timing data
to find out the crossover point at which the more complicated algorithms perform better.

27.2.4 (Display page) Insertion sort

Insertion sort

Insertion sort is basically an accumulation (recall accumulations from CS 3 and 61A?) that’s coded in
Scheme as follows:

(define (sorted L)
  (accumulate
    (lambda (x sortedSoFar) (insert x sortedSoFar))
    L) )

(You were briefly introduced to this algorithm back when you were learning about arrays and loop
invariants.) Here is code that applies the insertion sort algorithm to an array named values.

for (int k=1; k<values.length; k++) {
  // Elements 0 ... k-1 are in order.
  int temp = values[k];
  int j;
  // Find the place to put the next element
  // while shifting elements down.
  for (j=k-1; j>=0; j--) {
    if (values[j]>temp) {  // comparison step
      break;
    } else {
      values[j+1] = values[j];
      }
  }
  // Put the new element in its proper place.
  // Elements 0 ... k are now in order.
  values[j+1] = temp;
}

27.2.5 (Brainstorm) Insertion sort analysis

Here’s the insertion sort code.

for (int k=1; k<values.length; k++) {
  int temp = values[k];
  int j;
  for (j=k-1; j>=0; j--) {
    if (values[j]>temp) {  // comparison step
      break;
    } else {
      values[j+1] = values[j];
      }
  }
  values[j+1] = temp;
}

Describe how to arrange the values in the array so that, when the above algorithm is applied, the
number of comparisons done at the “// comparison” step is minimized. Then describe a worst-case
arrangement of array values that maximize the number of executions of the “// comparison” step.
Briefly explain your answers. Correct your answer if necessary after reviewing the responses of your
27.2.6 (Display page) Insertion sort applied to linked lists

Insertion sort applied to a linked list

The file \$cs61b/code/IntList.java contains an implementation of a doubly-linked list, along with methods for sorting the list values. Supply the body of the `insert` method to complete the coding of the insertion sort algorithm.

27.2.7 (Display page) Selection sort

Selection sort

The selection sort algorithm, applied to a collection of \(N\) elements, involves the following loop:

\[
\text{for (int } k=0; k<\text{values.length}; k++) { }
\]

1. Find and remove the largest element in the collection, and add it to the end of another sorted collection.

Given below are implementations of the selection sort algorithm for arrays and linked lists. Array sorting

\[
\text{for (int } k=0; k<\text{values.length}; k++) { }
\]

1. // Elements 0 ... k-1 are in their proper positions
2. // in sorted order.
3. int maxSoFar = value[k];
4. int maxIndex = k;
5. // Find the largest element among elements k ... N-1.
6. for (int j=k+1; j<\text{values.length}; j++) {
7. if (\text{values[j]} > maxSoFar) {
8. maxSoFar = \text{values[j]};
9. maxIndex = j;
10. }
11. }
12. // Put the element in its proper place.
13. // Elements 0 ... k are now in their proper positions.
14. swap(value, maxIndex, k);
15. }

Linked list sorting

\[
\text{intList sorted = new IntList (); }
\]

1. while (myHead != null) {
2. int maxSoFar = myHead.myItem;
3. DListNode maxPtr = myHead;
4. // Find the node in the list pointed to by myHead
5. // whose value is largest.
6. for (DListNode p=myHead; p!=null; p=p.myNext) {
7. if (p.myItem > maxSoFar) {
8. maxSoFar = p.myItem;
9. maxPtr = p;
10. }
11. }
12. sorted.addToEnd (maxSoFar);
13. remove (maxPtr);
14. }
15. myHead = sorted.myHead;

One may observe that, in the first iteration of the loop, element 0 is compared with all the others. Then element 1 is compared with \(N-2\) other elements, element 2 is compared with \(N-3\) other elements, and so on. The total number of comparisons is \(N-1 + N-2 + \ldots + 1 = \frac{N(N-1)}{2}\), no matter what the ordering of elements in the array or linked list prior to sorting.

27.2.8 (Display page) Choosing a better sorting structure

Choosing a better sorting structure

In pseudocode, the selection sort algorithm is

\[
\text{for (int } k=0; k<\text{values.length}; k++) { }
\]

1. find and remove the largest element in the collection, and add it to the end of another sorted collection.

