CS 61B: Lecture 1
Wednesday, January 22, 2014
Prof. Jonathan Shewchuk, jrs@cory.eecs
Email to prof & all TAs at once (preferred): cs61b@cory.eecs

Today's reading: Sierra & Bates, pp. 1-9, 18-19, 84.
Handout: Course Overview (also available from CS 61B Web page)

Also, read the CS 61B Web page as soon as possible!
>>> http://www.cs.berkeley.edu/~jrs/61b <<<
YOU are responsible for keeping up with readings & assignments. Few reminders.
The Piazza board is required reading: piazza.com/berkeley/spring2014/cs61b

Labs

Labs (in 271, 273, 275, 330 Soda) start Thursday. Discussion sections start Monday. You must attend your scheduled lab (as assigned by Telebears) to
1) get an account (needed for Lab 1 and Homework 1), and
2) login to turn on your ability to turn in homework (takes up to 24 hours).
You may only attend the lab in which you are officially enrolled. If you are not enrolled in a lab (on the waiting list or in concurrent enrollment), you must attend a lab that has space. (Show up and ask the TA if there's room for you.)

You will not be enrolled in the course until you are enrolled in a lab. If you're on the waiting list and the lab you want is full, you can change to one that isn't, or you can stay on the waitlist and hope somebody drops.

If you're not yet enrolled in a lab, just keep going to them until you find one that has room for you (that week). Once you get enrolled in a lab, though, please always attend the one you're enrolled in.

Prerequisites

Ideally, you have taken CS 61A or E 7, or at least you're taking one of them this semester. If not, you might get away with it, but if you have not mastered recursion, expect to have a very hard time in this class. If you've taken a data structures course before, you might be able to skip CS 61B. See the Course Overview and Brian Harvey (781 Soda) for details.

Textbooks

Kathy Sierra and Bert Bates, Head First Java, Second Edition, O'Reilly, 2005.
ISBN # 0-596-00920-8. (The first edition is just as good.)
Michael T. Goodrich and Roberto Tamassia, Data Structures and Algorithms in Java, Fifth Edition, John Wiley & Sons, 2010. ISBN # 0-470-38326-7.
(The first/third/fourth/sixth edition is just as good, but not the second.)

We will use Sierra/Bates for the first month. Lay your hands on a copy as soon as possible.

Buy the CS 61B class reader at Vick Copy, 1879 Euclid. The bulk of the reader is old CS 61B exams, which will not be provided online. The front of the reader is stuff you'll want to have handy when you're in lab, hacking.

Grading

10 pts	Labs	There are 200 points total you can earn in this course,
20 pts	Homeworks	broken down at left. 185+ points is an A+, 175-184 is
70 pts	Projects	an A, and so on down to D- (85-94). There is NO CURVE.
25 pts	Midterm I	Late homeworks and labs will NOT be accepted, period.
25 pts	Midterm II	Late projects are penalized 1% of your score for every
50 pts	Final Exam	two hours by which you miss the deadline.
200 pts		

There will be three projects, worth 20, 30, and 20 points respectively. You will do the first project individually, and the last two as part of a group of two or three students. You may not work alone on the last two projects.
All homeworks and projects will be turned in electronically.

Cheating

...will be reported to the Office of Student Conduct.
1) "No Code Rule": Never have a copy of someone else's program in your possession and never give your program to someone else.
2) Discussing an assignment without sharing any code is generally okay. Helping someone to interpret a compiler error message is an example of permissible collaboration. However, if you get a significant idea from someone, acknowledge them in your assignment.
3) These rules apply to homeworks and projects. No discussion whatsoever in exams, of course.
4) In group projects, you share code freely within your team, but not between teams.

Goals of CS 61B

1) Learning efficient data structures and algorithms that use them.
2) Designing and writing large programs.
3) Understanding and designing data abstraction and interfaces.
4) Learning Java.

THE LANGUAGE OF OBJECT-ORIENTED PROGRAMMING

Object: An object is a repository of data. For example, if MyList is a ShoppingList object, MyList might record your shopping list.
Class: A class is a type of object. Many objects of the same class might exist; for instance, MyList and YourList may both be ShoppingList objects.
Method: A procedure or function that operates on an object or a class. A method is associated with a particular class. For instance, addItem might be a method that adds an item to any ShoppingList object. Sometimes a method is associated with a family of classes. For instance, addItem might operate on any List, of which a ShoppingList is just one type.
Inheritance: A class may inherit properties from a more general class. For example, the ShoppingList class inherits from the List class the property of storing a sequence of items.
Polymorphism: The ability to have one method call work on several different classes of objects, even if those classes need different implementations of the method call. For example, one line of code might be able to call the "addItem" method on _every_ kind of List, even though adding an item to a ShoppingList is completely different from adding an item to a ShoppingCart.
Object-Oriented: Each object knows its own class and which methods manipulate objects in that class. Each ShoppingList and each ShoppingCart knows which implementation of addItem applies to it.

In this list, the one thing that truly distinguishes object-oriented languages from procedural languages (C, Fortran, Basic, Pascal) is polymorphism.

Java

Java allows you to store data in variables, but first you must _declare_ them and specify their _type_.

```
    Python:  x = 1       Scheme:  (let ((x 1)) )       Java:  int x;
                                                                x = 1;
```

This Java declaration does two things.

(1) It allocates a chunk of memory big enough to store an integer, which Java calls type "int".

(2) It names the variable (chunk of memory) "x".

```
---
|1|
---
 x
```

Variables are used not just to store numbers, but also to _reference_ objects. There are two ways to get classes of objects to play with:

(1) Use one defined by somebody else. Java has tons of pre-defined classes you can use. Many come in the "Java standard library" provided with every Java compiler.

(2) Define your own.

For example, Java has a built-in class called String.

```
String myString;
```

This does _not_ create a String object. Instead, it declares a variable (chunk of memory) that can store a _reference_ to a String object. I draw it as a box.

```
          ---
myString | |                  <-- This box is a variable (not an object).
          ---
```

Initially, myString doesn't reference anything. You can make it reference a String object by writing an assignment statement. But how do we get ahold of an actual String object? You can create one.

```
myString = new String();
```

This line performs two distinct steps. First, the phrase "new String()" is called a _constructor_. It constructs a brand new String object. Second, the assignment "=" causes myString to _reference_ the object. You can think of this as myString pointing to the object.

```
          ---     ------
myString |.+---->|    | a String object
          ---     ------
```

Unlike Scheme and Python, Java programs must be compiled before you can run them. _Compilation_ converts your written code to a machine-readable bytecode. The advantage is a faster program than one written in Scheme. The disadvantage is that you have to wait for it to compile.

```
            Python                      Java
            ------                      ----
      Python program (.py)        Java program (.java)
              |                           |
              | python                    | javac
              v                           v
            Answer                   .class files
                                          |
                                          | java
                                          v
                                        Answer
```

Postscript: Anatomy of a Small Java Program (for discussion section)

Assume the following code is in a file called HelloWorld.java:

```
class HelloWorld {
  public static void main (String[] args) {
    System.out.println("Hello, world");
  }
}
```

The classes are "HelloWorld", "String", and "System".
The objects are "args", "System.out", and the string "Hello, world".
(Actually, the first two of these are _references_ to objects.)
The methods are "main" and "println". The println method prints its parameter, and the main method prints the string "Hello, world".

You don't have the knowledge to completely understand this code yet, but don't worry about it. We'll take the first two lines on faith as the standard way to start a Java program. They'll become clear in the coming weeks.

Let's look at the innermost line first; it does all the action. "out" references an object whose class is PrintStream. A PrintStream is a path by which characters can be output by a program. The characters that are sent through "out" find their way to your computer screen.

System is a class which happens to contain the variable out (among many other variables). We have to write "System.out" to address the output stream, because other classes might have variables called "out" too, with their own meanings.

"println" is a method (procedure) of the class PrintStream. Hence, we can invoke "println" from any PrintStream object, including System.out. "println" takes one parameter, which can be a string.

"main" is a method in the "HelloWorld" class. The HelloWorld class knows how to do the "main" operation, just like the PrintStream class knows how to do the "println" operation.

```
                                                    ------------------------
    ------------------   ----------------------    |                        |
    |                |   |                    |    |                        |
    |        ---     |   |  println (method)  |    |                        |
    |    out |.+-----+-->|                    |    |                        |
    |        ---     |   |                    |    |    main (method)      |
    --System (class)--   --An object of class--    |                        |
                              PrintStream           |                        |
                                                    ---HelloWorld (class)----
```

The classes System and PrintStream are all automatically provided for you by any Java compiler. Somebody has programmed them for you, so that you don't have to figure out for yourself how to send characters to the terminal.

CS 61B: Lecture 2
Friday, January 24, 2014

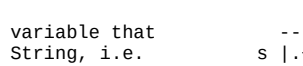
Today's reading: Sierra & Bates, Chapter 2; pp. 54-58, 154-160, 661, 669.

OBJECTS AND CONSTRUCTORS

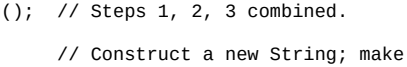
```
String s;           // Step 1:      declare a String variable.
s = new String();   // Steps 2, 3:  construct new empty String; assign it to s.
```

At this point, s is a variable that references an "empty" String, i.e. a String containing zero characters.


```
String s = new String();  // Steps 1, 2, 3 combined.

s = "Yow!";               // Construct a new String; make s a reference to it.
```




```
String s2 = s;            // Copy the reference stored in s into s2.
```



Now s and s2 reference the same object.

```
s2 = new String(s);       // Construct a copy of object; store reference in s2.
```

Now they refer to two different, but identical, objects.

Think about that. When Java executes that line, it does the following things, in the following order.
- Java looks inside the variable s to see where it's pointing.
- Java follows the pointer to the String object.
- Java reads the characters stored in that String object.
- Java creates a new String object that stores a copy of those characters.
- Java stores a reference to the new String object in s2.

We've seen three String constructors:
(1) new String() constructs an _empty_string_--it's a string, but it contains zero characters.
(2) "Yow!" constructs a string containing the characters Yow!.
(3) new String(s) takes a _parameter_ s. Then it makes a copy of the object that s references.

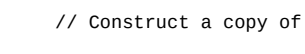
Constructors _always_ have the same name as their class, except the special constructor "stuffinquotes". That's the only exception.

Observe that "new String()" can take no parameters, or one parameter. These are two different constructors--one that is called by "new String()", and one that is called by "new String(s)". (Actually, there are many more than two--check out the online Java API to see all the possibilities.)

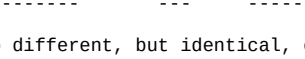
METHODS

Let's look at some methods that aren't constructors.

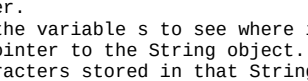
```
s2 = s.toUppercase();       // Create a string like s, but in all upper case.
```




```
String s3 = s2.concat("!!");             // Also written:  s3 = s2 + "!!";
```




```
String s4 = "*".concat(s2).concat("*");  // Also written:  s4 = "*" + s + "*";
```



Now, here's an important fact: when Java executed the line

```
  s2 = s.toUppercase();
```

the String object "Yow!" did _not_ change. Instead, s2 itself changed to reference a new object. Java wrote a new "pointer" into the variable s2, so now s2 points to a different object than it did before.

Unlike in C, in Java Strings are _immutable_--once they've been constructed, their contents never change. If you want to change a String object, you've got to create a brand new String object that reflects the changes you want. This is not true of all objects; most Java objects let you change their contents.

You might find it confusing that methods like "toUppercase" and "concat" return newly created String objects, though they are not constructors. The trick is that those methods calls constructors internally, and return the newly constructed Strings.

I/O Classes and Objects in Java

Here are some objects in the System class for interacting with a user:

```
  System.out is a PrintStream object that outputs to the screen.
  System.in is an InputStream object that reads from the keyboard.
    [Reminder:  this is shorthand for "System.in is a variable that references
                an InputStream object."]
```

But System.in doesn't have methods to read a line directly. There is a method called readLine that does, but it is defined on BufferedReader objects.

- How do we construct a BufferedReader? One way is with an InputStreamReader.
- How do we construct an InputStreamReader? We need an InputStream.
- How do we construct an InputStream? System.in is one.

(You can figure all of this out by looking at the constructors in the online Java libraries API--specifically, in the java.io library.)

Why all this fuss?

InputStream objects (like System.in) read raw data from some source (like the keyboard), but don't format the data.

InputStreamReader objects compose the raw data into characters (which are typically two bytes long in Java).

BufferedReader objects compose the characters into entire lines of text.

Why are these tasks divided among three different classes? So that any one task can be reimplemented (say, for improved speed) without changing the other two.

Here's a complete Java program that reads a line from the keyboard and prints it on the screen.

```
import java.io.*;

class SimpleIO {
  public static void main(String[] arg) throws Exception {
    BufferedReader keybd =
          new BufferedReader(new InputStreamReader(System.in));
    System.out.println(keybd.readLine());
  }
}
```

Don't worry if you don't understand the first three lines; we'll learn the underlying ideas eventually. The first line is present because to use the Java libraries, other than java.lang, you need to "import" them. java.io includes the InputStreamReader and BufferedReader classes.

The second line just gives the program a name, "SimpleIO".

The third line is present because any Java program always begins execution at a method named "main", which is usually defined more or less as above. When you write a Java program, just copy the line of code, and plan to understand it a few weeks from now.

Classes for Web Access

Let's say we want to read a line of text from the White House Web page. (The line will be HTML, which looks ugly. You don't need to understand HTML.)

How to read a line of text? With readLine on BufferedReader.
How to create a BufferedReader? With an InputStreamReader.
How to create a InputStreamReader? With an InputStream.
How to create an InputStream? With a URL.

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

class WHWWW {
  public static void main(String[] arg) throws Exception {
    URL u = new URL("http://www.whitehouse.gov/");
    InputStream ins = u.openStream();
    InputStreamReader isr = new InputStreamReader(ins);
    BufferedReader whiteHouse = new BufferedReader(isr);
    System.out.println(whiteHouse.readLine());
  }
}
```

Postscript: Object-Oriented Terminology (not examinable)

In the words of Turing Award winner Nicklaus Wirth, "Object-oriented programming (OOP) solidly rests on the principles and concepts of traditional procedural programming. OOP has not added a single novel concept ... along with the OOP paradigm came an entirely new terminology with the purpose of mystifying the roots of OOP." Here's a translation guide.

Procedural Programming	Object-Oriented Programming
record / structure	object
record type	class
extending a type	declaring a subclass
procedure	method
procedure call	sending a message to the method [ack! phthhht!]

I won't ever talk about "sending a message" in this class. I think it's a completely misleading metaphor. In computer science, message-passing normally implies asynchrony: that is, the process that sends a message can continue executing while the receiving process receives the message and acts on it. But that's NOT what it means in object-oriented programming: when a Java method "sends a message" to another method, the former method is frozen until the latter methods completes execution, just like with procedure calls in most languages. But you should probably know that this termology exists, much as it sucks, because you'll probably run into it sooner or later.

CS 61B: Lecture 3
Monday, January 27, 2014

Today's reading: Sierra & Bates, pp. 71-74, 76, 85, 240-249, 273-281, 308-309.

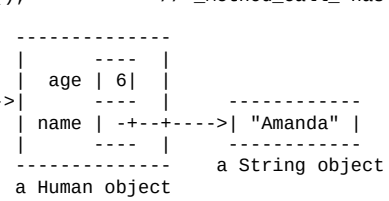
DEFINING CLASSES
==========

An object is a repository of data. _Fields_ are variables that hold the data stored in objects. Fields in objects are also known as _instance_variables_. In Java, fields are addressed much like methods are, but fields never have parameters, and no parentheses appear after them. For example, suppose that amanda is a Human object. Then amanda.introduce() is a method call, and amanda.age is a field. Let's write a _class_definition_ for the Human class.

```
class Human {
  public int age;               // The Human's age (an integer).
  public String name;           // The Human's name.

  public void introduce() {     // This is a _method_definition_.
    System.out.println("I'm " + name + " and I'm " + age + " years old.");
  }
}
```

"age" and "name" are both fields of a Human object. Now that we've defined the Human class, we can construct as many Human objects as we want. Each Human object we create can have different values of age and name. We might create amanda by executing the following code.

```
  Human amanda = new Human();    // Create amanda.
  amanda.age = 6;                // Set amanda's fields.
  amanda.name = "Amanda";
  amanda.introduce();            // _Method_call_ has amanda introduce herself.
```



The output is: I'm Amanda and I'm 6 years old.

Why is it that, inside the definition of introduce(), we don't have to write "amanda.name" and "amanda.age"? When we invoke "amanda.introduce()", Java remembers that we are calling introduce() _on_ the object that "amanda" references. The methods defined inside the Human class remember that we're referring to amanda's name and age. If we had written "rishi.introduce()", the introduce method would print rishi's name and age instead. If we want to mix two or more objects, we can.

```
class Human {
  // Include all the stuff from the previous definition of Human here.

  public void copy(Human original) {
    age = original.age;
    name = original.name;
  }
}
```

Now, "amanda.copy(rishi)" copies rishi's fields to amanda.

Constructors

Let's write a constructor, a method that constructs a Human. The constructor won't actually contain code that does the creating; rather, Java provides a brand new object for us right at the beginning of the constructor, and all you have to write (if you want) in the constructor is code to initialize the new object.

```
class Human {
  // Include all the stuff from the previous definitions here.

  public Human(String givenName) {
    age = 6;
    name = givenName;
  }
}
```

Notice that the constructor is named "Human", and it returns an object of type "Human". This constructor is called whenever we write "new Human(s)", where s is a String reference. Now, we can shorten amanda's coming-out party to

```
  Human amanda = new Human("Amanda");
  amanda.introduce();
```

These lines accomplish precisely the same result as amanda's previous four lines.

You might ask...why were we able to create a Human object before we wrote a constructor? Java provides every class with a default constructor, which takes no parameters and does no initializing. Hence, when we wrote

```
  Human amanda = new Human();
```

we created a new, blank Human. If the default constructor were explicitly written, it would look like this:

```
  public Human() {
  }
```

Warning: if you write your own Human constructor, even if it takes parameters, the default constructor goes away. If you want to have the default constructor _and_ another constructor, you must define both explicitly.

You can override the default constructor by explicitly writing your own constructor with no parameters.

```
class Human {
  // Include all the stuff from the previous definitions here.

  public Human() {
    age = 0;
    name = "Untitled";
  }
}
```

The "this" Keyword

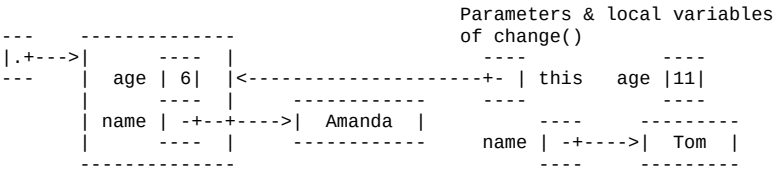
A method invocation, like "amanda.introduce()", implicitly passes an object (in this example, amanda) as a parameter called "this". So we can rewrite our last constructor as follows without changing its meaning.

```
  public Human() {
    this.age = 0;
    this.name = "Untitled";
  }
```

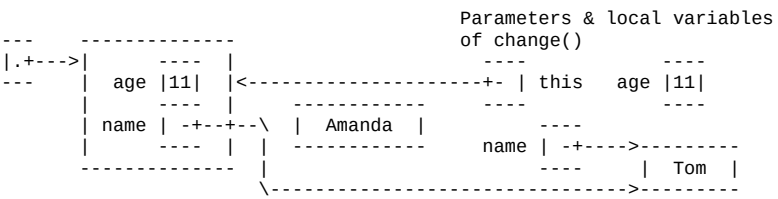
In this case, "this" is optional. However, if the parameters or local variables of a method have the same name as the fields of an object, then the former have priority, and the "this" keyword is needed to refer to the object's fields.

```
  public void change(int age) {
    String name = "Tom";

    this.age = age;
    this.name = name;
  }
```

When we call "amanda.change(11)", "this" is assigned the same value as "amanda" before the change() method begins execution.



Now, when Java executes "this.age = age", it overwrites the 6 with an 11. When Java executes "this.name = name", it overwrites amanda's name as below.