Adding something to the end of a sorted array or linked list can be done in constant time. Finding and removing the maximum element in a collection can also be done quickly if we organize the collection appropriately first. A max heap fills the bill; removal of the largest element from a heap of \(N\) elements can be done in time proportional to \(\log N\), so once the heap is created, sorting can be done in time proportional to \(N \log N\). (A removal from the heap are done, each taking time at worst proportional to \(\log N\).) An unexpected bonus is that we can organize \(N\) array elements into a heap in linear time. Here’s an example of how to do this with an array containing 3, 1, 6, 4, 9, 2, 8. These values represent the tree

\[
\text{(1, 2, 3, 4, 5, 6, 7)}
\]

1. The leaves of the heap are 4, 9, 2, 8. Viewed in isolation, they are already heaps (each containing one element).
2. Now we process the internal heap nodes, starting at 6. Its children are 2 and 8, so it should trade places with the 8 to make the values 6, 2, 8 into a heap.
3. We move now to 1. Its children are 4 and 9; we exchange 1 with 9 to build a heap of those three elements.
4. Finally, we bubble 3 down.

Why is this faster? The number of levels each element moves down is at most its height. The sum of the heights of nodes in a binary tree of \(N\) nodes turns out to be proportional to \(N\).

27.2.9 (Self Test) Building a heap in linear time

What array results from applying the linear-time heap creation algorithm to an array containing 5, 2, 3, 4, 6, 7?
Choosing another good sorting structure

Insertion sort, in pseudocode, is

for each element in the collection to be sorted,
insert it into its proper place in another collection.

The insertion sort algorithm we just considered did its insertion in an array, where elements had to be shifted over to make room for the new element. Choice of a structure that allows faster insertion—namely, a binary search tree—produces a faster algorithm. We build the tree through repeated insertions, then traverse it in linear time to produce the sorted sequence.

"Divide and conquer"

Another way to speed up sorting is by using a "divide and conquer" technique:

1. Split the elements to be sorted into two collections.
2. Sort each collection recursively.
3. Combine the sorted collections.

Compared to selection sort, which involves comparing every element with every other, this dividing and conquering has the potential to reduce the number of comparisons significantly. Two algorithms that apply this approach are merge sort and Quicksort.

Merge sort

Merge sort works as follows.

1. Split the collection to be sorted in half somehow.
2. Sort each half.
3. Merge the sorted half-lists.

Merging two sorted lists takes time proportional to the sum of the lengths of the two lists in the worst case. Splitting the collection in half requires a single pass through the elements. The processing pattern is depicted in the diagram below.

```
<table>
<thead>
<tr>
<th>split in half, requiring N/2 operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>split in half, requiring N/2 operations</td>
</tr>
<tr>
<td>N/2 operations</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>merge pairwise</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>merge</td>
</tr>
</tbody>
</table>
```

The diagram is a collection of processes that all together run in linear time. Since there are \( \log N \) levels, the total time is proportional to \( N \log N \).

Quicksort

Another application of dividing and conquering appears in the Quicksort algorithm:

1. Split the collection to be sorted into two collections by partitioning around a "divider" element. One collection consists of elements greater than the divider, the other of elements less or equal to the divider.
2. Quicksort the two subcollections.
3. Concatenate the sorted large values with the divider, and then with the sorted small values.

Here's an example of how this might work, sorting an array containing 3, 1, 4, 5, 9, 2, 8, 6.

1. Choose 3 as the divider. (We'll explore how to choose the divider shortly.)
2. Put 4, 5, 9, 8, and 6 into the "large" collection and 1 and 2 into the "small" collection.
3. Sort the large collection into 9, 8, 6, 5, 4; sort the small collection into 2, 1; combine the two collections with the divider to get 9, 8, 6, 5, 4, 3, 2, 1.

Concatenation in an array or linked list can be done in constant time. If partitioning can be done in time proportional to the number of elements \( N \), and it splits the collection more or less in half, this produces an algorithm that runs in time proportional to \( N \log N \). (Reasoning is similar to what we used for merge sort.)

Picking the divider

The median element would be the best divider. What would be the impact on Quicksort's running time if we partitioned the elements to be sorted by first finding the median, then using that value as the divider?
27.2.15 (Brainstorm) Worst case behavior?
Suppose that the first element is selected as the divider. Describe an arrangement of values in an array or linked list that would produce the worst possible behavior of the Quicksort algorithm.

27.2.16 (Display page) Practical methods for selecting the divider

Practical methods for selecting the divider

Selection of the divider needs to be done quickly. Choosing the first element in the array or list is quick; a somewhat better choice when sorting an array is the median of the first, last, and middle elements. This avoids the worst case you came up with in the previous step.