```
  ------------------------------------------------------
  | IMPORTANT:  You CANNOT change the value of "this"! |
  ------------------------------------------------------
```

A statement like "this = amanda;" will trigger a compile-time error.

The "static" Keyword

A _static_field_ is a single variable shared by a whole class of objects; its value does not vary from object to object. For example, if "numberOfHumans" is the number of Human objects that have been constructed, it is not appropriate for each object to have its own copy of this number; every time a new Human is created, we would have to update every Human.

If we declare a field "static", there is just one field for the whole class. Static fields are also called _class_variables_.

```
class Human {
  public static int numberOfHumans;

  public int age;
  public String name;

  public Human() {
    numberOfHumans++;    // The constructor increments the number by one.
  }
}
```

If we want to look at the variable numberOfHumans from another class, we write it in the usual notation, but we prefix it with the class name rather than the name of a specific object.

```
  int kids = Human.numberOfHumans / 4;  // Good.
  int kids = amanda.numberOfHumans / 4; // This works too, but has nothing to
                                        // do with amanda specifically.  Don't
                                        // do this; it's bad (confusing) style.
```

System.in and System.out are other examples of static fields.

Methods can be static too. A _static_method_ doesn't implicitly pass an object as a parameter.

```
class Human {
  ...
  public static void printHumans() {
    System.out.println(numberOfHumans);
  }
}
```

Now, we can call "Human.printHumans()" from another class. We can also call "amanda.printHumans()", and it works, but it's bad style, and amanda will NOT be passed along as "this".

The main() method is always static, because when we run a program, we are not passing an object in.

```
  ------------------------------------------------------
  | IMPORTANT:  In a static method, THERE IS NO "this"! |
  ------------------------------------------------------
```

Any attempt to reference "this" will cause a compile-time error.

Lifetimes of Variables

- A local variable (declared in a method) is gone forever as soon as the method in which it's declared finishes executing. (If it references an object, the object might continue to exist, though.)
- An instance variable (non-static field) lasts as long as the object exists. An object lasts as long as there's a reference to it.
- A class variable (static field) lasts as long as the program runs.

CS 61B: Lecture 4
Wednesday, January 29, 2014

Today's reading: S&B pp. 10-14, 49-53, 75, 78-79, 86, 117, 286-287, 292, 660.

PRIMITIVE TYPES

Not all variables are references to objects. Some variables are primitive types, which store values like "3", "7.2", "h", and "false". They are:

- byte: A 8-bit integer in the range -128...127. (One bit is the sign.)
- short: A 16-bit integer in the range -32768...32767.
- int: A 32-bit integer in the range -2147483648...2147483647.
- long: A 64-bit integer, range -9223372036854775808...9223372036854775807.
- double: A 64-bit floating-point number like 18.355625430920409.
- float: A 32-bit floating-point number; has fewer digits of precision.
- boolean: "true" or "false".
- char: A single character.

long values are written with an L on the end: long x = 43L;
This tells the compiler to internally write out "43" in a 64-bit format.
double and float values must have a decimal point: double y = 18.0;
float values are written with an f at the end: float f = 43.9f;

	Object types	Primitive types
Variable contains a	reference	value
How defined?	class definition	built into Java
How created?	"new"	"6", "3.4", "true"
How initialized?	constructor	default (usually zero)
How used?	methods	operators ("+", "*", etc.)

Operations on int, long, short, and byte types.

```
    -x              x * y
    x + y           x / y    <-- rounds toward zero (drops the remainder).
    x - y           x % y    <-- calculates the remainder of x / y.
```

Except for "%", these operations are also available for doubles and floats.
Floating-point division ("/") doesn't round to an integer, but it does round off after a certain number of bits determined by the storage space.

The java.lang library has more operations in...

```
    - the Math class.
      x = Math.abs(y);  // Absolute value.  Also see Math.sqrt, Math.sin, etc.
    - the Integer class.
      int x = Integer.parseInt("1984");  // Convert a string to a number.
    - the Double class.
      double d = Double.parseDouble("3.14");
```

Converting types: integers can be assigned to variables of longer types.

```
    int i = 43;
    long l = 43;    // Okay, because longs are a superset of ints.
    l = i;          // Okay, because longs are a superset of ints.
    i = l;          // Compiler ERROR.
    i = (int) l;    // Okay.
```

The string "(int)" is called a cast, and it casts the long into an int. In the process, high bits will be lost if l does not fit in the range -2147483648...2147483647. Java won't let you compile "i = l" because it's trying to protect you from accidentally creating a nonsense value and a hard-to-find bug. Java requires you to explicitly cast longs to ints to show your acknowledgment that you may be destroying information.

Similarly, "float f = 5.5f; double d = f;" is fine, but you need an explicit cast for "double d = 5.5; float f = (float) d;". Integers (even longs) can be directly assigned to floating-point variables (even floats) without a cast, but the reverse requires a cast because the number is truncated to an integer.

Boolean Values

A boolean value is either "true" or "false". Booleans have operations of their own, signified "&&" (and), "||" (or), and "!" (not).

a	b	a && b	a || b	!a
false	false	false	false	true
false	true	false	true	
true	false	false	true	false
true	true	true	true	

Boolean values can be specified directly ("true", "false") or be created by the comparison operators "==", "<", ">", "<=", ">=", "!=" (not equal to).

```
    boolean x = 3 == 5;       // x is now false.
    x = 4.5 >= 4.5;           // x is now true.
    x = 4 != 5 - 1;           // x is now false.
    x = false == (3 == 0);    // x is now true.
```

CONDITIONALS

An "if" statement uses a boolean expression to decide whether to execute a set of statements. The form is

```
    if (boolValue) {
      statements;
    }
```

The statements are executed if and only if "boolValue" is "true". The parentheses around the boolean expression are required (for no good reason).

```
    boolean pass = score >= 75;
    if (pass) {
      output("You pass CS 61B");
    } else {
      // The following line executes if and only if score < 75.
      output("You are such an unbelievable loser");
    }
```

if-then-else clauses can be (1) nested and (2) daisy-chained. Nesting allows you to build decision trees. Daisy-chaining allows you to present more than two alternatives. For instance, suppose you want to find the maximum of three numbers.

```
    if (x > y) {
      if (x > z) {
        maximum = x;
      } else {
        maximum = z;
      }
    } else if (y > z) {
      maximum = y;
    } else {
      maximum = z;
    }
```

Some long chains of if-then-else clauses can be simplified by using a "switch" statement. "switch" is appropriate only if every condition tests whether a variable x is equal to some constant.

```
    switch (month) {        |      if (month == 2) {
    case 2:                 |        days = 28;
      days = 28;            |      } else if ((month == 4) || (month == 6) ||
      break;                |                 (month == 9) || (month == 11)) {
    case 4:                 |        days = 30;
    case 6:                 |      } else {
    case 9:                 |        days = 31;
    case 11:                |      }
      days = 30;            |
      break;
    default:
      days = 31;
      break;
    }                   //  These two code fragments do exactly the same thing.
```

IMPORTANT: "break" jumps to the end of the "switch" statement. If you forget a break statement, the flow of execution will continue right through past the next "case" clause, which is why cases 4, 6, and 9 work right. If month == 12 in the following example, both Strings are printed.

```
    switch (month) {
    case 12:
      output("It's December.");
      // Just keep moving right on through.
    case 1:
    case 2:
    case 11:
      output("It's cold.");
    }
```

However, this is considered bad style, because it's hard to read and understand. If there's any chance that other people will need to read or modify your code (which is the norm when you program for a business), don't code it like this. Use break statements in the switch, and use subroutines to reuse code and clarify the control flow.

Observe that the last example doesn't have a "default:" case. If "month" is not 1 nor 2 nor 11 nor 12, Java jumps right to the end of the "switch" statement (just past the closing brace) and continues execution from there.

THE "return" KEYWORD
====================

Like conditionals, "return" affects the flow of control of a program. It causes a method to end immediately, so that control returns to the calling method.

Here's a recursive method that prints the numbers from 1 to x.

```
    public static void oneToX(int x) {
      if (x < 1) {
        return;
      }
      oneToX(x - 1);
      System.out.println(x);
    }
```

The return keyword serves a dual purpose: it is also the means by which a function returns a value. A _function_ is a method that is declared to return a non-void type. For instance, here's a function that returns an int.

```
    public int daysInMonth(int month) {
      switch (month) {
      case 2:
        return 28;
      case 4:
      case 6:
      case 9:
      case 11:
        return 30;
      default:
        return 31;
      }
    }
```

The "return" value can be an expression. Some examples:

```
    return x + y - z;

    return car.velocity(time);
```

CS 61B: Lecture 5
Wednesday, January 29, 2014

Today's reading: Sierra & Bates pp. 59-62, 83, 114-116, 293-300, 670.

LOOPS

"while" Loops

A "while" statement is like an "if" statement, but the body of the statement is executed repeatedly, as long as the condition remains true. The following example tests whether n is a prime number by attempting to divide it by every integer in the range 2...n - 1.

```
    public static boolean isPrime(int n) {
      int divisor = 2;
      while (divisor < n) {          _ <- "divisor < n" is the _loop_condition_.
        if (n % divisor == 0) {       |
          return false;               | These lines inside the braces
        }                             | are called the _loop_body_.
        divisor++;                   _|
      }
      return true;
    }
```

Here's how the loop executes.

- When Java reaches this "while" loop, it tests whether the loop condition "divisor < n" is true.
- If divisor < n, Java executes the loop body {in braces}.
- When Java finishes the loop body (i.e. after executing "divisor++"), it tests _again_ whether "divisor < n" is true.
- If it's still true, Java jumps back up to the beginning of the loop body and executes it again.
- If Java tests the loop condition and finds that "divisor < n" is false, Java continues execution from the next line of code _after_ the loop body.

An _iteration_ is a pass through the loop body. In this example, if n is 2 or less, the loop body won't iterate even once.

"for" Loops

"for" loops are a convenient shorthand that can be used to write some "while" loops in a more compact way. The following "for" loop is equivalent to the following "while" loop.

```
    for (initialize; condition; next) {      |    initialize;
      statements;                            |    while (condition) {
    }                                        |      statements;
                                             |      next;
                                             |    }
```

By convention, the "initialize" and "next" are both expressions that affect a variable that changes every loop iteration and is central to the test. Most commonly, "for" statements are used to iterate while advancing an index variable over a fixed range of values. isPrime can be rewritten thus:

```
    public static boolean isPrime(int n) {
      for (int divisor = 2; divisor < n; divisor++) {    _
        if (n % divisor == 0) {                           |
          return false;                                   | Loop body.
        }                                                _|
      }
      return true;
    }
```

A common mistake among beginning Java and C programmers is to get the condition wrong and do one loop iteration too few. For example, suppose you want to print all the prime numbers in the range 2...n.

```
    public static void printPrimes(int n) {
      int i;
      for (i = 2; i < n; i++) {        // ERROR!!! Condition should be i <= n.
        if (isPrime(i)) {
          System.out.print(" " + i);
        }
      }
    }
```

Suppose we correct this method so the loop condition is "i <= n". Think carefully: what is the value of i when the printPrimes method ends?

We'll come back to iteration, but first let's investigate something more interesting to iterate on.

ARRAYS

An array is an object consisting of a numbered list of variables, each of which is a primitive type or a reference to another object. The variables in an array are always indexed from zero in increments of one. For example, here is an array of characters.

Like any object, an array is only useful if we can reference it, usually through some reference variable like "c" above. We declare c thusly:

```
    char[] c;           // Reference to an array (of any length) of characters.
```

We can construct an array of four characters as follows.

```
    c = new char[4];
```

Now that we have an array object, we may fill in its values by indexing c.

```
    c[0] = 'b';         // Store the character 'b' at index 0.
    c[1] = 'l';
    c[2] = 'u';
    c[3] = 'e';
```

The characters in a four-element array are indexed from 0 to 3. If we try to address any index outside this range, we will trigger a run-time error.

```
    c[4] = 's';         // Program stops with ArrayIndexOutOfBoundsException
```

A _run-time_error_ is an error that doesn't show up when you compile the code, but does show up later when you run the program and the Java Virtual Machine tries to access the out-of-range index.

When c references an array, you can find out its length by looking at the field "c.length". You can never assign a value to the "length" field, though. Java will give you a compile-time error if you try.

Primes Revisited

The printPrimes method is embarrassingly slow when n is large. Arrays can help us write a faster method to identify the primes from 2 to n.

The method uses an ancient algorithm called the Sieve of Eratosthenes. All integers are assumed prime until proven composite. The algorithm iterates through all possible divisors, and marks as non-prime every integer divisible by a given divisor. Here's the beginning of the method.

```
    public static void printPrimes(int n) {
      boolean[] prime = new boolean[n + 1];                 // Numbered 0...n.
      int i;
      for (i = 2; i <= n; i++) {
        prime[i] = true;                        // Prime until proven composite.
      }
```

Why did we construct an array of length n + 1? Because if we'd constructed an array of length n, its elements would be numbered from 0 to n - 1. But we'd like to have an element numbered n.

To continue the method, we iterate over all possible divisors from 2 to the square root of n. For each prime value of divisor, we mark as non-prime all integers divisible by divisor, except divisor itself.

```
      for (int divisor = 2; divisor * divisor <= n; divisor++) {
        if (prime[divisor]) {
          for (i = 2 * divisor; i <= n; i = i + divisor) {
            prime[i] = false;                     // i is divisible by divisor.
          }
        }
      }
```

Math question: why do we only need to consider divisors up to the square root of n?

Finally, we print every integer from 2 to n that hasn't been marked non-prime.

```
      for (i = 2; i <= n; i++) {
        if (prime[i]) {
          System.out.print(" " + i);
        }
      }
    }
```

Observe that elements 0 and 1 of the array are never used. A tiny bit of memory is wasted, but the readability of the code is better for it.

Multi-Dimensional Arrays

A _two-dimensional_array_ is an array of references to arrays. A three-dimensional array is an array of arrays of arrays. As an example, consider Pascal's Triangle.

```
                                    1                 <-- row 0
                                 1     1
                              1     2     1
                           1     3     3     1
                        1     4     6     4     1
                     1     5    10    10     5     1   <-- row 5
```

Each entry is the sum of the two nearest entries in the row immediately above. If the rows are numbered from zero, row i represents the coefficients of the polynomial (x + 1)^i. For example, (x + 1)^4 = x^4 + 4x^3 + 6x^2 + 4x + 1.

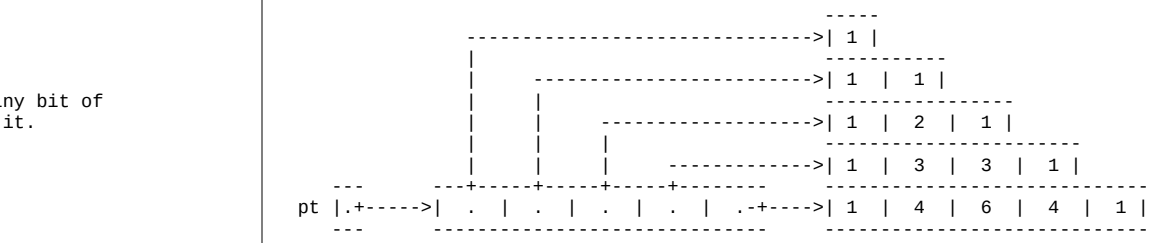
The following method returns an array of arrays of ints that stores the first n rows of Pascal's Triangle.

```
     public static int[][] pascalTriangle(int n) {
       int[][] pt = new int[n][];
```

Here, we've just declared pt to reference an array of arrays, and constructed an array for it to reference. However, the arrays that this array will reference do not yet exist. They are constructed and filled in by the following loop.

```
       for (int i = 0; i < n; i++) {
         pt[i] = new int[i + 1];                            // Construct row i.
         pt[i][0] = 1;                                   // Leftmost value of row i.
         for (int j = 1; j < i; j++) {
           pt[i][j] = pt[i - 1][j - 1] + pt[i - 1][j];  // Sum 2 entries above.
         }
         pt[i][i] = 1;                                  // Rightmost value of row i.
       }
       return pt;
     }
```

Our array objects look like this:



CS 61B: Lecture 6
Monday, February 3, 2014

Today's reading: Sierra & Bates pp. 282-285.

MORE ARRAYS

Automatic Array Construction

Last lecture, we used a loop to construct all the arrays that the top-level array references. This was necessary to construct a triangular array. But if you want a rectangular multi-dimensional array, rather than a triangular one, Java can construct all of the arrays for you at once.

```
int[][] table = new int[x][y];
```

This declaration constructs an array of x references to arrays. It also constructs x arrays of y ints. The variable "table" references the array of arrays; and each entry in the array of arrays references one of the arrays of ints. All the arrays are constructed for you at once. Similarly, Java can construct three- or ten-dimensional arrays for you, memory permitting.

We could have used a square array to store Pascal's Triangle, but that would have unnecessarily wasted memory. If you have enough memory, you might not care.

When you declare a variable, you can also construct array entries by using initializers.

```
Human[] b = {amanda, rishi, new Human("Paolo")};
int[][] c = {{7, 3, 2}, {x}, {8, 5, 0, 0}, {y + z, 3}};
```

In the second example, Java constructs a non-rectangular two-dimensional array, composed of one array of arrays and four arrays of ints.

Outside of declarations, you need a more complicated notation.

```
d = new int[] {3, 7};
f(new int[] {1, 2, 3});
```

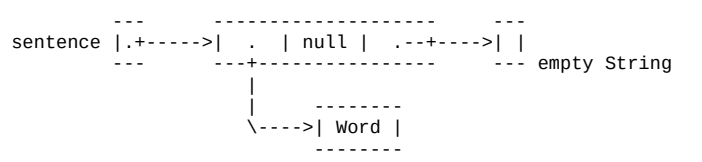
Another subtlety of array declarations is the following.

```
int[] a, b, c;                          // a, b, and c all reference arrays.
int a[], b, c[][];          // a is 1D; c is 2D; b is not a reference/array.
int[] a, b[];            // a references a 1D array; b references a 2D array.
```

Arrays of Objects

When you construct a multi-dimensional array, Java can construct all the arrays for you. But when you construct an array of objects, Java does not construct the objects automatically. The array contains space for references to the objects. You must construct the objects yourself.

```
String[] sentence = new String[3];
sentence[0] = "Word";
sentence[2] = new String();
```



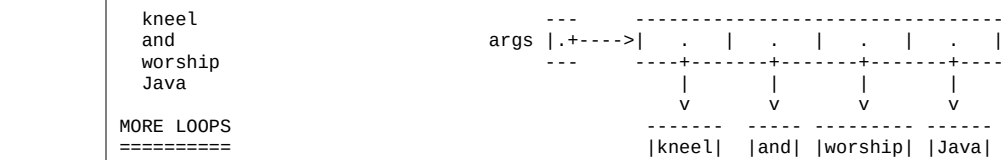
main()'s Parameter

What is the array of Strings that the main() method takes as a parameter? It's a list of command-line arguments sent to your Java program, prepared for you by Java. Consider the following program.

```
class Echo {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++) {
      System.out.println(args[i]);
    }
  }
}
```

If we compile this and type "java Echo kneel and worship Java", java prints

```
kneel
and
worship
Java
```



MORE LOOPS

"do" Loops

A "do" loop has just one difference from a "while" loop. If Java reaches a "do" loop, it _always_ executes the loop body at least once. Java doesn't check the loop condition until the end of the first iteration. "do" loops are appropriate for any loop you always want executed at least once, especially if the variables in the condition won't have meaningful assignments until the loop body has been executed.

```
do {
  s = keybd.readLine();
  process(s);
} while (s.length() > 0);              // Exit loop if s is an empty String.
```

The "break" and "continue" Statements

A "break" statement immediately exits the innermost loop or "switch" statement enclosing the "break", and continues execution at the code following the loop or "switch".

In the loop example above, we might want to skip "process(s)" when s is a signal to exit (in this case, an empty String). We want a "time-and-a-half" loop--we want to enter the loop at a different point in the read-process cycle than we want to exit the loop at. Here are two alternative loops that do the right thing. They behave identically. Each has a different disadvantage.

```
s = keybd.readLine();
while (s.length() > 0) {
  process(s);
  s = keybd.readLine();
}
```

Disadvantage: The line "s = keybd..." is repeated twice. It's not really a disadvantage here, but if input took 100 lines of code, the duplication would make the code harder to maintain. Why? Because a programmer improving the code might change one copy of the duplicated code without noticing the need to change the other to match.

```
while (true) {       // Loop forever.
  s = keybd.readLine();
  if (s.length() == 0) {
    break;
  }
  process(s);
}
```

Disadvantage: Somewhat obfuscated for the reader, because the loop isn't aligned with its natural endpoint.

Some loops have more than one natural endpoint. Suppose we want to iterate the read-process loop at most ten times. In the example at left below, the "break" statement cannot be criticized, because the loop has two natural endpoints. We could get rid of the "break" by writing the loop as at right below, but the result is longer and harder to read.

```
for (int i = 0; i < 10; i++) {
  s = keybd.readLine();
  if (s.length() == 0) {
    break;
  }
  process(s);
}
```

```
int i = 0;
do {
  s = keybd.readLine();
  if (s.length() > 0) {
    process(s);
  }
  i++;
} while ((i < 10) &&
         (s.length() > 0));
```

There are anti-break zealots who claim that the loop on the right is the "correct" way to do things. I disagree, because the left loop is clearly more readable.

Some of the zealots feel this way because "break" statements are a little bit like the "go to" statements found in some languages like Basic and Fortran (and the machine language that microprocessors really execute). "go to" statements allow you to jump to any line of code in the program. It sounds like a good idea at first, but it invariably leads to insanely unmaintainable code. For example, what happens if you jump to the middle of a loop? Turing Award winner Edsger Dijkstra wrote a famous article in 1968 entitled "Go To Statement Considered Harmful", which is part of the reason why many modern languages like Java don't have "go to" statements.

Both "break" and "return" are limited forms of "go to" statements. Their limitations prohibit the worst abuses of "go to". They allow control flow to jump in your program in ways that are straightforward to understand.

WARNING: It's easy to forget exactly where a "break" statement will jump to. For example, "break" does not jump to the end of the innermost enclosing "if" statement. An AT&T programmer introduced a bug into telephone switching software in a procedure that contained a "switch" statement, which contained an "if" clause, which contained a "break", which was intended for the "if" clause, but instead jumped to the end of the "switch" statement. As a result, on January 15, 1990, AT&T's entire U.S. long distance service collapsed for eleven hours. (That code was actually written in C, but Java and C use identical syntax and semantics for loops, "switch", and "break".)

The "continue" statement is akin to the "break" statement, except
(1) it only applies to loops, and
(2) it jumps to the end of the loop body but it doesn't necessarily exit the loop; another iteration will commence if the loop condition is satisfied.

Finally, I told you that "for" loops are identical to certain "while" loops, but there's actually a subtle difference when you use "continue". What's the difference between the following two loops?

```
int i = 0;
while (i < 10) {
  if (condition(i)) {
    continue;
  }
  call(i);
  i++;
}
```

```
for (int i = 0; i < 10; i++) {
  if (condition(i)) {
    continue;
  }
  call(i);
}
```

Answer: when "continue" is called in the "while" loop, "i++" is not executed. In the "for" loop, however, i is incremented at the end of _every_ iteration, even iterations where "continue" is called.

CONSTANTS

Java's "final" keyword is used to declare a value that can never be changed. If you find yourself repeatedly using a numerical value with some "meaning" in your code, you should probably turn it into a "final" constant.

```
BAD:     if (month == 2) {

GOOD:    public final static int FEBRUARY = 2;    // Usually near top of class.

         ...

         if (month == FEBRUARY) {
```

Why? Because if you ever need to change the numerical value assigned to February, you'll only have to change one line of code, rather than hundreds.

You can't change the value of FEBRUARY after it is declared and initialized. If you try to assign another value to FEBRUARY, you'll have a compiler error.

The custom of rendering constants in all-caps is long-established and was inherited from C. (The compiler does not require it, though.)

For any array x, "x.length" is a "final" field.

You can declare local parameters "final" to prevent them from being changed.

```
void myMethod(final int x) {
  x = 3;                              // Compiler ERROR.  Don't mess with X's!
}
```

"final" is usually used for class variables (static fields) and parameters, but it can be used for instance variables (non-static fields) and local variables too. It only makes sense for these to be "final" if the variable is declared with an initializer that calls a method or constructor that doesn't always return the same value.

```
class Bob {
  public final long creationTime = System.currentTimeMillis();
}
```

When objects of the Bob class are constructed, they record the time at that moment. Afterward, the creationTime can never be changed.

SCOPE

The _scope_ of a variable is the portion of the program that can access the variable. Here are some of Java's scoping rules.

- Local variables and parameters are in scope only inside the method that declares them. Furthermore, a local variable is in scope only from the variable declaration down to the innermost closing brace that encloses it. A local variable declared in the initialization part of a "for" loop is in scope only in the loop body.
- Class variables (static fields) are in scope everywhere in the class, except when shadowed by a local variable or parameter of the same name.
- Instance variables (non-static fields) are in scope in non-static methods of the class, except when shadowed.

CS 61B: Lecture 7
Wednesday, February 5, 2014

Today's reading: Goodrich & Tamassia, Section 3.2.

LISTS
=====
Let's consider two different data structures for storing a list of things:
an array and a linked list.

An array is a pretty obvious way to store a list, with a big advantage: it
enables very fast access of each item. However, it has two disadvantages.

First, if we want to insert an item at the beginning or middle of an array, we
have to slide a lot of items over one place to make room. This takes time
proportional to the length of the array.

Second, an array has a fixed length that can't be changed. If we want to add
items to the list, but the array is full, we have to allocate a whole new array
and move all the ints from the old array to the new one.

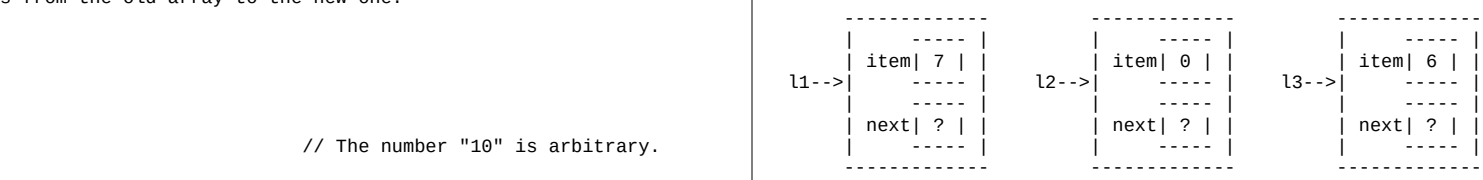
```
public class AList {
  int a[];
  int lastItem;

  public AList() {
    a = new int[10];                          // The number "10" is arbitrary.
    lastItem = -1;
  }

  public void insertItem(int newItem, int location) {
    int i;

    if (lastItem + 1 == a.length) {              // No room left in the array?
      int b[] = new int[2 * a.length];  // Allocate a new array, twice as long.
      for (i = 0; i <= lastItem; i++) {      // Copy items to the bigger array.
        b[i] = a[i];
      }
      a = b;                    // Replace the too-small array with the new one.
    }
    for (i = lastItem; i >= location; i--) {       // Shift items to the right.
      a[i + 1] = a[i];
    }
    a[location] = newItem;
    lastItem++;
  }
}
```

LINKED LISTS (a recursive data type)
============
We can avoid these problems by choosing a Scheme-like representation of lists.
A linked list is made up of _nodes_. Each node has two components: an item,
and a reference to the next node in the list. These components are analogous
to "car" and "cdr". However, our node is an explicitly defined object.

```
  public class ListNode {          // ListNode is a recursive type
    public int item;
    public ListNode next;          // Here we're using ListNode before
  }                                //   we've finished declaring it.
```

Let's make some ListNodes.

```
  ListNode l1 = new ListNode(), l2 = new ListNode(), l3 = new ListNode();
  l1.item = 7;
  l2.item = 0;
  l3.item = 6;
```

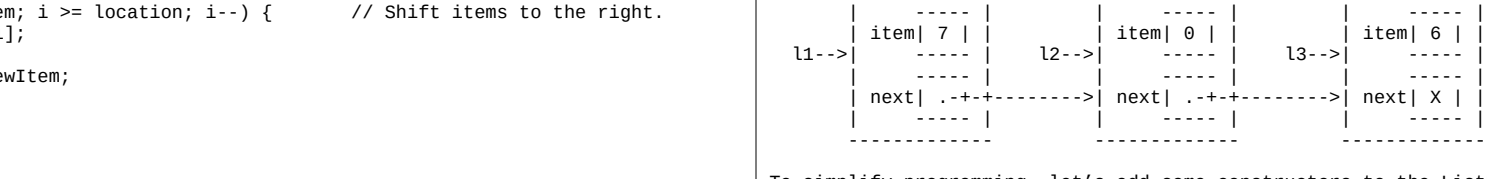


Now let's link them together.

```
  l1.next = l2;
  l2.next = l3;
```

What about the last node? We need a reference that doesn't reference anything.
In Java, this is called "null".

```
  l3.next = null;
```



To simplify programming, let's add some constructors to the ListNode class.

```
public ListNode(int i, ListNode n) {
  item = i;
  next = n;
}

public ListNode(int i) {
  this(i, null);
}
```

These constructors allow us to emulate Scheme's "cons" operation.

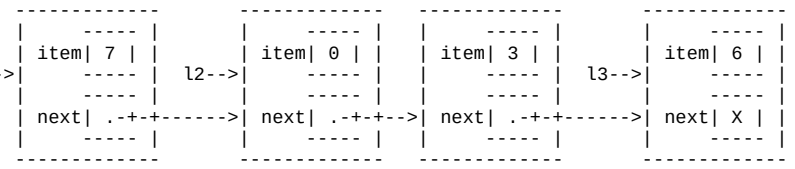
```
  ListNode l1 = new ListNode(7, new ListNode(0, new ListNode(6)));
```

Linked lists vs. array lists

Linked lists have several advantages over array-based lists. Inserting an item into the middle of a linked list takes just a small constant amount of time, if you already have a reference to the previous node (and don't have to walk through the whole list searching for it). The list can keep growing until memory runs out.

The following method inserts a new item into the list immediately after "this".

```
  public void insertAfter(int item) {
    next = new ListNode(item, next);
  }

  l2.insertAfter(3);
```



However, linked lists have a big disadvantage compared to arrays. Finding the nth item of an array takes a tiny, constant amount of time. Finding the nth item of a linked list takes time proportional to n. You have to start at the head of the list and walk forward n - 1 nodes, one "next" at a time.

Many of the data structures we will study in this class will be attempts to find a compromise between arrays and linked lists. We'll learn data structures that are fast for both arbitrary lookups (like arrays) _and_ arbitrary insertions (like linked lists).

Lists of Objects

For greater generality, let's change ListNodes so that each node contains not an int, but a reference to any Java object. In Java, we can accomplish this by declaring a reference of type Object.

```
  public class SListNode {
    public Object item;
    public SListNode next;
  }
```

The "S" in "SListNode" stands for singly-linked. This will make sense when we contrast these lists with doubly-linked lists later. You'll see the SListNode class in next week's lab and homework.

A List Class

There are two problems with SListNodes.

(1) Suppose x and y are pointers to the same shopping list. Suppose we insert a new item at the beginning of the list thusly:

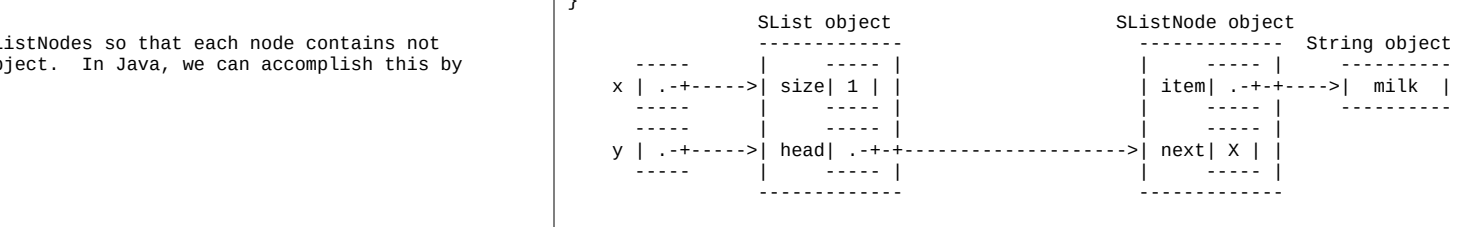
```
      x = new SListNode("soap", x);
```

y doesn't point to the new item; y still points to the second item in x's list. If y goes shopping for x, he'll forget to buy soap.

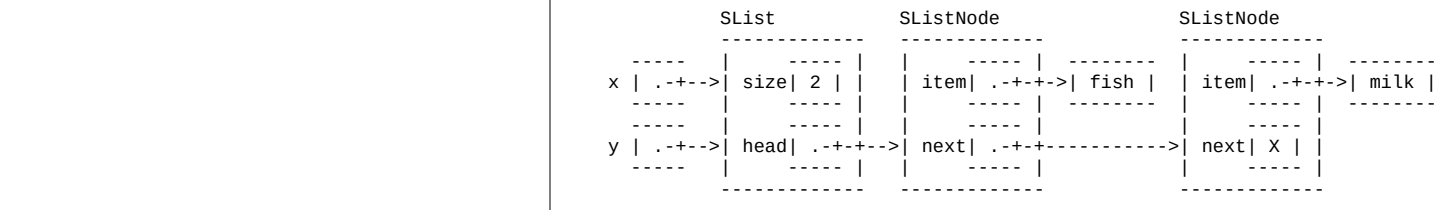
(2) How do you represent an empty list? The obvious way is "x = null". However, Java won't let you call a SListNode method--or any method--on a null object. If you write "x.insertAfter(item)" when x is null, you'll get a run-time error, even though x is declared to be a SListNode. (There are good reasons for this, which you'll learn later in the course.)

The solution is a separate SList class, whose job is to maintain the head (first node) of the list. We will put many of the methods that operate on lists in the SList class, rather than the SListNode class.

```
public class SList {
  private SListNode head;             // First node in list.
  private int size;                   // Number of items in list.

  public SList() {                    // Here's how to represent an empty list.
    head = null;
    size = 0;
  }

  public void insertFront(Object item) {
    head = new SListNode(item, head);
    size++;
  }
}
```



Now, when you call x.insertFront("fish"), every reference to that SList can see the change.



Another advantage of the SList class is that it can keep a record of the SList's size (number of SListNodes). Hence, the size can be determined more quickly than if the SListNodes had to be counted.

CS 61B: Lecture 8
Wednesday, February 5, 2014

Today's reading: Goodrich & Tamassia, Section 3.3.

THE "public" AND "private" KEYWORDS

Thus far, we've usually declared fields and methods using the "public" keyword. However, we can also declare a field or method "private". A private method or field is invisible and inaccessible to other classes, and can be used only within the class in which the field or method is declared.

Why would we want to make a field or method private?
(1) To prevent data within an object from being corrupted by other classes.
(2) To ensure that you can improve the implementation of a class without causing other classes that depend on it to fail.

In the following example, EvilTamperer tries to get around the error checking code of the Date class by fiddling with the internals of a Date object.

```
public class Date {                    |  public class EvilTamperer {
  private int day;                     |    public void tamper() {
  private int month;                   |      Date d = new Date(1, 1, 2006);
                                       |
  private void setMonth(int m) {       |      d.day = 100;    // Foiled!!
    month = m;                         |      d.setMonth(0);  // Foiled again!!
  }                                    |    }
                                       |  }
  public Date(int month, int day) {    |
    [Implementation with               |
     error-checking code here.]        |
  }
}
```

However, javac won't compile EvilTamperer, because the Date class has declared its vulnerable parts "private". setMonth is an internal helper method used within the Date class, whereas the Date constructor is a public part of the interface of the Date class. Error-checking code in the constructor ensures that invalid Dates are not constructed.

Here are some important definitions.

The _interface_ of a class is a set of prototypes for public methods (and sometimes public fields), plus descriptions of the methods' behaviors.

An _Abstract_Data_Type_ (ADT) is a class that has a well-defined interface, but its implementation details are firmly hidden from other classes. That way, you can change the implementation of a class without jeopardizing the programs that depend on it. The Date class is an ADT. We'll implement lots of ADTs this semester.

An _invariant_ is a fact about a data structure that is always true (assuming the code is bug-free), no matter what methods are called by external classes. For example, the Date ADT enforces the invariant that a Date object always represents a valid date. An invariant is enforced by allowing access to certain fields only through method calls.

An ADT is often a good thing to aspire to. In most of your classes, you should declare all fields private, as well as helper functions meant only for internal use, so that you can maintain sensible invariants on your data structures.

However, not all classes are ADTs! Some classes are nothing more than data storage units, and do not need to enforce any invariants. In such classes, all fields may be declared public.

The SList ADT

Last lecture, I created an SList class to solve the problems of representing empty lists and inserting items at the beginning of a list. Today, I want to introduce another advantage of the SList class.

We want the SList ADT to enforce two invariants:
(1) An SList's "size" variable is always correct.
(2) A list is never circularly linked; there is always a tail node whose "next" reference is null.

Both these goals are accomplished by making sure that _only_ the methods of the SList class can change the lists' internal data structures. SList ensures this by two means:
(1) The fields of the SList class (head and size) are declared "private".
(2) No method of SList returns an SListNode.

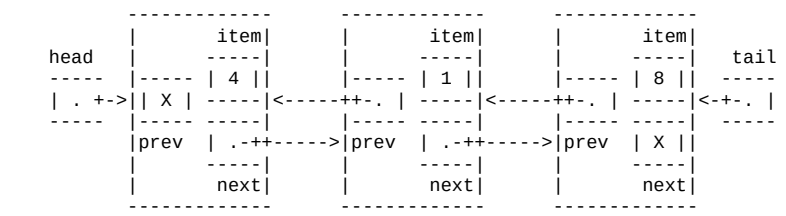
The first rule is necessary so that the evil tamperer can't change the fields and corrupt the SList or violate invariant (1). The second rule prevents the evil tamperer from changing list items, truncating a list, or creating a cycle in a list, thereby violating invariant (2).

DOUBLY-LINKED LISTS
==========
As we saw last class, inserting an item at the front of a linked list is easy. Deleting from the front of a list is also easy. However, inserting or deleting an item at the end of a list entails a search through the entire list, which might take a long time. (Inserting at the end is easy if you have a 'tail' pointer, as you will learn in Lab 3, but deleting is still hard.)

A doubly-linked list is a list in which each node has a reference to the previous node, as well as the next node.

```
  class DListNode {
    Object item;
    DListNode next;
    DListNode prev;
  }
```

```
  class DList {
    private DListNode head;
    private DListNode tail;
  }
```



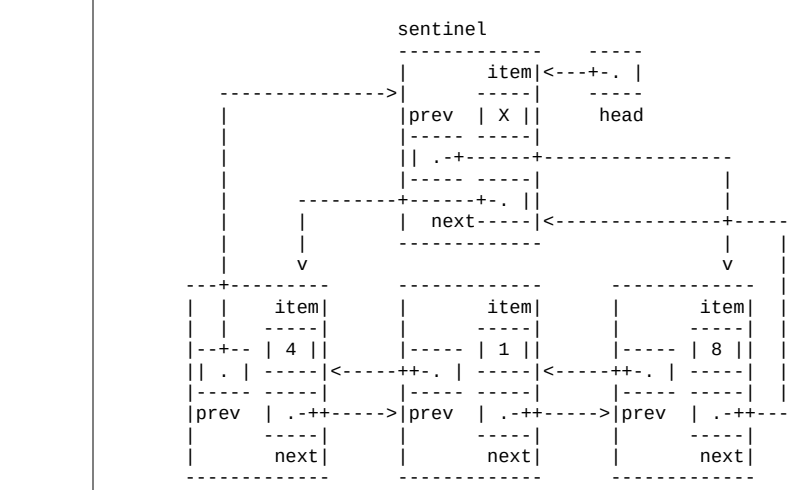
DLists make it possible to insert and delete items at both ends of the list, taking constant running time per insertion and deletion. The following code removes the tail node (in constant time) if there are at least two items in the DList.