27.2.17 (Display page) Quicksort a linked list

Some of the code is missing from the quicksort method in -cs61b/code/IntList.java. Supply it to complete the Quicksort implementation.

27.2.18 (Display page) Quicksort performance in practice

Quicksort performance in practice

Quicksort in practice turns out to be the best general-purpose sorting method. For example, it's the algorithm used in Java's Arrays.sort method. With some tuning, the most likely worst-case scenarios are avoided, and the average case performance is excellent. Here are some improvements to the basic Quicksort algorithm.

- When the number of items to sort gets small (the base case of the recursion), insertion sort is used instead.
- For large arrays, more effort is expended on finding a good divider element.
- Various machine-dependent methods are used to optimize the partitioning algorithm and the swap operation.

27.2.19 (Display page) Identifying mystery sorts

Identifying mystery sorts

This activity involves the use of a program called "Sort Detective" to identify "mystery" sort methods from their execution behavior. (Credit for Sort Detective goes to David Levine, Computer Science Department, St. Bonaventure University, New York.) We suggest that you do this with a partner. Start by copying the Sort Detective program to your own directory:

```
cp -r ~cs61b/code/sort.detective ~
```

Then cd to that directory and run Sort Detective:

```
cd ~/sort.detective
java SortDetective
```

The five buttons on the left correspond to the insertion sort, selection sort, merge sort, Quicksort, and heap sort algorithms. (The tested implementations are given in the file sort.detective/allSorts.txt; printed copies are available in lab.) Your task is to find out which button goes with which sort. Create a list to be sorted by selecting relevant radio buttons and then clicking on "Create list." Then run one or more of the sorting methods on that list. Identify the sorting methods by observing their execution behavior on large and small lists, and already-ordered and randomly ordered lists.

27.2.20 (Brainstorm) Sorting out the sorts

Which button corresponds to which sort algorithm? Explain how you determined this.

27.2.21 (Display page) Timing the sort methods

Timing the sort methods

To finish off, we'll do some timing experiments. The program ~cs61b/code/sorting/SortPerf.class applies one of a collection of sorting algorithms to randomly generated arrays whose size is provided as a command-line argument, and outputs timing data to a file. An example run of SortPerf would be

```
java SortPerf select 5 175 200 select.dat
```

The first argument to SortPerf is the sorting method ("select", "merge", "heap", "insert" or "quick"), the second is the smallest array to sort and the increment between array sizes, the third is the maximum array size, and the last is the number of times to run each size. Accurate results require that each test be run numerous times, since elapsed time for individual runs will vary depending on what else is happening on the computer. The reported times are total milliseconds for all runs of a given size. The java command just given applies selection sort to arrays of size 5, 10, 15, ... , 175 and runs each one 200 times, placing the timing results in the file named select.dat. (The timings often look a little "bumpy," that is, the running time does not appear to be strictly increasing with input size—. Running SortPerf again sometimes helps.) Copy the directory ~cs61b/code/sorting to your own directory, then cd to that directory. We have provided a shell script named runrace that you may use to run SortPerf for the same sizes on merge sort, insertion sort, Quicksort, heap sort, and selection sort; it creates data files named insert.dat, select.dat, merge.dat, heap.dat, and quick.dat. Use a program called gnuplot to plot the timing results. At the prompt in a shell window (not emacs or Eclipse), type

```
gnuplot
```

When gnuplot has been loaded, you will see a --> prompt. You can then plot the data produced by runrace using the command:

```
plot 'select.dat' with linesp 1, 'merge.dat' with linesp 2, 'insert.dat' with linesp 3, 'quick.dat' with linesp 4, 'heap.dat' with linesp 5
```

Type the two lines above as one command, all on one line. You should observe that although the five algorithms asymptotically behave as expected, it is unclear which algorithm is faster for small array sizes. At what values of N do you observe crossovers in execution time between the five sorting algorithms? To help see these crossovers better, use the runrace2 shell script, then replots.

27.2.22 (Brainstorm) Timing results

Determine (roughly) where the crossover point is between the N^2 algorithms and the N log N
27.2.2 Animations

Animations

There are a lot of web sites that provide animations of sorting algorithms. Here are some good ones:

  These are patterned on the "race" animations in the "Sorting Out Sorting" video.
  These animations of Quicksort and heap sort have lots of annotation.
- http://maven.smith.edu/~thiebaut/java/sort/demo.html
  This site has every algorithm we've considered today but merge sort.
  This animation of merge sort has lots of annotation.