```
  tail.prev.next = null;
  tail = tail.prev;
```

You'll need a special case for a DList with no items. You'll also need a special case for a DList with one item, because tail.prev.next does not exist. (Instead, head needs to be changed.)

Let's look at a clever trick for reducing the number of special cases, thereby simplifying our DList code. We designate one DListNode as a _sentinel_, a special node that does not represent an item. Our list representation will be circularly linked, and the sentinel will represent both the head and the tail of the list. Our DList class no longer needs a tail pointer, and the head pointer points to the sentinel.

```
  class DList {
    private DListNode head;
    private int size;
  }
```



The invariants of the DList ADT are more complicated than the SList invariants. The following invariants apply to the DList with a sentinel.
(1) For any DList d, d.head != null. (There's always a sentinel.)
(2) For any DListNode x, x.next != null.
(3) For any DListNode x, x.prev != null.
(4) For any DListNode x, if x.next == y, then y.prev == x.
(5) For any DListNode x, if x.prev == y, then y.next == x.
(6) A DList's "size" variable is the number of DListNodes, NOT COUNTING the sentinel (denoted by "head"), that can be accessed from the sentinel by a sequence of "next" references.

An empty DList is represented by having the sentinel's prev and next fields point to itself.

Here's an example of a method that removes the last item from a DList.

```
  public void removeBack() {
    if (head.prev != head) {        // Do nothing if the DList is empty.
      head.prev = head.prev.prev;   // Sentinel now points to second-last item.
      head.prev.next = head;        // Second-last item now points to sentinel.
      size--;
    }
  }
```

In Lab 4 and Homework 4, you'll implement more methods for this DList class.

CS 61B: Lecture 9
Monday, February 10, 2014

Today's reading: Sierra & Bates pp. 77, 235-239, 258-265, 663.

THE STACK AND THE HEAP

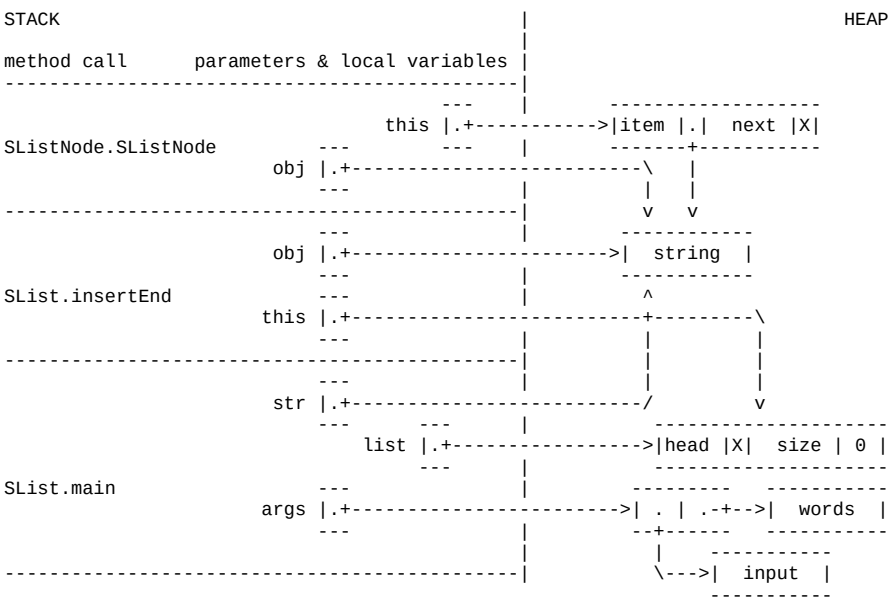
Java stores stuff in two separate pools of memory: the stack and the heap.

The _heap_ stores all objects, including all arrays, and all class variables (i.e. those declared "static").

The _stack_ stores all local variables, including all parameters.

When a method is called, the Java Virtual Machine creates a _stack_frame_ (also known as an _activation_record_) that stores the parameters and local variables for that method. One method can call another, which can call another, and so on, so the JVM maintains an internal _stack_ of stack frames, with "main" at the bottom, and the most recent method call on top.

Here's a snapshot of the stack while Java is executing the SList.insertEnd method. The stack frames are on the left. Everything on the right half of the page is in the heap. Read the stack from bottom to top, because that's the order in which the stack frames were created.



The method that is currently executing (at any point in time) is the one whose stack frame is on top. All the other stack frames represent methods waiting for the methods above them to return before they can continue executing.

When a method finishes executing, its stack frame is erased from the top of the stack, and its local variables are erased forever.

The java.lang library has a method "Thread.dumpStack" that prints a list of the methods on the stack (but it doesn't print their local variables). This method can be convenient for debugging--for instance, when you're trying to figure out which method called another method with illegal parameters.

Parameter Passing

As in Scheme, Java passes all parameters _by_value_. This means that the method has _copies_ of the actual parameters, and cannot change the originals. The copies reside in the method's stack frame for the method. The method can change these copies, but the original values that were copied are not changed.

In this example, the method doNothing sets its parameter to 2, but it has no effect on the value of the calling method's variable a:

```
method:                              | STACK (just before the method returns)
                                     |
  static void doNothing(int x) {     |      -----
    x = 2;                           |    x | 2 |
  }                                  |      -----     stack frame for doNothing
                                     |----------------------------------------
method call:                         |
                                     |      -----
  int a = 1;                         |    a | 1 |
  doNothing(a);                      |      -----     stack frame for main
```

When the method call returns, a is still 1. The doNothing method, as its name suggests, failed to change the value of a or do anything relevant at all.

However, when a parameter is a reference to an object, the reference is copied, but the object is not; the original object is shared. A method can modify an object that one of its parameters points to, and the change will be visible everywhere. Here's an example that shows how a method can make a change to an object that is visible to the calling method:

```
method:                              | STACK              | HEAP
                                     |              set3|
class IntBox {                       |      -----         |
  public int i;                      |   ib | .-+----------------\
  static void set3(IntBox ib) {      |      -----         |      |
    ib.i = 3;                        |                    |      |
  }                                  |--------------------|      v
                                     |      -----         |    ------
method call:                         |    b | .-+-------------->|i |3|
                                     |      -----     main|    ------
  IntBox b = new IntBox();
  set3(b);
```

For those of you who are familiar with programming languages that have "pass by reference," the example above is as close as you can get in Java. But it's not "pass by reference." Rather, it's passing a reference by value.

Here's an example of a common programming error, where a method tries and fails to make a change that is visible to the calling method. (Assume we've just executed the example above, so b is set up.)

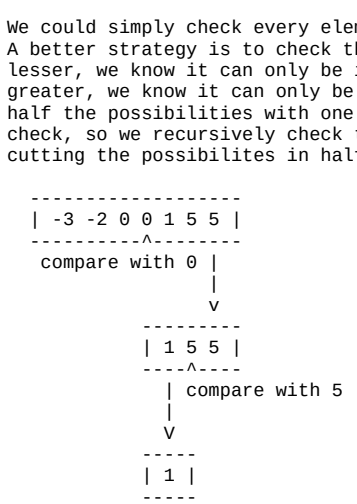
```
method:                              | STACK              | HEAP
                                     |            badSet4|
class IntBox {                       |      -----         |    ------
  static void badSet4(IntBox ib) {   |   ib | .-+-------------->|i |4|
    ib = new IntBox();               |      -----         |    ------
    ib.i = 4;                        |                    |
  }                                  |--------------------|
                                     |      -----         |    ------
method call:                         |    b | .-+-------------->|i |3|
                                     |      -----     main|    ------
  badSet4(b);
```

Binary search

When a method calls itself recursively, the JVM's internal stack holds two or more stack frames connected with that method. Only the top one can be accessed.

Here's a recursive method that searches a sorted array of ints for a particular int. Let i be an array of ints sorted from least to greatest--for instance, {-3, -2, 0, 0, 1, 5, 5}. We want to search the array for the value "findMe". If we find "findMe", we return its array index; otherwise, we return FAILURE.

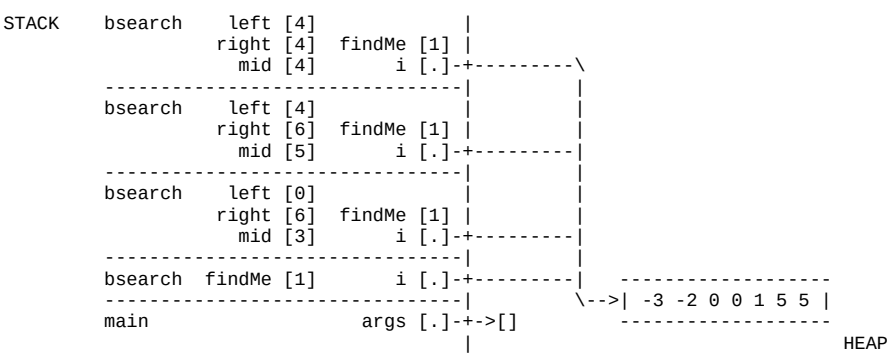
We could simply check every element of the array, but that would be slow. A better strategy is to check the middle array element first. If findMe is lesser, we know it can only be in the left half of the array; if findMe is greater, we know it can only be in the right half. Hence, we've eliminated half the possibilities with one comparison. We still have half the array to check, so we recursively check the middle element of that half, and so on, cutting the possibilites in half each time. Suppose we search for 1.



The recursion has two base cases.
(1) If findMe equals the middle element, return its index; in the example above, we return index 4.
(2) If we try to search a subarray of length zero, the array does not contain "findMe", and we return FAILURE.

```
  public static final int FAILURE = -1;

  private static int bsearch(int[] i, int left, int right, int findMe) {
    if (left > right) {
      return FAILURE;                   // Base case 2:  subarray of size zero.
    }
    int mid = (left + right) / 2;            // Halfway between left and right.
    if (findMe == i[mid]) {
      return mid;                                      // Base case 1:  success!
    } else if (findMe < i[mid]) {
      return bsearch(i, left, mid - 1, findMe);            // Search left half.
    } else {
      return bsearch(i, mid + 1, right, findMe);          // Search right half.
    }
  }

  public static int bsearch(int[] i, int findMe) {
    return bsearch(i, 0, i.length - 1, findMe);
  }
```

How long does binary search take? Suppose the array has n elements. In one call to bsearch, we eliminate at least half the elements from consideration. Hence, it takes log_2 n (the base 2 logarithm of n) bsearch calls to pare down the possibilities to one. Binary search takes time proportional to log_2 n. If you're not comfortable with logarithms, please review Goodrich & Tamassia Sections 4.1.2 & 4.1.7.



The stack frames appear at right in the figure above. There are three different local variables named "left" on the stack, three named "right", three named "mid", four named "i", and four named "findMe". While the current invocation of bsearch() is executing, only the topmost copy of "left" is in scope, and likewise for "right" and "mid". The other copies are hidden and cannot be accessed or changed until the current invocation of bsearch() terminates.

Most operating systems give a program enough stack space for a few thousand stack frames. If you use a recursive procedure to walk through a million-node list, Java will try to create a million stack frames, and the stack will run out of space. The result is a run-time error. You should use iteration instead of recursion when the recursion will be very deep.

However, our recursive binary search method does not have this problem. Most modern microprocessors cannot address more than 2^64 bytes of memory. Even if an array of bytes takes this much space, we will only have to cut the array in half 64 times to run a binary search. There's room on the stack for 64 stack frames, with plenty to spare. In general, recursion to a depth of roughly log n (where n is the number of items in a data structure) is safe, whereas recursion to a depth of roughly n is not.

Unfortunately, binary search can't be used on linked lists. Think about why.

Scope and Recursion

The _scope_ of a variable is the portion of the program that can access the variable. Here are some of Java's scoping rules.

- Local variables and parameters are in scope only inside the method that declares them, and only for the topmost stack frame. Furthermore, a local variable is in scope only from the variable declaration down to the innermost closing brace that encloses it. A local variable declared in the initialization part of a "for" loop is in scope only in the loop body.
- Class variables (static fields) are in scope everywhere in the class, except when shadowed by a local variable or parameter of the same name.
- Fully qualified class variables ("System.out", rather than "out") are in scope everywhere in the class, and cannot be shadowed. If they're public, they're in scope in _all_ classes.
- Instance variables (non-static fields) are in scope in non-static methods of the class, except when shadowed.
- Fully qualified instance variables ("amanda.name", "this.i") are in scope everywhere in the class, and cannot be shadowed. If they're public, they're in scope in all classes.

CS 61B: Lecture 10
Wednesday, February 12, 2014

Today's reading: All of Chapter 7, plus pp. 28-33, 250-257.

INHERITANCE

In Lab 3, you modified several methods in the SList class so that a "tail" reference could keep track of the end of the list, thereby speeding up the insertEnd() method.

We could have accomplished the same result without modifying SList--by creating a new class that inherits all the properties of SList, and then changing only the methods that need to change. Let's create a new class called TailList that inherits the fields and methods of the original SList class.

```
  public class TailList extends SList {
    // The "head" and "size" fields are inherited from SList.
    private SListNode tail;
```

This code declares a TailList class that behaves just like the SList class, but has an additional field "tail" not present in the SList class. TailList is said to be a _subclass_ of SList, and SList is the _superclass_ of TailList. A TailList has three fields: head, size, and tail.

A subclass can modify or augment a superclass in at least three ways:
(1) It can declare new fields.
(2) It can declare new methods.
(3) It can override old methods with new implementations.

We've already seen an example of the first. Let's try out the third. The advantage of TailList is that it can perform the insertEnd() method much more quickly than a tail-less SList can. So, let's write a new insertEnd() for TailList, which will _override_ SList's old, slow insertEnd() method.

```
    public void insertEnd(Object obj) {
      // Your solution to Lab 3 goes here.
    }
```

The isEmpty(), length(), nth(), and toString() methods of SList do not need any changes on account of the tail reference. These methods are inherited from SList, and there's no need to rewrite them.

Inheritance and Constructors

What happens when we construct a TailList? Java executes a TailList constructor, as you would expect, but _first_ it executes the code in the SList() constructor. The TailList constructor should initialize fields unique to TailList. It can also modify the work done by SList() if appropriate.

```
    public TailList() {
      // SList() constructor called automatically; sets size = 0, head = null
      tail = null;
    }
```

The zero-parameter SList() constructor is always called by default, regardless of the parameters passed to the TailList constructor. To change this default behavior, the TailList constructor can explicitly call any constructor for its superclass by using the "super" keyword.

```
    public TailList(int x) {
      super(x);
      tail = null;
    }
```

The call to "super()" must be the first statement in the constructor. If a constructor has no explicit call to "super", and its (nearest) superclass has no zero-parameter constructor, a compile-time error occurs. There is no way to tell Java not to call a superclass constructor. You have only the power to choose which of the superclass constructors is called.

Invoking Overridden Methods

Sometimes you want to override a method, yet still be able to call the method implemented in the superclass. The following example shows how to do this. Below, we want to reuse the code in SList.insertFront, but we also need to adjust the tail reference.

```
    public void insertFront(Object obj) {
      super.insertFront(obj);             // Insert at the front of the list.
      if (size == 1) {                    // If necessary,
        tail = head;                      //   adjust the tail reference.
      }
    }
  }
```

Unlike superclass constructor invocations, ordinary superclass method invocations need not be the first statement in a method.

The "protected" Keyword

I lied when I said that we don't need to modify SList. One change is necessary. The "head" and "size" fields in SList must be declared "protected", not "private".

```
  public class SList {
    protected SListNode head;
    protected int size;

    [Method definitions.]
  }
```

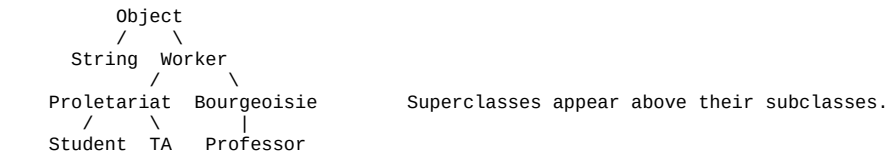
"protected" is a level of protection somewhere between "public" and "private". A "protected" field is visible to the declaring class and all its subclasses, but not to other classes. "private" fields aren't even visible to the subclasses.

If "head" and "size" are declared private, the method TailList.insertFront can't access them and won't compile. If they're declared protected, insertFront can access them because TailList is a subclass of SList.

When you write an ADT, if you think somebody might someday want to write a subclass of it, declare its vulnerable fields "protected", unless you have a reason for not wanting subclasses to see them. Helper methods often should be declared "protected" as well.

Class Hierarchies

Subclasses can have subclasses. Subclassing is transitive: if Proletariat is a subclass of Worker, and Student is a subclass of Proletariat, then Student is a subclass of Worker. Furthermore, _every_ class is a subclass of the Object class (including Java's built-in classes like String and BufferedReader.) Object is at the top of every class hierarchy.



That's why the "item" field in each listnode is of type Object: it can reference any object of any class. (It can't reference a primitive type, though.)

Dynamic Method Lookup

Here's where inheritance gets interesting. Any TailList can masquerade as an SList. An object of class TailList can be assigned to a variable of type SList--but the reverse is not true. Every TailList is an SList, but not every SList is a TailList. It merits repeating:

>>>!!!*** Every TailList *IS* an SList. ***!!!<<< For example:

```
  SList s = new TailList();         // Groovy.
  TailList t = new SList();         // COMPILE-TIME ERROR.
```

Memorize the following two definitions.
  _Static_type_: The type of a variable.
  _Dynamic_type_: The class of the object the variable references.

In the code above, the static type of s is SList, and the dynamic type of s is TailList. Henceforth, I will often just say "type" for static type and "class" for dynamic type.

When we invoke an overridden method, Java calls the method for the object's _dynamic_ type, regardless of the variable's static type.

```
  SList s = new TailList();
  s.insertEnd(obj);                 // Calls TailList.insertEnd()
  s = new SList();
  s.insertEnd(obj);                 // Calls SList.insertEnd()
```

This is called _dynamic_method_lookup_, because Java automatically looks up the right method for a given object at run-time. Why is it interesting?

WHY DYNAMIC METHOD LOOKUP MATTERS (Worth reading and rereading)
Suppose you have a method (in any class) that sorts an SList using only SList method calls (but doesn't construct any SLists). Your method now sorts TailLists too, with no changes.
Suppose you've written a class--let's call it RunLengthEncoding--that uses SLists extensively. By changing the constructors so that they create TailLists instead of SLists, your class immediately realizes the performance improvement that TailLists provide--without changing anything else in the RunLengthEncoding class.

Subtleties of Inheritance
----
(1) Suppose we write a new method in the TailList class called eatTail(). We can't call eatTail on an SList. We can't even call eatTail on a variable of type SList that references a TailList.

```
  TailList t = new TailList();
  t.eatTail();                      // Groovy.
  SList s = new TailList();         // Groovy--every TailList is an SList.
  s.eatTail();                      // COMPILE-TIME ERROR.
```

Why? Because not every object of class SList has an "eatTail()" method, so Java can't use dynamic method lookup on the variable s.

But if we define eatTail() in SList instead, the statements above compile and run without errors, even if no eatTail() method is defined in class TailList. (TailList inherits eatTail() from SList.)

(2) I pointed out earlier that you can't assign an SList object to a TailList variable. The rules are more complicated when you assign one variable to another.

```
  SList s;
  TailList t = new TailList();
  s = t;                            // Groovy.
  t = s;                            // COMPILE-TIME ERROR.
  t = (TailList) s;                 // Groovy.
  s = new SList();
  t = (TailList) s;                 // RUN-TIME ERROR:  ClassCastException.
```

Why does the compiler reject "t = s", but accept "t = (TailList) s"? It refuses "t = s" because not every SList is a TailList, and it wants you to confirm that you're not making a thoughtless mistake. The cast in the latter statement is your way of reassuring the compiler that you've designed the program to make sure that the SList s will always be a TailList.

If you're wrong, Java will find out when you run the program, and will crash with a "ClassCastException" error message. The error occurs only at run-time because Java cannot tell in advance what class of object s will reference.

Recall that SLists store items of type Object. When they're recovered, they usually have to be cast back to a more specific type before they can be used. Suppose we have a list of Integers. Recall that nth() returns type Object.

```
  int x = t.nth(1).intValue();                // COMPILE-TIME ERROR.
  int y = ( (Integer) t.nth(1) ).intValue();  // Groovy.
```

Some methods are defined on every Object, though.

```
  String z = t.nth(1).toString();             // Groovy.
```

(3) Java has an "instanceof" operator that tells you whether an object is of a specific class. WARNING: The "o" in "instanceof" is not capitalized.

```
  if (s instanceof TailList) {
    t = (TailList) s;
  }
```

This instanceof operation will return false if s is null or doesn't reference a TailList. It returns true if s references a TailList object--even if it's a subclass of TailList.

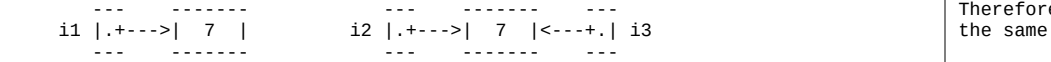
CS 61B: Lecture 11
Wednesday, February 12, 2014

Today's reading: Sierra & Bates, pp. 95-109, 662.

equals()

Every class has an equals() method. If you don't define one explictly, you inherit Object.equals(), for which "r1.equals(r2)" returns the same boolean value as "r1 == r2", where r1 and r2 are references. However, many classes override equals() to compare the _content_ of two objects.

Integer (in the java.lang library) is such a class; it stores one private int. Two distinct Integer objects are equals() if they contain the same int. In the following example, "i1 == i2" is false, but "i1.equals(i2)" is true. "i2 == i3" and "i2.equals(i3)" are both true.



IMPORTANT: r1.equals(r2) throws a run-time exception if r1 is null.

There are at least four different degrees of equality.
(1) Reference equality, ==. (The default inherited from the Object class.)
(2) Shallow structural equality: two objects are "equals" if all their fields are ==. For example, two SLists whose "size" fields are equal and whose "head" fields point to the same SListNode.
(3) Deep structural equality: two objects are "equals" if all their fields are "equals". For example, two SLists that represent the same sequence of items (though the SListNodes may be different).
(4) Logical equality. Two examples:
    (a) Two "Set" objects are "equals" if they contain the same elements, even if the underlying lists store the elements in different orders.
    (b) The Fractions 1/3 and 2/6 are "equals", even though their numerators and denominators are all different.

The equals() method for a particular class may test any of these four levels of equality, depending on what seems appropriate. Let's write an equals() method for SLists that tests for deep structural equality. The following method returns true only if the two lists represent identical sequences of items.

```
  public class SList {
    public boolean equals(Object other) {
      if (!(other instanceof SList)) {          // Reject non-SLists.
        return false;
      }

      SList o = (SList) other;
      if (size != o.size) {
        return false;
      }

      SListNode n1 = head;
      SListNode n2 = o.head;
      while (n1 != null) {
        if (!n1.item.equals(n2.item)) {         // Deep equality of the items.
          return false;
        }
        n1 = n1.next;
        n2 = n2.next;
      }
      return true;
    }
  }
```

Note that this implementation may fail if the SList invariants have been corrupted. (A wrong "size" field or a loop in an SList can make it fail.)

IMPORTANT: Overriding DOESN'T WORK if we change the signature of the original method, even just to change a parameter to a subclass. In the Object class, the signature is equals(Object), so in the code above, we must declare "other" to be an Object too. If we declare "other" to be an SList, the equals() method will compile but it will NOT override. That means the code

```
  Object s = new SList();
  s.equals(s);
```

will call Object.equals(), not SList.equals(). Dynamic method lookup won't care that s is an SList, because the equals() method above is not eligible to override Object.equals().

Therefore, if you want to override a method, make sure the signature is EXACTLY the same.

"for each" LOOPS

Java has a "for each" loop for iterating through the elements of an array.

```
  int[] array = {7, 12, 3, 8, 4, 9};

  for (int i : array) {
    System.out.print(i + " ");
  }
```

Note that i is _not_ iterating from 0 to 5; it's taking on the value of each array element in turn. You can iterate over arrays of any type this way.

```
  String concat = "";
  for (String s : stringArray) {
    concat = concat + s;
  }
```

For some reason, the type declaration _must_ be in the "for" statement. The compiler barfs if you try

```
  int i;
  for (i : array) { ... }
```

TESTING
=======
Complex software, like Project 1, is easier to debug if you write lots of test code. We'll consider three types of testing:

(1) Modular testing: testing each method and each class separately.
(2) Integration testing: testing a set of methods/classes together.
(3) Result verification: testing results for correctness, and testing data structures to ensure they still satisfy their invariants.

(1) Modular Testing

When you write a program and it fails, it can be quite difficult to determine which part of the code is responsible. Even experienced programmers often guess wrong. It's wise to test every method you write individually.

There are two types of test code for modular testing: test drivers and stubs.

(a) Test drivers are methods that call the code being tested, then check the results. In Lab 3 and Homework 3, you've seen test drivers in the SList class that check that your code is doing the right thing.

Both public and private methods should be tested. Hence, a test driver usually needs to be inside the class it tests. In a class intended for use by other classes, the obvious place to put a test driver is in the main() method, as we did in Lab 3 and Homework 3. However, if a class is the entry point for the program, you can't put your test driver in main(). Instead, put it in a method with a name like testDriver(), and then write _another_ class whose main() method calls your test driver.

(b) Stubs are small bits of code that are _called_ by the code being tested. They are often quite short. They serve three purposes.

(i) If you write a method that calls other methods that haven't yet been implemented, you can write simple stubs that fake the missing methods.
(ii) Suppose you are having difficulty determining whether a bug lies in a calling method, or a method it calls. You can temporarily replace the callee with a stub that returns controlled results to the caller, so you can see if the caller is responsible for the problem.
(iii) Stubs allow you to create repeatable test cases that might not arise often in practice. For instance, suppose a subroutine fetches and returns input from an airline database, and your code calls this subroutine. You might want to test whether your code operates correctly when ten airplanes depart at the same time. Such an event might be rare in practice, but you can replace the database access subroutine with a stub that feeds fake data to your code. There are two advantages:

- Stubs can produce test data that the real code rarely or never produces.
- Stubs produce _repeatable_ test data, so that bugs can be reproduced.

(2) Integration Testing

Integration testing is testing all the components together (preferably _after_ you have tested them in isolation). Sometimes bugs arise during integration because your test cases weren't thorough enough. Other times, they arise because of misunderstandings about how the components are supposed to interact with each other. Integration testing is harder than modular testing, because it's harder to determine where a bug is, or to identify your mistaken assumptions about how the components interact.

The most important task in avoiding these bugs is to define your interfaces well and unambiguously. There should be no ambiguity in the descriptions of the behavior of your methods, especially in unusual cases. We'll talk a lot more about this in later lectures.

The best advice I can give on integration testing: learn to use a debugger.

(3) Result Verification

A result verifier is a method that checks the results of other methods. There are at least two types of result verifiers you can write.

(a) Data structure integrity checkers. A method can inspect a data structure (like a list) and verify that all the invariants are satisfied. For Project 1, we are asking you to write a simple checker named "check()" that verifies the integrity of your run-length encodings.
(b) Algorithm result checkers. A method can inspect the output of another method for correctness. For example, if a method is supposed to sort an array of numbers, a result checker can walk through the output and check that each item really is less than or equal to its successor.

An _assertion_ is a piece of code that tests an invariant or a result. Java offers an "assert" keyword that tests whether an assertion evaluates to "true". If the assertion comes up "false", Java terminates the program with an "AssertionError" error message, a stack trace, and an optional message of your own choosing.

```
  assert x == 3;
  assert list.size == list.countLength() : "wrong SList size:  " + list.size;
```

At the end of each method that changes a data structure, add assertions (possibly a call to an integrity checker). At the end of each method that computes a result, add an assertion that calls a result checker.

Assertions are convenient because you can turn them on or off. To turn them on when you're testing your code, run your code with "java -ea" (for "enable assertions"). To turn them off for greater speed, run with "java -da" (for "disable assertions"). The default (if you specify no switch) is -da. WARNING: when assertions are turned off, the method "list.countLength()" above is never called. Good for speed, but countLength() must not perform a task that is necessary for your program's correctness.

Regression Testing

A _regression_test_ is a test suite can be re-run whenever changes are made to the code. Nearly every software company has reams of regression tests for each product. They run them again every time they fix a bug or add a feature.

Some principles of regression testing:

(a) All-paths testing: your test cases should try to test every path through the code. Test every method. For every "if" statement, you should try to write a test case for each of the two paths.
(b) "Boundary cases" should be tested, as well as non-boundary cases. For instance, if you write a binary search method, test it on arrays of lengths zero and one, as well as longer lengths. Test the cases where the item sought is the first element, the last element, in the middle, not present. For every loop in the code, try to test the cases where it iterates zero or one times, as well as the case where it iterates several times. Test the branch "if (x >= 1)" for x equal to 0, 1, and 2.
(c) Generally, methods can be divided into two types: extenders, which construct or change an object; and observers, which return information about an object. (Some methods do both, but you should always think hard about whether that's good design.) Ideally, your test cases should test every combination of extender and observer.

In real-world software development, the size of the test code is often larger than the size of the code being tested.

CS 61B: Lecture 12
Wednesday, February 19, 2014

Today's reading: Sierra & Bates, Chapter 8.

ABSTRACT CLASSES

An abstract class is a class whose sole purpose is to be extended.

```
public abstract class List {
  protected int size;

  public int length() {
    return size;
  }

  public abstract void insertFront(Object item);
}
```

Abstract classes don't allow you to create objects directly. You can declare a variable of type List, but you can't create a List object.

```
  List myList;                // Right on.
  myList = new List();        // COMPILE-TIME ERROR.
```

However, abstract classes can be extended in the same way as ordinary classes, and the subclasses are usually not abstract. (They can be, but usually they're normal subclasses with complete implementations.)

The abstract List class above includes an abstract method, insertFront. An abstract method lacks an implementation. One purpose of an abstract method is to guarantee that every non-abstract subclass will implement the method. Specifically, every non-abstract subclass of List must have an implementation for the insertFront method.

```
  public class SList extends List {
    // inherits the "size" field.
    protected SListNode head;

    // inherits the "length" method.

    public void insertFront(Object item) {
      head = new SListNode(item, head);
      size++;
    }
  }
```

If you leave out the implementation of insertFront in SList, the Java compiler will complain that you must provide one. A non-abstract class may never contain an abstract method, nor inherit one without providing an implementation.

Because SList is not abstract, we can create SList objects; and because SLists are Lists, we can assign an SList to a List variable.

```
  List myList = new SList();   // Right on.
  myList.insertFront(obj);     // Right on.
```

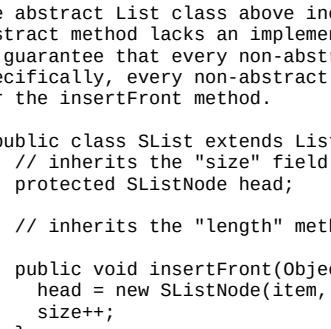
What are abstract classes good for? It's all about the interface.

```
 ------------------------------------------------
|  An abstract class lets you define an interface  |
|    - for multiple classes to share,              |
|    - without defining any of them yet.           |
 ------------------------------------------------
```

Let's consider the List class. Although the List class is abstract, it is an ADT--even without any implementation!-- because it has an interface with public method prototypes and well-defined behaviors. We can implement an algorithm--for example, a list sorter--based on the List interface, without ever knowing how the lists will be implemented. One list sorter can sort every kind of List.

```
  public void listSort(List l) { ... }
```

In another part of the universe, your project partners can build lots of subclasses of List: SList, DList, TailList, and so on. They can also build special-case List subclasses: for example, a TimedList that records the amount of time spent doing List operations, and a TransactionList that logs all changes made to the list on a disk so that all information can be recovered if a power outage occurs. A library catalogue application that uses DLists can send them to your listSort algorithm to be sorted. An airline flight database that uses TransactionLists can send them to you for sorting, too, and you don't have to change a line of sorting code. You may have written your list sorter years before TransactionLists were ever thought of.



The list sorter is built on the foundation of a list ADT, and the application is built on the foundation of the list sorter. However, it's the application, and not the list sorter, that gets to choose what kind of list is actually used, and thereby obtains special features like transaction logging. This is a big advantage of object-oriented languages like Java.

JAVA INTERFACES
===============
Java has an "interface" keyword which refers to something quite different than the interfaces I defined in Lecture 8, even though the two interfaces are related. Henceforth, when I say "interfaces" I mean public fields, public method prototypes, and the behaviors of public methods. When I say "Java interfaces" I mean Java's "interface" keyword.

A Java interface is just like an abstract class, except for two differences.
(1) In Java, a class can inherit from only one class, even if the superclass is an abstract class. However, a class can "implement" (inherit from) as many Java interfaces as you like.
(2) A Java interface cannot implement any methods, nor can it include any fields except "final static" constants. It only contains method prototypes and constants.

```
public interface Nukeable {               // In Nukeable.java
  public void nuke();
}

public interface Comparable {             // In java.lang
  public int compareTo(Object o);
}

public class SList extends List implements Nukeable, Comparable {
  [Previous stuff here.]

  public void nuke() {
    head = null;
    size = 0;
  }

  public int compareTo(Object o) {
    [Returns a number < 0 if this < o,
                        0 if this.equals(o),
                      > 0 if this > o.]
  }
}
```

Observe that the method prototypes in a Java interface may be declared without the "abstract" keyword, because it would be redundant; a Java interface cannot contain a method implementation.

The distinction between abstract classes and Java interfaces exists because of technical reasons that you might begin to understand if you take CS 164 (Compilers). Some languages, like C++, allow "multiple inheritance," so that a subclass can inherit from several superclasses. Java does not allow multiple inheritance in its full generality, but it offers a sort of crippled form of multiple inheritance: a class can "implement" multiple Java interfaces.

Why does Java have this limitation? Multiple inheritance introduces a lot of problems in both the definition of a language and the efficient implementation of a language. For example, what should we do if a class inherits from two different superclasses two different methods or fields with the same name? Multiple inheritance is responsible for some of the scariest tricks and traps of the C++ language, subtleties that cause much wailing and gnashing of teeth. Java interfaces don't have these problems.

Because an SList is a Nukeable and a Comparable, we can assign it to variables of these types.

```
Nukeable n = new SList();
Comparable c = (Comparable) n;
```

The cast is required because not every Nukeable is a Comparable.

"Comparable" is a standard interface in the Java library. By having a class implement Comparable, you immediately gain access to Java's sorting library. For instance, the Arrays class in java.util includes a method that sorts arrays of Comparable objects.

```
public static void sort(Object[] a)       // In java.util
```

The parameter's type is Object[], but a run-time error will occur if any item stored in a is not a Comparable.

Interfaces can be extended with subinterfaces. A subinterface can have multiple superinterfaces, so we can group several interfaces into one.

```
public interface NukeAndCompare extends Nukeable, Comparable { }
```

We could also add more method prototypes and constants, but in this example I don't.

CS 61B: Lecture 13
Wednesday, February 19, 2014

Today's reading: Sierra & Bates, pp. 154-160, 587-591, 667-668.

JAVA PACKAGES

In Java, a _package_ is a collection of classes and Java interfaces, and possibly subpackages, that trust each other. Packages have three benefits.

(1) Packages can contain hidden classes that are used by the package but are not visible or accessible outside the package.
(2) Classes in packages can have fields and methods that are visible by all classes inside the package, but not outside.
(3) Different packages can have classes with the same name. For example, java.awt.Frame and photo.Frame.

Here are two examples of packages.

(1) java.io is a package of I/O-related classes in the standard Java libraries.
(2) Homework 4 uses "list", a package containing the classes DList and DListNode. You will be adding two additional classes to the list package.

Package names are hierarchical. java.awt.image.Model refers to the class Model inside the package image inside the package awt inside the package java.

Using Packages

You can address any class, field, or method with a fully-qualified name. Here's an example of all three in one.

```
java.lang.System.out.println("My fingers are tired.");
```

Java's "import" command saves us from the tedium of using fully-qualified names all the time.

```
import java.io.File;  // Can now refer to File class, not just java.io.File.
import java.io.*;     // Can now refer to everything in java.io.
```

Every Java program implicitly imports java.lang.*, so you don't have to import it explicitly to use System.out.println(). However, if you import packages that contain multiple classes with the same name, you'll need to qualify their names explicitly throughout your code.

```
java.awt.Frame.add(photo.Frame.canvas);
```

Any package you create must appear in a directory of the same name. For example, the photo.Frame class bytecode appears in photo/Frame.class, and x.y.z.Class appears in x/y/z/Class.class. Where are the photo and x directories? They can appear in any of the directories on your "classpath". You can specify a classpath on the command line, as when you type

```
javac -cp ".:~jrs/classes:libraries.jar" *.java
```

This means that Java first looks in ".", the current directory, then looks in ~jrs/classes/, then finally in the _Java_archive_ libraries.jar when it's looking for the photo and x directories. The classpath does not include the location of the Java standard library packages (those beginning with java or javax). The Java compiler knows where to find them.

Building Packages

The files that form a package are annotated with a "package" command, which specifies the name of the package, which must match the name of the directory in which the files appear.

```
/* list/SList.java */                |  /* list/SListNode.java */
                                     |
package list;                        |  package list;
                                     |
public class SList {                 |  class SListNode {
  SListNode head;                    |    Object item;
  int size;                          |    SListNode next;
}                                    |  }
```

Here, the SListNode class and its fields are marked neither public, private, nor protected. Instead, they have "package" protection, which falls somewhere between "private" and "protected". Package protection is specified not by using the word "package", but by using no modifier at all. Variables are package by default unless declared public, private, or protected.

A class or variable with package protection is visible to any class in the same package, but not to classes outside the package (i.e., files outside the directory). The files in a package are presumed to trust each other, and are usually implemented by the same person. Files outside the package can only see the public classes, methods, and fields. (Subclasses outside the package can see the protected methods and fields as well.)

Before we knew about packages, we had to make the fields of SListNode public so that SList could manipulate them. Our list package above solves this problem by giving SListNode and its fields package protection, so that the SList class may use SListNodes freely, but outside applications cannot access them.

In Homework 4, you'll see a different approach. There, the DListNode class is public, so that DListNodes can be directly held by application programs, but the "prev" and "next" fields have package protection, so an application cannot access these fields or corrupt the DList ADT. But an application can hop quickly from node to node because it can store DListNode references and use them as parameters in DList method calls.

Each public class must be declared in a file named after the class, but a class with package protection can be declared in any .java file (usually found together with a class that uses it). So a public SList class and a package SListNode class can both be declared in the file list/SList.java, if you feel like it.

Compiling and running files in a package is a bit tricky, because it must be done from outside the package, using the following syntax:

```
javac -g list/SList.java
java list.SList
```

Here's the correspondence between declarations and their visibility.

Visible: / Declaration	in the same package	in a subclass	everywhere
"public"	X	X	X
"protected"	X	X	
default (package) "private"	X		

ITERATORS
=========
In java.util there is a standard Java interface for iterating over sequences of objects.

```
  public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();                          // The remove() method is optional.
  }
```

Part of Project 1 is to write a class RunIterator that implements an Iterator for your RunLengthEncoding class. Its purpose is to provide an interface by which other classes can read the runs in your run-length encoding, one by one.

An Iterator is like a bookmark. Just as you can have many bookmarks in a book, you can have many Iterators iterating over the same data structure, each one independent of the others. One Iterator can advance without disturbing other Iterators that are iterating over the same data structure.

The first time next() is called on a newly constructed Iterator, it returns the first item in the sequence. Each subsequent time next() is called, it returns the next item in the sequence. After the Iterator has returned every item in the sequence, every subsequent call to next() throws an exception and halts with an error message. (I find this annoying; I would prefer an interface in which next() returns null. The Java library designers disagree.)

To help you avoid triggering an exception, hasNext() returns true if the Iterator has more items to return, or false if it has already returned every item in the sequence. It is usually considered good practice to check hasNext() before calling next(). (In the next lecture we'll learn how to catch exceptions; that will give us an alternative way to prevent our program from crashing when next() throws an exception.)

There is usually no way to reset an Iterator back to the beginning of the sequence. Instead, you construct a new Iterator.

Most data structures that support Iterators "implement" another interface in java.util called "Iterable".