27.3 Homework (1 step)

27.3.1 Preparation for next week

Preparation for next week

T.a.s will expect to see progress on your project in lab next Tuesday and Wednesday. Evidence of progress should include the following:

- a list of what you and your partner are doing;
- a description of what you have working so far;
- a list of the things you intend to complete by the May 6-7 checkoff;
- what, if any, bottlenecks you've encountered in your work so far.

This information would also be good to have in your README file. Among the things the t.a.s would like to see are several of the following:

- input of blocks from a file into a tray data structure;
- comparison of a tray with a goal configuration;
- generating moves from a given configuration;
- making a move;
- implementation of a good hashing scheme;
- a comprehensive isOK method for trays.

28 Activities for April 29 and 30 2008-4-29 ~ 2008-4-30 (2 activities)

28.2 Work on project 3 (2 steps)

28.2.1 Project checkoff

Project 3 checkoff

Your t.a. will award up to 5 points for your progress on project 3. You earn these points as follows:

<table>
<thead>
<tr>
<th>Score</th>
<th>Task Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Meeting your prediction from last week, or being able to solve a puzzle.</td>
</tr>
<tr>
<td>4</td>
<td>Completion of three of the tasks listed below, with evidence that they work correctly.</td>
</tr>
<tr>
<td>3</td>
<td>Completion of two of the tasks listed below, with evidence that they work correctly, plus significant progress on one of the other tasks.</td>
</tr>
<tr>
<td>2</td>
<td>Completion of two of the tasks listed below, with evidence that they work correctly.</td>
</tr>
<tr>
<td>1</td>
<td>Completion of one of the tasks listed below, with evidence that it works correctly.</td>
</tr>
</tbody>
</table>

The tasks:

- input of blocks from a file into a tray data structure
- comparison of a tray with a goal configuration
- generating moves from a given configuration
- making a move
- implementation of a good hashing scheme
- a comprehensive isOK method for trays

28.2.2 Project work

Project work

Use the rest of the lab period to work on your project. Reminder: the best approach to this project is to get something straightforward running, and only then to start optimizing. Your first target should be the solution of the easy puzzles. You still have almost two weeks to solve the hard puzzles.

29 Balanced search trees, and sorting odds and ends 2008-5-01 ~ 2008-5-02 (4 activities)

29.2 Explore techniques for balancing a search tree (15 steps)

29.2.1 Balanced search trees: an overview

Balanced search trees: an overview

Over the past several weeks, we have analyzed the performance of algorithms for access and insertion into binary search trees under the assumption that the trees were balanced. Informally, that means that the paths from root to leaves are all roughly the same length, and that we won't have to worry about lopsided trees in which search is linear rather than logarithmic. This balancing doesn't happen automatically, and we have seen how to insert items into a binary search tree to produce worst-case search behavior. There are two approaches to tree balancing: incremental balance, where at each insertion or deletion we do a bit of work to keep the tree balanced; and all-at-once balancing, where we don't do anything to keep the tree balanced until it gets too lopsided, then we completely rebalance the tree. In the activities of this segment, we start by analyzing some tree
balancing code. Then we explore how much work is involved in maintaining complete balance. We'll move on to explore two kinds of balanced search trees, AVL trees and 2-3 trees. Finally, we'll investigate tries, which provide a mechanism for improving on \( \log n \) access time. Chapter 11 in Objects, Abstractions, Data Structures, and Design provides more details about balancing schemes for search trees.

### 29.2.2 (Display page) Code for rebalancing a tree

#### Code for rebalancing a tree

Consider the following code for building a balanced tree out of items in a LinkedList.

```java
public BinaryTree (LinkedList list) {
    myRoot = ll2tree (list.iterator ( ), list.size ( ));
}

private TreeNode ll2tree (Iterator iter, int n) {
    if (n == 0) {
        return null;
    }
    if (n == 1) {
        return new TreeNode (iter.next ( ));
    }
    TreeNode child = ll2tree (iter, n/2);
    TreeNode root = new TreeNode (iter.next ( ));
    root.myLeft = child;
    root.myRight = ll2tree (iter, n - n/2 - 1);
    return root;
}
```

The next few steps ask questions about this code. You may wish to test the code (by incorporating it into your BinaryTree class) prior to answering the questions.