```
  public interface Iterable {
    Iterator iterator();
  }
```

It is customary for applications that want to iterate over a data structure DS to call DS.iterate(), which constructs and returns a DSIterator whose fields are initialized so it is ready to return the first item in DS.

A benefit of creating an Iterable class with its own Iterator is that Java has a simple built-in loop syntax, a second kind of "for each" loop, that iterates over the items in a data structure. Suppose we design an SList that implements Iterator. The following loop (which can appear in any class) iterates through the items in an SList l.

```
  for (Object o : l) {
    System.out.println(o);
  }
```

This loop is equivalent to

```
  for (Iterator i = l.iterator(); i.hasNext(); ) {
    Object o = i.next();
    System.out.println(o);
  }
```

To make all this more concrete, here is a complete implementation of an SListIterator class and a partial implementation of SList, both in the "list" package.

```
/* list/SListIterator.java */

package list;
import java.util.*;

public class SListIterator implements Iterator {
  SListNode n;

  public SListIterator(SList l) {
    n = l.head;
  }

  public boolean hasNext() {
    return n != null;
  }

  public Object next() {
    if (n == null) {
      /* We'll learn about throwing exceptions in the next lecture. */
      throw new NoSuchElementException();                       // In java.util
    }
    Object i = n.item;
    n = n.next;
    return i;
  }

  public void remove() {
    /* Doing it the lazy way.  Remove this, motherf! */
    throw new UnsupportedOperationException("Nice try, bozo."); // In java.lang
  }
}

/* list/SList.java */

package list;
import java.util.*;

public class SList implements Iterable {
  SListNode head;
  int size;

  public Iterator iterator() {
    return new SListIterator(this);
  }

  [other methods here]
}
```

Observe that an Iterator may mess up or even crash the program if the structure it is iterating over changes. For example, if the node "n" that an SListIterator references is removed from the list, the SListIterator will not be able to find the rest of the nodes.

An Iterator doesn't have to iterate over a data structure. For example, you can implement an Iterator subclass called Primes that returns each successive prime number as an Integer object.

CS 61B: Lecture 15
Wednesday, February 26, 2014

Today's reading: Sierra & Bates, pp. 315-338.

EXCEPTIONS

When a run-time error occurs in Java, the JVM "throws an exception," prints an error message, and quits. Oddly, an exception is a Java object (named Exception), and you can prevent the error message from printing and the program from terminating by "catching" the Exception that Java threw.

Purpose #1: Coping with Errors

Exceptions are a way of coping with unexpected errors. By catching exceptions, you can recover. For instance, if you try to open a file that doesn't exist or that you aren't allowed to read, Java will throw an exception. You can catch the exception, handle it, and continue, instead of letting the program crash.

```
  try {
    f = new FileInputStream("~cs61b/pj2.solution");
    i = f.read();
  }
  catch (FileNotFoundException e1) {
    System.out.println(e1);                    // An exception handler.
  }
  catch (IOException e2) {
    f.close();                                 // Another exception handler.
  }
```

What does this code do?
(a) It executes the code inside the "try" braces.
(b) If the "try" code executes normally, we skip over the "catch" clauses.
(c) If the "try" code throws an exception, Java does not finish the "try" code. It jumps directly to the first "catch" clause that matches the exception, and executes that "catch" clause. By "matches", I mean that the actual exception object thrown is the same class as, or a subclass of, the static type listed in the "catch" clause.

When the "catch" clause finishes executing, Java jumps to the next line of code immediately after all the "catch" clauses.

The code within a "catch" clause is called an _exception_handler_.

If the FileInputStream constructor fails to find the file, it will throw a FileNotFoundException. The line "i = f.read()" is not executed; execution jumps directly to the first exception handler.

FileNotFoundException is a subclass of IOException, so the exception matches both "catch" clauses. However, only one "catch" clause is executed--the first one that matches. The second "catch" clause would execute if the first were not present.

If the FileInputStream constructor runs without error, but the read() method throws an exception (for instance, because a disk track is faulty), it typically generates some sort of IOException that isn't a FileNotFoundException. This causes the second "catch" clause to execute and close the file. Exception handlers are often used to recover from errors and clean up loose ends like open files.

Note that you don't need a "catch" clause for every exception that can occur. You can catch some exceptions and let others propagate.

Purpose #2: Escaping a Sinking Ship

Believe it or not, you might want to throw your own exception. Exceptions are the easiest way to move program execution out of a method whose purpose has been defeated.

For example, suppose you're writing a parser that reads Java code and analyzes its syntactic structure. Parsers are quite complicated, and use many recursive calls and loops. Suppose that your parser is executing a method many methods deep within the program stack within many levels of loop nesting. Suddenly, your parser unexpectedly reaches the end of the file, because a student accidentally erased the last 50 lines of his program.

It's quite painful to write code that elegantly retraces its way back up through the method calls and loops when a surprise happens deep within a parser. A better solution? Throw an exception! You can even roll your own.

```
  public class ParserException extends Exception { }
```

This class doesn't have any methods except the default constructor. There's no need; the only purpose of a ParserException is to be distinguishable from other types of exceptions. Now we can write some parser methods.

```
  public ParseTree parseExpression() throws ParserException {
    [loops]
        if (somethingWrong) {
          throw new ParserException();
        }
        [more code]
      }
    }
    return pt;
  }
```

The "throw" statement throws a ParserException, thereby immediately getting us out of the routine. How is this different from a "return" statement? First, we don't have to return anything. Second, an exception can propagate several stack frames down the stack, not just one, as we'll see shortly.

The method signature has the modifier "throws ParserException". This is necessary; Java won't let you compile the method without it. "throws" clauses help you and the compiler keep track of which exceptions can propagate where.

```
  public ParseTree parse() throws ParserException, DumbCodeException {
    [loops and code]
        p = parseExpression();
        [more code]
      }
    }
  }

  public void compile() {
    ParseTree p;
    try {
      p = parse();
      p.toByteCode();
    }
    catch (ParserException e1) { }
    catch (DumbCodeException e2) { }
  }
```

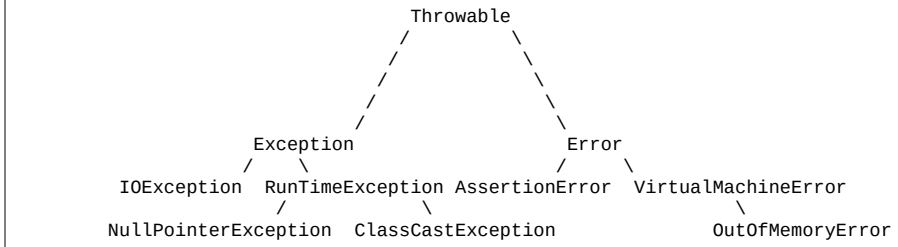
The parse() method above shows how to define a method that can throw two (or more) exceptions. Since every exception is a subclass of Exception, we could have replaced the two exceptions with "Exception", but then the caller would have to catch all types of Exceptions. We don't want (in this case) to catch NullPointerExceptions or otherwise hide our bugs from ourselves.

When parseExpression() throws an exception, it propagates right through the calling method parse() and down to compile(), where it is caught. compile() doesn't need a "throws ParserException" clause because it catches any ParserException that can occur. In this code, the "catch" clauses don't do anything except stop the exceptions.

If an exception propagates all the way out of main() without being caught, the JVM prints an error message and halts. You've seen this happen many times.

Checked and Unchecked Throwables

The top-level class of things you can "throw" and "catch" is called Throwable. Here's part of the Throwable class hierarchy.



An Error generally represents a fatal error, like running out of memory or stack space. Failed "assert" statements also generate a subclass of Error called an AssertionError. Although you can throw or catch any kind of Throwable, catching an Error is rarely appropriate.

Most Exceptions, unlike Errors, signify problems you could conceivably recover from. The subclass RunTimeException is made up of exceptions that might be thrown by the Java Virtual Machine, such as NullPointerException, ArrayIndexOutOfBoundsException, and ClassCastException.

There are two types of Throwables. _Unchecked_ Throwables are those a method can throw without declaring them in a "throws" clause. All Errors and RunTimeExceptions (including all their subclasses) are unchecked, because almost every method can generate them inadvertently, and it would be silly if we had to declare them.

All Exceptions except RunTimeExceptions are _checked_, which means that if your method might throw one without catching it, it must declare that possibility in a "throws" clause. Examples of checked exceptions include IOException and almost any Throwable subclass you would make yourself.

When a method calls another method that can throw a checked exception, it has just two choices.

(1) It can catch the exception, or
(2) it must be declared so that it "throws" the same exception itself.

The easiest way to figure out which exceptions to declare is to declare none and let the compiler's error messages tell you. (This won't work on the exams, though.)

CS 61B: Lecture 16
Wednesday, February 26, 2014

Today's reading: Sierra & Bates, pp. 189, 283.

EXCEPTIONS (continued)

The "finally" keyword

A finally clause can also be added to a "try."

```
  FileInputStream f = new FileInputStream("filename");
  try {
    statementX;
    return 1;
  } catch (IOException e) {
    e.printStackTrace();
    return 2;
  } finally {
    f.close();
  }
```

If the "try" statement begins to execute, the "finally" clause will be executed at the end, no matter what happens. "finally" clauses are used to do things that need to be done in both normal and exceptional circumstances. In this example, it is used to close a file.

If statementX causes no exception, then the "finally" clause is executed, and 1 is returned.

If statementX causes a IOException, the exception is caught, the "catch" clause is executed, and then the "finally" clause is executed. After the "finally" clause is done, 2 is returned.

If statementX causes some other class of exception, the "finally" clause is executed immediately, then the exception continues to propagate down the stack.

In the example above, we've invoked the method "printStackTrace" on the exception we caught. When an exception is constructed, it takes a snapshot of the stack, which can be printed later.

It is possible for an exception to occur in a "catch" or "finally" clause. An exception thrown in a "catch" clause will terminate the "catch" clause, but the "finally" clause will still get executed before the exception goes on. An exception thrown in a "finally" clause replaces the old exception, and terminates the "finally" clause and the method immediately.

However...you can nest a "try" clause inside a "catch" or "finally" clause, thereby catching those exceptions as well.

Exception constructors

By convention, most Throwables (including Exceptions) have two constructors. One takes no parameters, and one takes an error message in the form of a String.

```
  class MyException extends Exception {
    public MyException() { super(); }
    public MyException(String s) { super(s); }
  }
```

The error message will be printed if it propagates out of main(), and it can be read by the Throwable.getMessage() method. The constructors usually call the superclass constructors, which are defined in Throwable.

GENERICS

Suppose you're using a list of Objects to store Strings. When you fetch a String from the list, you have to cast it back to type "String" before you can call the methods exclusive to Strings. If somehow an object that's not a String got into your list, the cast will throw an exception. It would be nice to have the compiler enforce the restriction that nothing but Strings can ever get into your list in the first place, so you can sleep at night knowing that your family is safe from a ClassCastException.

So Java offers _generics_, which allow you to declare general classes that produce specialized objects. For example, you can create an SList for Strings only, and another SList for Integers only, even though you only wrote one SList class. To specify the class, SList takes a _type_parameter_.

```
class SListNode<T> {                          // T is the formal parameter.
  T item;
  SListNode<T> next;

  SListNode(T i, SListNode<T> n) {
    item = i;
    next = n;
  }
}

public class SList<T> {
  SListNode<T> head;

  public void insertFront(T item) {
    head = new SListNode<T>(item, head);
  }
}
```

You can now create and use an SList of Strings as follows.

```
  SList<String> l = new SList<String>();     // String is the actual parameter.
  l.insertFront("Hello");
```

Likewise, you can create an SList of Integers by using "SList<Integer>" in the declaration and constructor.

What are the advantages of generics? First, the compiler will ensure at compile-time that nothing but Strings can ever enter your SList<String>. Second, you don't have to cast the Objects coming out of your SList back to Strings, so there is no chance of an unexpected ClassCastException at run time. If some bug in your program is trying to put Integer objects into your SList, it's much easier to diagnose the compiler refusing to put an Integer into an SList<String> than it is to diagnose a ClassCastException occurring when you remove an Integer from a regular SList and try to cast it to String.

Generics are a complicated subject. Consider this to be a taste of them; hardly a thorough treatment. A good tutorial is available at https://www.seas.upenn.edu/~cis1xx/resources/generics-tutorial.pdf .

Although Java generics are superficially similar to C++ templates, there's a crucial difference between them. In the example above, Java compiles bytecode for only a single SList class. This SList bytecode can be used by all different object types. It is the compiler, not the bytecode itself, that enforces the fact that a particular SList object can only store objects of a particular class. Conversely, C++ recompiles the SList methods for every type that you instantiate SLists on. The C++ disadvantage is that one class might turn into a lot of machine code. The C++ advantages are that you can use primitive types, and you get code optimized for each type. Java generics don't work with primitive types.

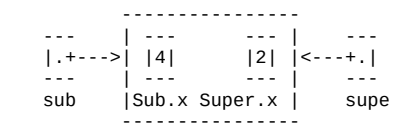
FIELD SHADOWING

Just as methods can be overridden in subclasses, fields can be "shadowed" in subclasses. However, shadowing works quite differently from overriding. Whereas the choice of methods is dictated by the _dyanamic_type_ of an object, the choice of fields is dictated by the _static_type_ of a variable or object.

```
  class Super {
    int x = 2;
    int f() {
      return 2;
    }
  }

  class Sub extends Super {
    int x = 4;                // shadows Super.x
    int f() {                 // overrides Super.f()
      return 4;
    }
  }
```

Any object of class Sub now has _two_ fields called x, each of which store a different integer. How do we know which field is accessed when we refer to x? It depends on the static type of the expression whose x field is accessed.

```
  Sub sub = new Sub();
  Super supe = sub;       // supe and sub reference the same object.
  int i;
```




```
  i = supe.x;             // 2
  i = sub.x;              // 4
  i = ((Super) sub).x;    // 2
  i = ((Sub) supe).x;     // 4
```

The last four statements all use the same object, but yield different results. Recall that method overriding does not work the same way. Since both variables reference a Sub, the method Sub.f always overrides Super.f.

```
  i = supe.f();           // 4
  i = sub.f();            // 4
  i = ((Super) sub).f();  // 4
  i = ((Sub) supe).f();   // 4
```

What if the variable whose shadowed field you want to access is "this"? You can cast "this" too, but a simpler alternative is to replace "this" with "super".

```
  class Sub extends Super {
    int x = 4;                // shadows Super.x
    void g() {
      int i;

      i = this.x;           // 4
      i = ((Super) this).x  // 2
      i = super.x;          // 2
    }
  }
```

Whereas method overriding is a powerful benefit of object orientation, field shadowing is largely a nuisance. Whenever possible, avoid having fields in subclasses whose names are the same as fields in their superclasses.

Static methods can be shadowed too; they follow the same shadowing rules as fields. This might seem confusing: why do ordinary, non-static methods use one system (overriding) while static methods use an entirely different system (shadowing)? The reason is because overriding requires dynamic method lookup. Dynamic method lookup looks up the dynamic type of an object. A static method is not called on an object, so there's nothing whose dynamic type we can look up. Therefore, static methods _can't_ use dynamic method lookup or overriding. So they use shadowing instead.

Static method shadowing, like field shadowing, is largely a nuisance.

"final" METHODS AND CLASSES

A method can be declared "final" to prevent subclasses from overriding it. Any attempt to override it will cause a compile-time error.

A class can be declared "final" to prevent it from being extended. Any attempt to declare a subclass will cause a compile-time error.

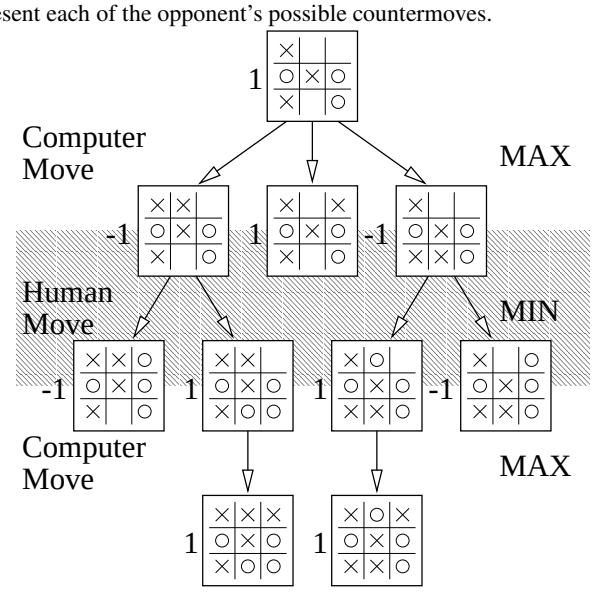
The only reason to declare a method or class "final" is to improve the speed of a program. The compiler can speed up method calls that cannot be overridden.

CS 61B: Lecture 17
Monday, March 3, 2014

Game tree search

How could we design a program that plays Tic Tac Toe? The standard technique searches for the best moves by using a *game tree*, which looks a bit like a family tree. A game tree is *not* a data structure; it is the structure of a sequence of recursive method calls.

The ancestor at the top of the tree is the current grid, accompanied by a record of whose turn it is. The children of the ancestor form the set of possible grids that follow from considering each legal move. These children have children of their own, which represent each of the opponent's possible countermoves.



If we assume that both opponents are infinitely intelligent and will always choose the best move, then we can determine the computer's best move using the game tree. Each legal grid (coupled with information about whose turn it is) is assigned a numerical score that indicates how optimistic we are about winning. For concreteness, give a grid a score of 1 if the computer is guaranteed a win (assuming it plays perfectly), a score of -1 if the computer's opponent is guaranteed a win, and a score of 0 if perfect players will draw. (It's wise to give these constants names like `COMPUTER_WIN`, `HUMAN_WIN`, and `DRAW`.)

How do we score a given grid? It's easy if the game is over. Suppose the computer is playing X. If there are three X's in a row, assign the grid a 1; for three collinear O's, assign it a -1. If the grid has no empty squares left, and nobody has won, assign it a 0.

The score of any other grid is computed by a *minimax* algorithm. We consider each possible move, and determine the child grid that each move creates. We assign a score to each child grid by calling the minimax algorithm recursively. Then we assign a score to the current (parent) grid. If it's the computer's turn, we choose the move that yields the maximum score, and assign the same score to the current grid. If it's the opponent's turn, we assume that the opponent plays perfectly. So we choose the move that yields the minimum score, and assign that score to the current grid.

Minimax is recursive, and the figure above illustrates the recursive method calls that minimax executes when evaluating several grids' scores. Again, no tree data structure is created! At any one point in time, the program stack represents one path down the tree (but the root of the tree is at the bottom of the stack). Each grid's score is printed to its left. Because the top-level grid has a score of one, the computer is guaranteed a win, which is obtained by choosing the second of the three possible moves.

The following pseudocode computes a grid's score and the best move from that grid (which determines the grid's score). A `Best` object holds a record of the best `move` and its `score`.

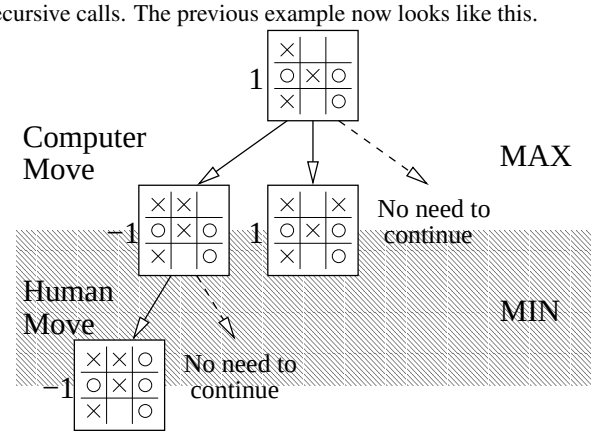
```
public class Grid {
  public Best chooseMove(boolean side) {
    Best myBest = new Best();      // My best move
    Best reply;           // Opponent's best reply

    if ("this" Grid is full or has a win) {
      return a Best with Grid's score, no move;
    }
    if (side == COMPUTER) {
      myBest.score = -1;
    } else {
      myBest.score = 1;
    }
    myBest.move = any legal move;
    for (each legal move m) {
      perform move m;      // Modifies "this" Grid
      reply = chooseMove(! side);
      undo move m;         // Restores "this" Grid
      if ((side == COMPUTER &&
           reply.score > myBest.score) ||
          (side == HUMAN &&
           reply.score < myBest.score)) {
        myBest.move = m;
        myBest.score = reply.score;
      }
    }
    return myBest;
  }
}
```

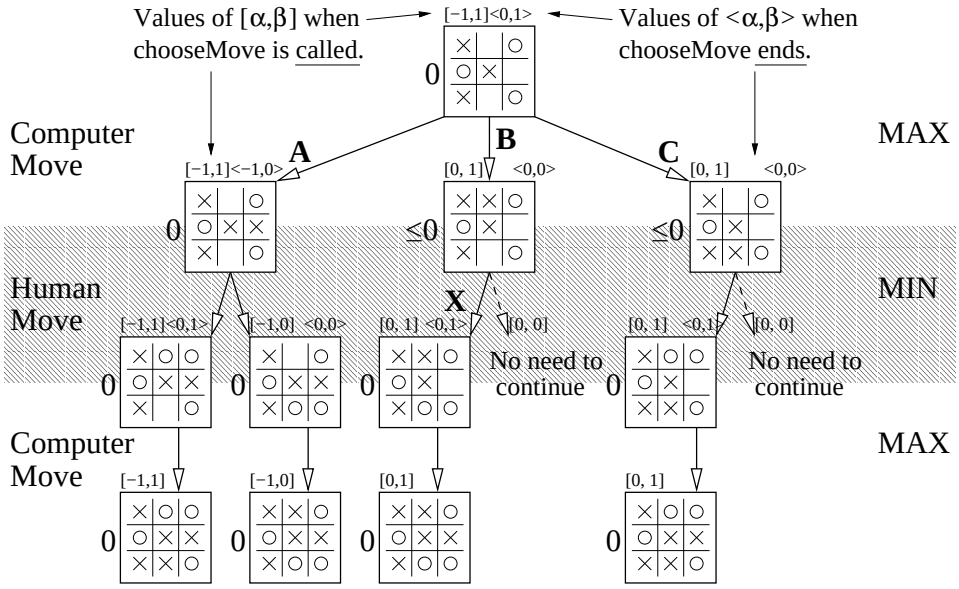
Observe that `myBest.move` is initialized to an arbitrary legal move. This is necessary so that if *every* move leads to a guaranteed loss, some move is returned nevertheless.

Each grid in the game tree at left represents one invocation of `chooseMove`. The children of each grid represent recursive calls by the parent; these child grids are processed in order from left to right. When any particular grid is invoked, that grid and its ancestors (parent, grandparent, etc.) are stack frames on the program stack.

Simple pruning

If, at some grid in the game tree, a player discovers a guaranteed winning move, there's no reason to continue to search for a better move from that grid. We can save time by pruning away some recursive calls. The previous example now looks like this.





Alpha-beta pruning

A more aggressive pruning technique (which subsumes simple pruning) is based on the following observation. (See the figure above.) Suppose the computer has discovered a move (A) that guarantees a draw, and is investigating an alternative move (B). The computer begins to consider all the moves its opponent could make from grid B. It discovers that if the opponent makes move X in response, the opponent can force a draw.

It would be a waste of the computer's time to continue investigating moves the opponent could make from grid B. Since the opponent can, at the very least, force a draw, move B is no better for the computer than move A—though it might be worse. The computer should simply forget move B and consider move C.

To turn this insight into an algorithm, we pass two additional parameters to the `chooseMove()` method: α and β. The parameter α is a score that the computer knows with certainty it can achieve; for instance, if $\alpha = 0$, then the computer knows it can force a draw, and is only interested in searching for moves guaranteed to do better. Conversely, β is a guarantee that the opponent can achieve a score of β or lower. For any grid, we maintain values of α and β based on our current knowledge of the best moves discovered thus far. If β becomes equal to or less than α, then further investigation of the current grid is useless.

In the figure above, for instance, grid A has a score of 0. The top grid is a MAX grid, so the computer cannot score lower than zero (worse than a draw), and it sets $\alpha = 0$ at the topmost grid and uses the parameters $[\alpha, \beta] = [0, 1]$ to begin investigating grid B.

Minimax computes that Grid X has a score of 0. We do not know the exact score of B, but we know it is less than or equal to zero, because B is a "MIN" grid. In other words, the computer's opponent can force a draw from grid B. Therefore, move B is no better than the best known alternative move (A), and it might be worse. There is no point to investigating the other children of grid B. Alpha-beta search acknowledges that B's score ≤ 0 by setting $\beta = 0$ for grid B. The fact that the parameters $[\alpha, \beta] = [0, 0]$ have met in the middle cues the algorithm to prune B's remaining child and go on to move C.

For the same reason, alpha-beta search prunes C's second child.

The following pseudocode formalizes game tree search with alpha-beta pruning. Note that α only changes during a computer (MAX) move, and β only changes during an opponent (MIN) move. The top-level chooseMove invocation should be called with $[\alpha, \beta] = [-1, 1]$.

```
public Best chooseMove(boolean side,
                       int alpha, int beta) {
  Best myBest = new Best();       // My best move
  Best reply;            // Opponent's best reply

  if ("this" Grid is full or has a win) {
    return a Best with the Grid's score, no move;
  }
  if (side == COMPUTER) {
    myBest.score = alpha;
  } else {
    myBest.score = beta;
  }
  myBest.move = any legal move;
  for (each legal move m) {
    perform move m;       // Modifies "this" Grid
    reply = chooseMove(! side, alpha, beta);
    undo move m;          // Restores "this" Grid
    if (side == COMPUTER &&
        reply.score > myBest.score) {
      myBest.move = m;
      myBest.score = reply.score;
      alpha = reply.score;
    } else if (side == HUMAN &&
               reply.score < myBest.score) {
      myBest.move = m;
      myBest.score = reply.score;
      beta = reply.score;
    }
    if (alpha >= beta) { return myBest; }
  }
  return myBest;
}
```

Combinatorially huge game trees

Game trees grow exponentially with tree depth. Even with every technique at our disposal, we cannot hope to explore the entire tree for a game of chess. Hence, game tree search typically limits its recursion to a specified depth. Positions at the maximum depth are evaluated not by recursive search, but by less accurate heuristics called *evaluation functions*. Evaluation functions compute numerical estimates of the computer's optimism. These scores are not just -1, 0, or 1, but can take on a continuous range between (for example) -1.0 and 1.0. Even with this infinite-valued (rather than three-valued) scoring system, alpha-beta search still works correctly; in fact, it is at its best in such circumstances.

CS 61B: Lecture 18
Wednesday, March 5, 2014

Today's reading: Sierra & Bates, pp. 80-84.

ENCAPSULATION

A _module_ is a set of methods that work together as a whole to perform some task or set of related tasks. A module is _encapsulated_ if its implementation is completely hidden, and it can be accessed only through a documented interface.

As you know, an abstract data type (ADT) is an encapsulated data structure. Not all encapsulated modules are ADTs, though. Algorithms (like list sorters) and applications (like network routing software) can also be encapsulated, even if they are distinct from the data structures they use.

So far, I've discussed encapsulation as a way of preventing "evil tamperers" from corrupting your data structures. Who are these evil tamperers? Sometimes, they're your coworkers, or other programmers who will work on a project long after you're gone. Often the evil tamperer is you.

A Cautionary Tale

Doug Whole, a programmer at a Silicon Valley startup, implements a singly-linked list much like the one you used in Homework 3, but all its fields are public. Doug also writes application code that uses linked lists. One day, Doug needs to write code that splices the second node out of a list. It would only take one line, and he doesn't foresee ever needing to use the same operation anywhere else. Being lazy, Doug doesn't feel like adding a new method to the List class. Instead, he just does the work directly.

```
public class ListMangler {
  [lotsa code]

  /* Gosh, I am sooooooooooooooooooo tired. */
  list.head.next = list.head.next.next;

  [lotsa more code]
}
```

Two years later, another programmer, Jeannie Yess, decides to improve the speed of their list data structure. After careful thought, she decides to reprogram the List class so that it uses doubly-linked lists internally. A "previous" field is added to ListNode, and the List methods are rewritten.

Jeannie tests her new List implementation extensively, and can find no bugs. But when she replaces Doug's List class with her own, the company's landmark ListMangler application repeatedly produces the wrong results. After two long days of debugging, Jeannie discovers the culprit: Doug's single line of code.

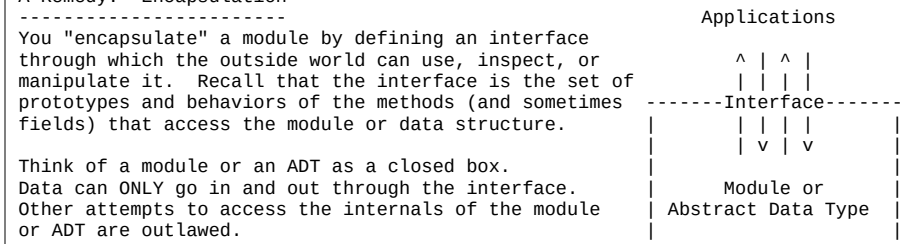
This kind of bug is one of the most difficult to find and fix. It's also very common in commercial software systems, and it can have far-reaching effects.

You see, Doug's line of code is not the only one that reads or modifies the list data structure directly. Jeannie still has to debug 100,000 lines of Doug's code in other failing applications, as well as 500,000 lines more written by other programmers who also directly manipulated ListNodes. The List improvement project is abandoned.

A Remedy: Encapsulation

You "encapsulate" a module by defining an interface through which the outside world can use, inspect, or manipulate it. Recall that the interface is the set of prototypes and behaviors of the methods (and sometimes fields) that access the module or data structure.

Think of a module or an ADT as a closed box. Data can ONLY go in and out through the interface. Other attempts to access the internals of the module or ADT are outlawed.



Why encapsulation is your friend:

[1] The implementation is independent of the functionality. A programmer who has the documentation of the interface can implement a new version of the module or ADT independently. A new, better implementation can replace an old one.

[2] Encapsulation prevents Doug from writing applications that corrupt a module's internal data. In real-world programming, encapsulation reduces debugging time. A lot.

[3] ADTs can guarantee that their invariants are preserved.

[4] Teamwork. Once you've rigorously defined interfaces between modules, each programmer can independently implement a module without having access to the other modules. A large, complex programming project can be broken up into dozens of pieces.

[5] Documentation and maintainability. By defining an unambiguous interface, you make it easier for other programmers to fix bugs that arise years after you've left the company. Many bugs are a result of unforeseen interactions between modules. If there's a clear specification of each interface and each module's behavior, bugs are easier to trace.

[6] When your Project 2 doesn't work, it will be easier to figure out which teammate to blame.

An interface is a CONTRACT between module writers, specifying exactly how they will communicate.

Enforcing Encapsulation

Many languages offer only one construct for enforcing the encapsulation of ADTs: self-discipline.

As we've seen, Java offers facilities that fortify your self-discipline, especially Java packages and the "private", package, and "protected" modifiers for field and method declarations.

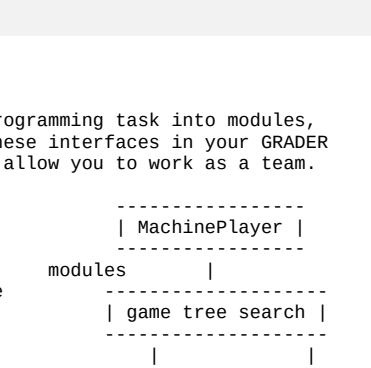
Java's facilities aren't always enough, though. There are circumstances in which you'll want to have multiple modules in the same package. For instance, in Project 2 it would be reasonable to put all your modules in the "player" package. If you do that, you'll have to fall back on self-discipline. This means defining your modules and interfaces before you start programming, and resisting the temptation to let one module snoop through or change another module's data structures.

One way to find this self-discipline is, wherever one module uses another, to have a different team member work on each module. If neither team member reveals their code to the other, it's much harder to yield to temptation.

Modules and Interfaces in Project 2

In Project 2, you are required to divide the programming task into modules, define interfaces between them, and document these interfaces in your GRADER file, before you start programming. This will allow you to work as a team.

The game-playing program you will write for Project 2 can easily be broken down into a number of modules. Four likely examples are illustrated at right. Your MachinePlayer, its game tree search (with alpha-beta pruning), the board evaluation function, and the module that identifies winning networks can all be implemented completely independently, even though they will ultimately work together.



You should probably break your MachinePlayer down into a few more modules than this (the project README gives a few more suggestions), but don't try to break it up too much. You will reach a point where it is no longer possible to subdivide any module into pieces that are independent and communicate through _simple_ interfaces.

You might still be confused: what exactly _is_ a module? It's a collection of methods that provide some functionality through a single (hopefully elegant) interface. The main difference between a module and a class is this:

- A _module_ is organized around the _functionality_ it provides.
- A _class_ is organized around a _data_storage_unit_. (Remember that an object is a repository of data.)

The concept of modules is a bit abstract for several reasons.

- A module can be made up of several classes, or a class could be made up of several modules. Module boundaries and class boundaries can be independent of each other.
  - o Why would a module have several classes? Because it might implement a data structure made up of several classes of objects. For example, a List ADT has a list object and node objects. A Graph ADT has a graph object, vertices, and edges.
  - o Why would a class have several modules? Because a single class of object might support many independent operations. The game tree search and the evaluation function above both operate on a Network game board, but they are independent enough of each other that you could change the implementation of one without changing the other. (Of course, if you change the way a game board is represented, you'll have to change both implementations.)
- A module may include many methods, or as few as one. (But not every method needs its own module!)
- A package may contain one module or many.

For Project 2, you should document your modules and interfaces as follows.

- List the modules.
- For each module, specify its interface.
  - o Recall that an interface includes the prototype(s) for the methods by which the module can be called. This list does not necessarily include all the methods in the module! It only includes the methods that are available for _external_ callers (outside the module).
  - o An interface also includes, for each prototype, a comment that describes precisely the module's behavior from an _external_ observer's point of view. Your description does not need to state how the module is implemented, though. For instance, a module that does game tree search should say that it returns a good, legal move, but it does not need to say that it does alpha-beta pruning. (It's not forbidden to say this, though.) Likewise, you should state that the "network identifier" determines whether a game board contains a winning network for a given player, but the interface does not need to specify what algorithm is used to look for winning networks. (A description of the algorithm should be included in the comments _in_ the implementation, but it is not part of the _interface_.)
  - o The behavior comment should also describe, for each prototype, every parameter and the return value (if any), and how they are interpreted. Here you are making a _contract_ that your module will speak a certain language when it communicates with external callers.

Here's a short example of an interface you might put in your GRADER file. (You are not required to implement it this way; this is just an example. Other modules will probably require longer behavioral descriptions.)