#### 29.2.3 (Brainstorm) Comments for ll2tree

Provide a good comment for the `ll2tree` method that specifies preconditions and invariant relations involving the `iter` and `n` arguments, and describes the returned tree in terms of `iter` and `n`.

#### 29.2.4 (Brainstorm) Runtime estimate for ll2tree

Estimate `ll2tree`'s running time, and briefly justify your estimate.

#### 29.2.5 (Brainstorm) Shape of the tree ll2tree builds

Is the tree built by `ll2tree` heap-shaped? Briefly explain why, or give a counterexample.

#### 29.2.6 (Brainstorm) Worst case for maintaining complete balance

Suppose we wished to maintain a heap-shaped binary search tree after each insertion. Insertion of 1 into the tree below, as marked in bold, is a worst case. How many `TreeNode` would change as a result of inserting 1 into the above tree? Briefly explain your answer.

#### 29.2.7 (Display page) "Almost-balanced" trees

"Almost-balanced" trees

The preceding step suggests that maintaining complete balance requires too much effort. A solution is to allow more flexibility via "almost-balanced" trees. In a height-balanced tree, the heights of the two subtrees differ by at most 1 for each non-leaf node in the tree. Here are some examples.

| | | 
|---|---|---|
| Height of left subtree is 2, height of right subtree is 1. Balance requirements also hold for the children. | Height of left subtree is 2, height of right subtree is 3. Balance requirements also hold for the children. |

#### 29.2.8 (Brainstorm) Question about the definition

The definition of height-balanced trees requires that the heights of subtrees differ by at most 1 in every node of the tree. Why is this extra condition necessary? (We might instead require only that the heights of the root's left and right subtrees differ by at most 1.)

#### 29.2.9 (Display page) AVL trees

AVL trees

AVL trees (named after their Russian inventors, Adel'son-Vel'ski and Landis) are height-balanced binary search trees, in which information about tree height is stored in each node along with the `myItem` information. Restructuring of an AVL tree after insertion is done via rotations, an example of which is shown below.

<table>
<thead>
<tr>
<th>before insertion</th>
<th>after insertion</th>
<th>after rotation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="attachment.png" alt="before insertion" /></td>
<td><img src="attachment.png" alt="after insertion" /></td>
<td><img src="attachment.png" alt="after rotation" /></td>
</tr>
</tbody>
</table>

Such a rotation preserves the order relationship among tree elements, but changes the levels of some of the nodes. Most insertions would not require any special handling. There are essentially two cases that do require tree restructuring, however. These are shown below. In each case, an insertion is done that increases the height of the subtree outlined in bold (from \( h \) to \( h+1 \)).
The second case is essentially a left rotation followed by a right rotation. Other cases are mirror images of the ones shown. Figures 11.3 through 11.5 and 11.9 through 11.17 in Objects, Abstractions, Data Structures, and Design also illustrate this process. The resulting tree may itself be too high compared to its sibling, so an insertion potentially propagates to the top of the tree, producing a worst case proportional to $\log N$ (where $N$ is the number of tree elements).

29.2.10 (Self Test) Assignment statements involved in a rotation

Estimate the number of operations required to do a single rotation. Assume that $x$ is the number of items in the tree.

29.2.11 (Brainstorm) Balancing a tree

Consider the following tree. Describe the result of applying a rotation to balance the tree.

29.2.12 (Display page) 2-3 trees

2-3 trees

Another approach to "almost-balanced" trees is to relax the requirement that a node has only two subtrees. In a 2-3 tree, each nonleaf stores either one key and two subtree pointers, or two keys and three subtree pointers. A one-key, two-subtree node is like its BST counterpart; in a node with two keys, all the values in the left subtree are less than or equal to the first key, all the values in the middle subtree are between the two keys, and all the values in the right subtree are greater than or equal to the second key. Thus search proceeds essentially as in a BST, with an extra comparison possibly required at each two-key node. All paths from the root to a leaf are the same length. This means search is required at worst time proportional to $\log N$, where $N$ is the number of keys in the tree. Insertion into a 2-3 tree occurs at the bottom, just as in a BST. We first consider the simplest case of an insertion for which there is plenty of room, for example, inserting "E" into the tree

```
    G
   / \    / \
 C   E   I   M
```

Following the link, we find that "E" belongs to the right of "C", so we just add a key for the corresponding node:

```
    G
   / \    / \
 C   E   I   M
    / \
   N
```

Now we insert "N". It goes to the right of "M", but there's no more room in that node. We must split the node into two nodes, each with one key ("I" in the first, "M" in the second), and send the middle key back up the tree to be inserted above. Since the "G" node has room, "M" is inserted there.

```
    G
   / \    / \
 C   E   I   N
    / \
   M
```

A third case arises when we insert into a full node and the nodes are full all the way back up the tree. In this case we must split the root node and increase the height of the tree. (Thus 2-3 trees grow "up" at the root rather than "down" at the leaves.) This case arises from inserting "D" into the tree above. It doesn't fit in the node with "C" and "E", so that node is split and "D" sent up a level. It also doesn't fit the node with "G" and "M", so that node is split and "G" sent up a level. The following tree results.

```
    G
   / \    / \
 C   D   I   N
```

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
After an insertion that propagates all the way up the tree, it will normally be a long time before another such general restructuring is necessary. Section 11.4 in Objects, Abstractions, Data Structures, and Design explains operations to maintain a 2-3 tree in greater detail.

29.2.13 (Brainstorm) Growing a 2-3 tree
Suppose the keys 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 are inserted sequentially into an initially empty 2-3 tree. Which insertion causes the second split to take place? Briefly explain your answer.

29.2.14 (Display page) B-trees
A 2-3 tree is a special case of a structure called a B-tree. What varies among B-trees is the number of keys/subtrees per node. B-trees with lots of keys per node are especially useful for organizing storage of disk files, for example, in an operating system or database system. Since retrieval from a hard disk is quite slow compared to computation operations involving main memory, we generally try to process information in large chunks in such a situation, in order to minimize the number of disk accesses. The directory structure of a file system could be organized as a B-tree so that a single disk access would read an entire node of the tree. This would consist of (say) the text of a file or the contents of a directory of files. The java.util.TreeMap and TreeSet classes are implemented as "red-black trees," which are a variation of B-trees. Sections 11.3 and 11.4 in Objects, Abstractions, Data Structures, and Design describe B-trees and red-black trees in more detail.

29.2.15 (Display page) Tries
When we covered hashing, we worked with a hashCode function for words that returned the internal code of the word's first letter. Though this was not a very good hash function, it suggests a method for organizing words in a tree structure. Here's an illustration, involving the storage of the words "a", "abase", "abash", "abate", "abbas", "axe", "axolotl", "fabric", and "facet".

We can continue this process, forming the tree shown below. The internal nodes are labeled to show the string prefixes to which they correspond; the square "character" represents the end of a word.

This structure is called a trie (pronounced "try"). If the nodes are implemented in a way that allows quick access to each child node, the time required to find a word in the structure will be at worst proportional to the number of letters in the word. Depending on how many words we're storing, this can provide a dramatic improvement over structures we've considered so far. Suppose, for example, that we want to store most of the 26^5 words consisting of five lower-case letters. Putting them in an array and using binary search requires over 20 comparisons to find a given word, as compared to 5 using a trie. The big disadvantage of using a trie is that we trade space efficiency for time efficiency.

29.3 Do a few more sorting-related activities. (9 steps)

29.3.1 (Display page) Finding the kth largest element
Finding the kth largest element
Suppose we want to find the kth largest element in an array. We could just sort the array to do this. Finding the kth item ought to be easier, however, since it asks for less information than sorting, and indeed finding the smallest or largest requires just one pass through the collection. You may recall the activity of finding the kth largest item (1-based) in a binary search tree augmented by size information in each node. The desired item was found as follows:

1. If the left subtree has x-1 items, then the root is the x item, so return it.
2. If the left subtree has x or more items, find the kth largest item in the left subtree.
3. Otherwise, let j be the number of items in the left subtree, and find the k-j-1st item in the right subtree.

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
A binary search tree is similar to the recursion tree produced by the Quicksort algorithm. The root of the BST corresponds to the divider element in Quicksort; a lopsided BST, which causes poor search performance, corresponds exactly to a Quicksort in which the divider splits the elements unevenly. We use this similarity to adapt the Quicksort algorithm to the problem of finding the $k$th largest element. Here was the Quicksort algorithm.

1. If the collection to be sorted is empty or contains only one element, it's already sorted, so return.
2. Split the collection in two by partitioning around a "divider" element. One collection consists of elements greater than the divider, the other of elements less or equal to the divider.
3. Quicksort the two subcollections.
4. Concatenate the sorted large values with the divider, and then with the sorted small values.

The adaptation is as follows.

1. If the collection contains only one element, that's the one we're looking for, so return it.
2. Otherwise, partition the collection as in Quicksort, and let $j$ be the number of values greater than the divider.
3. If $k$ (1-based) is less than or equal to $j$, the item we're looking for must be among the values greater than the divider, so return the result of a recursive call.
4. If $k = j$, the $k$th largest item is the divider, so return it.
5. Otherwise, the item we're looking for is the $k-j-1$ largest item among the values less than or equal to the divider, so return the result of a recursive call.

The worst-case running time for this algorithm, like that for Quicksort, occurs when all the dividers produce splits as uneven as possible. On the average, however, it runs in time proportional to $N$, where $N$ is the number of items in the collection being searched. Here's a brief explanation of why the algorithm runs in linear time in a good case. Partitioning is where all the work comes in. With optimal dividing, we partition the array of $n$ elements, then a subarray of approximately $n/2$ elements, then a subarray of $n/4$ elements, and so on down to 1. Suppose that partitioning of $j$ elements requires $Pj$ operations, where $P$ is a constant. Then the total amount of partitioning time is approximately

$$Pn + Pn/2 + Pn/4 + \ldots + P1 = P(n + n/2 + n/4 + \ldots + 1) = P(n + n-1) = 2Pn - P$$

which is proportional to $n$.

29.3.2 (Brainstorm) Selection sort using a LinkedList

Here's an implementation of selection sort using LinkedLists.

```java
to LinkedList selectionSort (LinkedList values) {
    LinkedList sorted = new LinkedList ( );
    while (!values.isEmpty ( )) {
        Iterator iter = values.iterator ( );
        Comparable max = (Comparable) values.getFirst ( );
        while (iter.hasNext ( )) {
            Comparable obj = (Comparable) iter.next ( );
            if (max.compareTo (obj) < 0) {
                max = obj;
            }
        }
        values.remove (max); // removes first occurrence of max
        sorted.add (max);
    }
    return sorted;
}
```

In the version of selection sort we saw last week, we maintained a pointer to the max-so-far. Here, we keep track only of the maximum element; the call to remove must then do a linear search for the item to remove from the list. Does this linear search add more than a linear factor to the running time of selection sort? Briefly explain your answer.

29.3.3 (Display page) Sorting with multiple keys

Sorting with multiple keys

So far, we've been concerned with structuring or sorting using only a single set of keys. Much more common is the situation where a key has multiple components, and we'd like the keys to be sorted using any of those components. An example is the collection of files in a UNIX directory. The command

```
ls
```
lists the files sorted by name. The command

```
ls -s
```
lists the files sorted by size.

```
ls -t
```
lists the files sorted by the time of the last modification.

29.3.4 (Display page) An example

An example

Consider the following example, that represents a flat (nonhierarchical) directory of file entries.

```java
public class Directory {
    private FileEntry [ ] myFiles;
    private class FileEntry {
        public String myName;
        public int mySize;
        public GregorianDate myLastModifDate;
    }
}
```

http://spring08.ucwise.org/builder/builderPortal.php?BUILDER_menu=curriculumSummary
public void listBySize ( ) ...
public void listByLastModifDate ( ) ...
public void add (FileEntry f) ...
public void remove (FileEntry f) ...

One approach to supporting all three "listBy..." methods would be to have one copy of the directory's file
entries—the myFiles array—and to sort the file entries in the array into an appropriate sequence
at each call. The figure below shows an example of this approach.

![Example Figure]

Another way, which trades memory efficiency for time efficiency in the case where the directory doesn't change very often, is to have a separate
array for each list order, as shown below.

![Another Example Figure]

Those of you with experience using data base programs may recognize this technique. Each entry in the data base
typically contains a bunch of fields, and the data base program maintains index arrays that allow the entries to be listed by one field or another.

29.3.5 (Brainstorm) Implementing the "add" method

Briefly explain how to implement the add method that uses the myFilesByName, myFilesBySize, and
myFilesByLastModifDate as just described.

29.3.6 (Display page) Stable sorting

Stable sorting

A sorting method is referred to as stable if it maintains the relative order of entries with equal keys.
Consider, for example, the following set of file entries:

data.txt    10003   Sept 3, 2005
hw2.java     1596   Sept 5, 2005
hw2.class    4100   Sept 5, 2005

Suppose these entries were to be sorted by date. The entries for hw2.java and hw2.class have the
same date; a stable sorting method would keep hw2.java before hw2.class in the resulting
ordering, while an unstable sorting method might switch them.

29.3.7 (Brainstorm) Stability of selection sort?

Here is the code for a version of selection sort that sorts an array of file entries into chronological
order using the date of a file entry as the key for sorting. The earlier method returns true exactly
when its first argument date is earlier than its second.

```java
public static void selectionSort(FileEntry list[]) {
    for (int j=list.length-1; j>0; j--) {
        int latestPos = 0;
        for (int k=1; k<=j; k++) {
            if (earlier (list[latestPos], list[k])) {
                latestPos = k;
            }
        }
        if (j != latestPos) {
            FileEntry temp = list[j];
            list[j] = list[latestPos];
            list[latestPos] = temp;
        }
    }
}
```

Is this sorting method stable? Explain why or why not.

29.3.8 (Brainstorm) Stability of insertion sort?

Here is code for a version of insertion sort that sorts file entries into chronological order using dates
as keys as in the previous example. Again, the earlier method returns true exactly when its first
argument date is earlier than its second.

```java
public static void insertionSort (FileEntry list[]) {
    for (int j=1; j<list.length; j++) {
        FileEntry temp = list[j];
        int k = j;
        while( k > 0 && earlier (temp, list[k-1]) ) {
            list[k] = list[k-1];
            k--;
        }
        list[k] = temp;
    }
}
```

Is this sorting method stable? Explain why or why not.

29.3.9 (Display page) Linear-time sorting with radix sort

Linear-time sorting

All the sorting methods we've seen so far are comparison-based, that is, they use comparisons to
determine whether two elements are out of order. One can prove that any comparison-based sort
needs at least O(N log N) comparisons to sort N elements. However, there are sorting methods that
don't depend on comparisons that allow sorting of N elements in time proportional to N A description
of one such algorithm, radix exchange sort, follows. The radix of a number system is the number of
values a single digit can take one. Binary numbers form a radix-2 system; decimal notation is radix 10. Any radix sort examines elements in passes, one pass for (say) the rightmost digit, one for the next-to-rightmost digit, and so on. Radix exchange sort is used to sort elements of an array. Here’s how it works.

1. You look at the array elements one digit at a time, starting with the high-order (most significant, leftmost) digit. You go through the array adding elements to “buckets”, one bucket for each possible digit value (i.e. bucket[0], bucket[1], ..., bucket[9]). Now you’ve divided the array into ten pieces, bucket[0] to bucket[9], although the order of numbers within each piece is unknown.
2. Copy the values back to the array, keeping track of the boundaries between the values in the various buckets. Now you recursively sort each of the ten buckets, only you do the division based on the next highest digit.
3. When you’re down to the low-order digit, you’re finished.

The number of passes through the array is the number of digits in the largest value. If this is known already, it becomes a constant, and the time for the whole sorting process is proportional to that constant times the number of array elements. This algorithm, along with several others, is described in the Wikipedia entry for radix sort. Another radix sort that processes digits in the opposite order is described in Chapter 10, exercise 7 of Objects, Abstractions, Data Structures, and Objects. A nice demo of this algorithm is available online. Why not use radix sort all the time? Fast performance requires a large number of keys that aren’t too long, for which it is relatively straightforward to identify “digits”.

29.4 Homework (1 step)

29.4.1 (Display page) Project checkoffs

Project checkoffs

In the next lab, you will be asked to defend your hash function for trays. This involves answering three questions:

- Do equal trays have the same hash code?
- Is your hash function computable quickly?
- Does it mix up the trays sufficiently?

The InstrumentedHashMap class that you used in lab on April 1 and 2 may be helpful in generating evidence that your hash function is minimizing collisions. You will also be asked to demo your program on the easy puzzles.

30 Activities for May 6 and 7 (3 activities)

30.3 Consult with your t.a. about your project (1 step)

30.3.1 (Display page) Activities

There will be two project-related activities today. One will be to defend your hashing scheme for trays. Up to four homework points will be awarded for your defense:

- 1 for showing that equal trays have the same hash code and that your hash function can be computed quickly;
- 3 for providing evidence that your hash function scatters trays throughout the table.

You will also be awarded up to four homework points for solving easy puzzles (using the run_easy script in ~cs61b/code/proj3/easy). For solving 40 or more, you get 4 points. For solving 30 or more, you get 3. For solving 20 or more, you get 2. For solving 10 or more, you get 1 point. Solving fewer than 10 gets you 0. Reminder: the project is due at 11:59pm on Monday, May 12.