```
  /**
   *  hasValidNetwork() determines whether "this" GameBoard has a valid network
   *  for player "side".  (Does not check whether the opponent has a network.)
   *  A full description of what constitutes a valid network appears in the
   *  project "readme" file.
   *
   *  Unusual conditions:
   *    If side is neither MachinePlayer.COMPUTER nor MachinePlayer.OPPONENT,
   *          returns false.
   *    If GameBoard squares contain illegal values, the behavior of this
   *          method is undefined (i.e., don't expect any reasonable behavior).
   *
   *  @param side is MachinePlayer.COMPUTER or MachinePlayer.OPPONENT
   *  @return true if player "side" has a winning network in "this" GameBoard;
   *          false otherwise.
   **/
  protected boolean hasValidNetwork(int side)
```

Your description of how a module behaves should be complete and unambiguous, and should take into account unusual and erroneous inputs and circumstances. (It's sometimes okay if your module doesn't handle an erroneous input well, but you should document that.) When you and your partners are writing the interfaces for each module, think carefully about whether you believe these interfaces will really allow all the modules to do everything they need to do.

When you design your interfaces, they should appear (prototypes and behavioral descriptions both) in both the GRADER file and in the code itself. Once you've finished, decide which team members will implement which modules, and start programming.

You may find your team returning to modify the interfaces after a first attempt at programming, but that's okay. Just be sure to change the documentation (in both GRADER and the code comments) to reflect your new design decisions.

I recommend you write a draft of your interfaces this week so you'll have lots of time to program. The interfaces in the GRADER file are worth 10% of your project score. You will need to show them to your TA next week in Lab 8.

CS 61B: Lecture 19
Wednesday, March 5, 2014

Today's reading: Sierra & Bates, p. 664.

ENCAPSULATED LISTS (a case study in encapsulation)

Homeworks 3, 4, and 5 introduced you to three different implementations of linked lists, each fundamentally different.

With the Homework 3 lists, if an application writer wants to query the identity of every item in the list without modifying the list, it takes time proportional to the square of n, the number of items in the list (i.e., Theta(n^2) time), because you have to use nth(i) to identify each item in time proportional to i.

The lists in Homeworks 4 and 5 allow an application to directly hold a node in a list. By alternating between the next() method and the item field or method, you can query all the list's items in Theta(n) time. Similarly, if an application holds a node in the middle of a list, it can insert or delete c items there in time proportional to c, no matter how long the list is.

The Homework 5 lists (SList and DList) are well-encapsulated, whereas the Homework 4 DList has flaws. I will discuss these flaws today to illustrate why designing the really good list ADTs of Homework 5 was tricky. Let's ask some questions about how lists should behave.

(1) What happens if we invoke l.remove(n)--but the node n is in a different list than l?

In Homework 4, Part II asks whether it is possible for an application to break the DList invariants. One way to do this is to mismatch nodes and lists in method calls. When an application does this, the "size" field of the wrong list is updated, thereby breaking the invariant that a list's size field should be correct. How can we fix this?

ADT interface answer: The methods remove(), insertAfter(), etc. should always update the right list's "size" field.

Implementation answer: It's unacceptably slow to walk through a whole list just to see if the node n is really in the list l. Instead, every node should keep a reference to the list that contains it. In Homework 5, each ListNode has a "myList" field.

(2) Should insertAfter(), remove(), etc. be methods of List or ListNode?

Normally, we expect the methods that modify a data structure (like a List) to be methods within that data structure's class. However, if we define methods like insertAfter() and remove() in the ListNode class, rather than the List class, we completely avoid the question of what happens if they're invoked for a node that's not in "this" list. This way, the interface is more elegant.

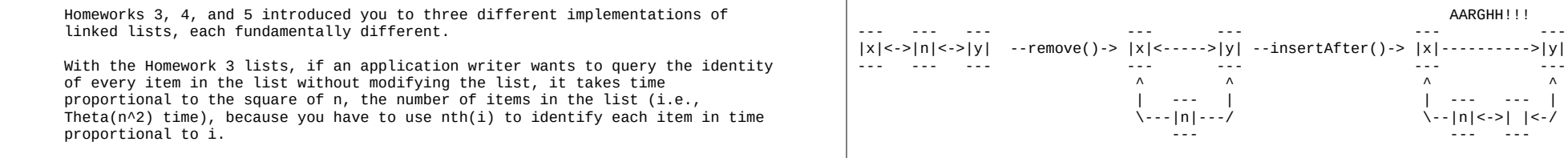
ADT interface answer: the list methods are divided among List and ListNode.

Some methods of List	Some methods of ListNode
public boolean isEmpty()	public Object item()
public void insertFront(Object item)	public ListNode next()
public ListNode front()	public void insertAfter(Object item)

Implementation answer: again, each node has a "myList" field so we can update a list's "size" field when we call n.remove(), n.insertAfter(), etc.

(3) What happens if we invoke l.remove(n), then l.insertAfter(i, n)?

Another way to trash the DList invariants is to treat a node that's been removed from a list as if it's still active. If we call insertAfter on a node we've already removed, we may mangle the pointers.



The result violates the invariant that if x.next == y, then y.prev == x. We would prevent the pointer mangling if remove(n) set n's pointers to null, but that wouldn't stop insertAfter() from incrementing the list's "size" field (or throwing a NullPointerException), which is not a reasonable result.

Calling remove(n) twice on the same node also corrupts "size".

How can we fix this?

ADT interface answer: After n.remove() is executed, removing n from the list, n is considered to be an "invalid" node. Any attempt to use n, except to call n.isValidNode(), throws an exception.

Why do we change the node, rather than erasing the reference to it? First, the remove() method can't erase the reference, which is passed by value. Second, there might be lots of other references to the same node, and we need to erase all of them too! All those other references could be used to corrupt the data structure if the node itself isn't neutralized.

Implementation answer: When an item is removed from a list, the corresponding ListNode's "myList" reference is set to null. This is just a convenient way to mark a node as "invalid". The "next" and "prev" references are also set to null. These steps eliminate opportunities for accidentally corrupting a list as illustrated above. (Also, they help Java's garbage collection to reclaim unused DListNodes. We'll discuss garbage collection near the end of the semester.)

Any ListNode whose "myList" reference is null is considered "invalid", and any attempt to use it will incite an exception.

(4) What happens if we walk off the end of a list? (Using the next() method.)

ADT interface answer: In Homework 4, if you invoke next() on the last node in a list, it returns null. In Homework 5, it returns an invalid node instead. There are two reasons for this change. First, it provides consistency, because invoking next() at the end of a list yields the same result as removing a node. Second, if you call a method on the result--for instance, n.next().item()--it throws an InvalidNodeException instead of a NullPointerException. This eliminates ambiguity; you can catch an InvalidNodeException without wondering why it was thrown, whereas many different bugs can cause NullPointerExceptions.

Implementation answer: Recall that our implementation uses a doubly-, circularly-linked list with a sentinel node. Any sentinel is considered an invalid node. This simplifies the implementations of the next() and prev() methods in the DList class.

However, if you apply next() to a sentinel, you won't get the first node of the list; you'll get an InvalidNodeException. Why? When n is the last node in a list, why not let n.next().next() be the first node? First, the fact that the implementation uses a sentinel should be completely hidden from the application. Second, we want to be able to change the implementation without breaking the application. Suppose we switch from DLists to SLists that don't have sentinels. We would need to "fix" SList so that n.next().next() still behaves the way it does with DLists. It's better not to allow applications to take advantage of such quirks from the start.

(5) How do we access an item?

ADT interface answer: In Homework 4, each node's "item" field is public. In Homework 5, we make the "item" field protected; applications must use the item() and setItem() methods to access it. Why? To make sure that applications can't store items in deleted nodes or sentinels. Any attempt to invoke item() or setItem() on an invalid node causes an exception. Why? So that the implementation can be changed without breaking an application. Suppose, for instance, that an application stores items in sentinel nodes. Would the application still work the same way if you switched from DLists to SLists, which don't have sentinel nodes?

This may seem like a strange justification. But in real-world programming, programmers often take advantage of undocumented quirks, like being able to store items in sentinel nodes. Once applications have been written that depend on these quirks, the quirks become "features" that must be preserved in any new List implementation. That's why ADTs should never do _more_ than what the documentation says they do.

In Frederick P. Brooks, Jr.'s famous book on software engineering, "The Mythical Man-Month" (page 65), he writes

> Invalid syntax always produces some result; in a policed system that result is an invalidity indication _and_nothing_more_. In an unpoliced system all kinds of side effects may appear, and these may have been used by programmers. When we undertook to emulate the IBM 1401 [processor] on System/360 [an operating system], for example, it developed that there were 30 different "curios"--side effects of supposedly invalid operations--that had come into widespread use and had to be considered as part of the definition. The implementation as a definition [of the functionality] overprescribed; it not only said what the machine must do, it also said a great deal about how it had to do it.

By ensuring that an implementation does not produce any result not specified in the interface--even for invalid inputs--a programmer makes it easy to fix bugs, optimize performance, and add new features without compromising existing applications.

This lecture's lesson is that design decisions can be complicated and have unexpected repercussions.

Our design decisions for the Homework 5 lists, described above, will carry over to our tree interfaces, which you'll encounter in an upcoming assignment.

One final thought. Why don't we simply keep a boolean "valid" flag in each ListNode, and use that to distinguish valid nodes from invalid ones? It would make the implementation clearer, and therefore more maintainable. However, it would also make each ListNode occupy more memory. I chose reduced memory use over readability, but this was an arbitrary choice.

CS 61B: Lecture 20
Monday, March 10, 2014

Today's reading: Goodrich & Tamassia, Chapter 4 (especially 4.2 and 4.3).

ASYMPTOTIC ANALYSIS (bounds on running time or memory)
==========
Suppose an algorithm for processing a retail store's inventory takes:
  - 10,000 milliseconds to read the initial inventory from disk, and then
  - 10 milliseconds to process each transaction (items acquired or sold).
Processing n transactions takes (10,000 + 10 n) ms. Even though 10,000 >> 10, we sense that the "10 n" term will be more important if the number of transactions is very large.

We also know that these coefficients will change if we buy a faster computer or disk drive, or use a different language or compiler. We want a way to express the speed of an algorithm independently of a specific implementation on a specific machine--specifically, we want to ignore constant factors (which get smaller and smaller as technology improves).

Big-Oh Notation (upper bounds on a function's growth)

Big-Oh notation compares how quickly two functions grow as n -> infinity.

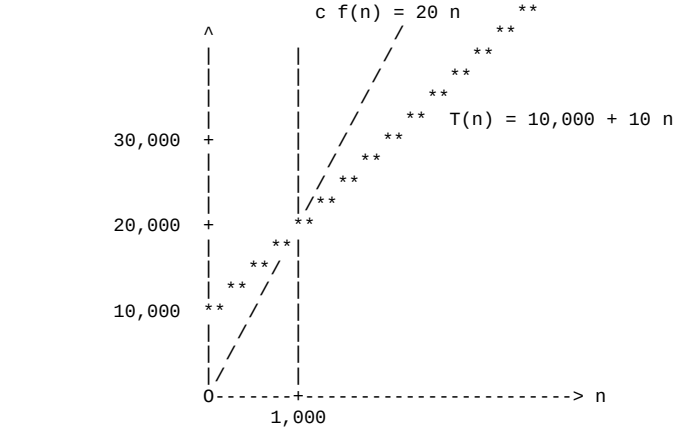
Let n be the size of a program's _input_ (in bits or data words or whatever).
Let T(n) be a function. For now, T(n) is the algorithm's precise running time in milliseconds, given an input of size n (usually a complicated expression).
Let f(n) be another function--preferably a simple function like f(n) = n.

We say that T(n) is in O(f(n)) IF AND ONLY IF T(n) <= c f(n)
WHENEVER n IS BIG, FOR SOME LARGE CONSTANT c.

* HOW BIG IS "BIG"? Big enough to make T(n) fit under c f(n).
* HOW LARGE IS c? Large enough to make T(n) fit under c f(n).

EXAMPLE: Inventory

Let's consider the function T(n) = 10,000 + 10 n, from our example above.
Let's try out f(n) = n, because it's simple. We can choose c as large as we want, and we're trying to make T(n) fit underneath c f(n), so pick c = 20.



As these functions extend forever to the right, their asymptotes will never cross again. For large n--any n bigger than 1,000, in fact--T(n) <= c f(n).
*** THEREFORE, T(n) is in O(f(n)). ***

Next, you must learn how to express this idea rigorously. Here is the central lesson of today's lecture, which will bear on your entire career as a professional computer scientist, however abstruse it may seem now:

```
|=============================================================================|
| FORMALLY:  O(f(n)) is the SET of ALL functions T(n) that satisfy:           |
|                                                                             |
|   There exist positive constants c and N such that, for all n >= N,         |
|                            T(n) <= c f(n)                                   |
|=============================================================================|
```

Pay close attention to c and N. In the graph above, c = 20, and N = 1,000.

Think of it this way: if you're trying to prove that one function is asymptotically bounded by another [f(n) is in O(g(n))], you're allowed to multiply them by positive constants in an attempt to stuff one underneath the other. You're also allowed to move the vertical line (N) as far to the right as you like (to get all the crossings onto the left side). We're only interested in how the functions behave as n shoots off toward infinity.

EXAMPLES: Some Important Corollaries
---
[1] 1,000,000 n is in O(n). Proof: set c = 1,000,000, N = 0.
  -> Therefore, Big-Oh notation doesn't care about (most) constant factors. We generally leave constants out; it's unnecessary to write O(2n), because O(2n) = O(n). (The 2 is not wrong; just unnecessary.)

[2] n is in O(n^3). [That's n cubed]. Proof: set c = 1, N = 1.
  -> Therefore, Big-Oh notation can be misleading. Just because an algorithm's running time is in O(n^3) doesn't mean it's slow; it might also be in O(n). Big-Oh notation only gives us an UPPER BOUND on a function.

[3] n^3 + n^2 + n is in O(n^3). Proof: set c = 3, N = 1.
  -> Big-Oh notation is usually used only to indicate the dominating (largest and most displeasing) term in the function. The other terms become insignificant when n is really big.

  Here's a table to help you figure out the dominating term.

Table of Important Big-Oh Sets

Arranged from smallest to largest, happiest to saddest, in order of increasing domination:

	function		common name
	O(1)	::	constant
is a subset of	O(log n)	::	logarithmic
is a subset of	O(log^2 n)	::	log-squared [that's (log n)^2]
is a subset of	O(root(n))	::	root-n [that's the square root]
is a subset of	O(n)	::	linear
is a subset of	O(n log n)	::	n log n
is a subset of	O(n^2)	::	quadratic
is a subset of	O(n^3)	::	cubic
is a subset of	O(n^4)	::	quartic
is a subset of	O(2^n)	::	exponential
is a subset of	O(e^n)	::	exponential (but more so)

Algorithms that run in O(n log n) time or faster are considered efficient. Algorithms that take n^7 time or more are usually considered useless. In the region between n log n and n^7, the usefulness of an algorithm depends on the typical input sizes and the associated constants hidden by the Big-Oh notation.

If you're not thoroughly comfortable with logarithms, read Sections 4.1.2 and 4.1.7 of Goodrich & Tamassia carefully. Computer scientists need to know logarithms in their bones.

Warnings
---

[1] Here's a fallacious proof:

n^2 is in O(n), because if we choose c = n, we get n^2 <= n^2.
-> WRONG! c must be a constant; it cannot depend on n.

[2] The big-Oh notation expresses a relationship between functions. IT DOES NOT SAY WHAT THE FUNCTIONS MEAN. In particular, the function on the left does not need to be the worst-case running time, though it often is. The number of emails you send to your Mom as a function of time might be in O(t^2). In that case, not only are you a very good child; you're an increasingly good child.

In binary search on an array,

- the worst-case running time is in O(log n),
- the best-case running time is in O(1),
- the memory use is in O(n), and
- 47 + 18 log n - 3/n is in O(the worst-case running time).

Every semester, a few students get the wrong idea that "big-Oh" always means "worst-case running time." Their brains short out when an exam question uses it some other way.

[3] "e^3n is in O(e^n) because constant factors don't matter."
"10^n is in O(2^n) because constant factors don't matter."
-> WRONG! I said that Big-Oh notation doesn't care about (most) constant factors. Here are some of the exceptions. A constant factor in an exponent is not the same as a constant factor in front of a term. e^3n is not bigger than e^n by a constant factor; it's bigger by a factor of e^2n, which is damn big. Likewise, 10^n is bigger than 2^n by a factor of 5^n.

[4] Big-Oh notation doesn't tell the whole story, because it leaves out the constants. If one algorithm runs in time T(n) = n log_2 n, and another algorithm runs in time U(n) = 100 n, then Big-Oh notation suggests you should use U(n), because T(n) dominates U(n) asymptotically. However, U(n) is only faster than T(n) in practice if your input size is greater than current estimates of the number of subatomic particles in the universe. The base-two logarithm log_2 n < 50 for any input size n you are ever likely to encounter.

Nevertheless, Big-Oh notation is still a good rule of thumb, because the hidden constants in real-world algorithms usually aren't that big.

CS 61B: Lecture 21
Wednesday, March 12, 2014

Today's reading: Goodrich & Tamassia, Sections 9.1, 9.2, 9.5-9.5.1.

DICTIONARIES
============
Suppose you have a set of two-letter words and their definitions. You want to be able to look up the definition of any word, very quickly. The two-letter word is the _key_ that addresses the definition.

Since there are 26 English letters, there are 26 * 26 = 676 possible two-letter words. To implement a dictionary, we declare an array of 676 references, all initially set to null. To insert a Definition into the dictionary, we define a function hashCode() that maps each two-letter word (key) to a unique integer between 0 and 675. We use this integer as an index into the array, and make the corresponding bucket (array position) point to the Definition object.

```
public class Word {
  public static final int LETTERS = 26, WORDS = LETTERS * LETTERS;
  public String word;

  public int hashCode() {                  // Map a two-letter Word to 0...675.
    return LETTERS * (word.charAt(0) - 'a') + (word.charAt(1) - 'a');
  }
}

public class WordDictionary {
  private Definition[] defTable = new Definition[Word.WORDS];

  public void insert(Word w, Definition d) {
    defTable[w.hashCode()] = d;               // Insert (w, d) into Dictionary.
  }

  Definition find(Word w) {
    return defTable[w.hashCode()];              // Return the Definition of w.
  }
}
```

What if we want to store every English word, not just the two-letter words? The table "defTable" must be long enough to accommodate pneumonoultramicroscopicsilicovolcanoconiosis, 45 letters long. Unfortunately, declaring an array of length 26^45 is out of the question. English has fewer than one million words, so we should be able to do better.

Hash Tables (the most common implementation of dictionaries)

Suppose n is the number of keys (words) whose definitions we want to store, and suppose we use a table of N buckets, where N is perhaps a bit larger than n, but much smaller than the number of _possible_ keys. A hash table maps a huge set of possible keys into N buckets by applying a _compression_function_ to each hash code. The obvious compression function is

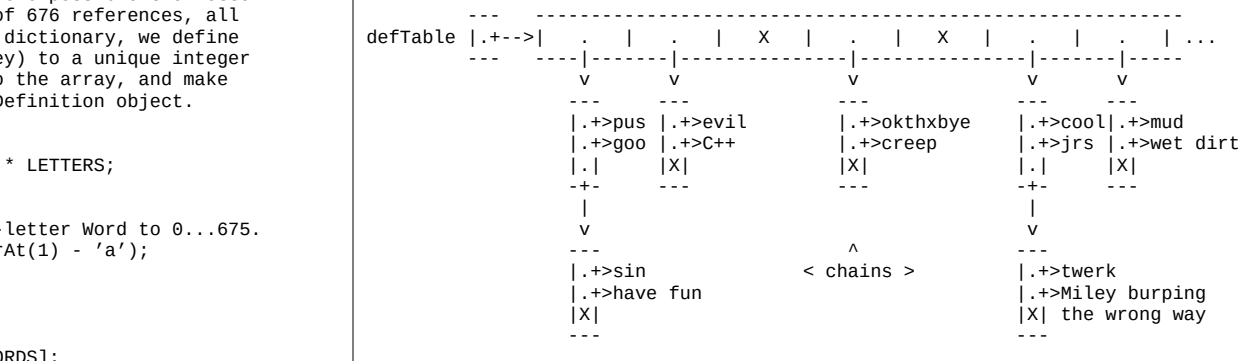
  h(hashCode) = hashCode mod N.

Hash codes are often negative, so remember that mod is not the same as Java's remainder operator "%". If you compute hashCode % N, check if the result is negative, and add N if it is.

With this compression function, no matter how long and variegated the keys are, we can map them into a table whose size is not much greater than the actual number of entries we want to store. However, we've created a new problem: several keys are hashed to the same bucket in the table if h(hashCode1) = h(hashCode2). This circumstance is called a _collision_.

How do we handle collisions without losing entries? We use a simple idea called _chaining_. Instead of having each bucket in the table reference one entry, we have it reference a linked list of entries, called a _chain_. If several keys are mapped to the same bucket, their definitions all reside in that bucket's linked list.

Chaining creates a second problem: how do we know which definition corresponds to which word? The answer is that we must store each key in the table with its definition. The easiest way to do this is to have each listnode store an _entry_ that has references to both a key (the word) and an associated value (its definition).



Hash tables usually support at least three operations. An Entry object references a key and its associated value.

public Entry insert(key, value)
  Compute the key's hash code and compress it to determine the entry's bucket.
  Insert the entry (key and value together) into that bucket's list.
public Entry find(key)
  Hash the key to determine its bucket. Search the list for an entry with the given key. If found, return the entry; otherwise, return null.
public Entry remove(key)
  Hash the key to determine its bucket. Search the list for an entry with the given key. Remove it from the list if found. Return the entry or null.

What if two entries with the same key are inserted? There are two approaches.
(1) Following Goodrich and Tamassia, we can insert both, and have find() or remove() arbitrarily return/remove one. Goodrich and Tamassia also propose a method findAll() that returns all the entries with a given key.
(2) Replace the old value with the new one, so only one entry with a given key exists in the table.
Which approach is best? It depends on the application.

WARNING: When an object is stored as a key in a hash table, an application should never change the object in a way that will change its hash code. If you do so, the object will thenceforth be in the wrong bucket.

The _load_factor_ of a hash table is n/N, where n is the number of keys in the table and N is the number of buckets. If the load factor stays below one (or a small constant), and the hash code and compression function are "good," and there are no duplicate keys, then the linked lists are all short, and each operation takes O(1) time. However, if the load factor grows too large (n >> N), performance is dominated by linked list operations and degenerates to O(n) time (albeit with a much smaller constant factor than if you replaced the hash table with one singly-linked list). A proper analysis requires a little probability theory, so we'll put it off until near the end of the semester.

Hash Codes and Compression Functions

Hash codes and compression functions are a bit of a black art. The ideal hash code and compression function would map each key to a uniformly distributed random bucket from zero to N - 1. By "random", I don't mean that the function is different each time; a given key always hashes to the same bucket. I mean that two different keys, however similar, will hash to independently chosen integers, so the probability they'll collide is 1/N. This ideal is tricky to obtain.

In practice, it's easy to mess up and create far more collisions than necessary. Let's consider bad compression functions first. Suppose the keys are integers, and each integer's hash code is itself, so hashCode(i) = i.

Suppose we use the compression function h(hashCode) = hashCode mod N, and the number N of buckets is 10,000. Suppose for some reason that our application only ever generates keys that are divisible by 4. A number divisible by 4 mod 10,000 is still a number divisible by 4, so three quarters of the buckets are never used! Thus the average bucket has about four times as many entries as it ought to.

The same compression function is much better if N is prime. With N prime, even if the hash codes are always divisible by 4, numbers larger than N often hash to buckets not divisible by 4, so all the buckets can be used.

For reasons I won't explain (see Goodrich and Tamassia Section 9.2.4 if you're interested),

```
h(hashCode) = ((a * hashCode + b) mod p) mod N
```

is a yet better compression function. Here, a, b, and p are positive integers, p is a large prime, and p >> N. Now, the number N of buckets doesn't need to be prime.

I recommend always using a known good compression function like the two above. Unfortunately, it's still possible to mess up by inventing a hash code that creates lots of conflicts even before the compression function is used. We'll discuss hash codes next lecture.

CS 61B: Lecture 22
Wednesday, March 12, 2014

Today's reading: Goodrich & Tamassia, Chapter 5.

DICTIONARIES (continued)
============

Hash Codes

Since hash codes often need to be designed specially for each new object, you're left to your own wits. Here is an example of a good hash code for Strings.

```
  private static int hashCode(String key) {
    int hashVal = 0;
    for (int i = 0; i < key.length(); i++) {
      hashVal = (127 * hashVal + key.charAt(i)) % 16908799;
    }
    return hashVal;
  }
```

By multiplying the hash code by 127 before adding in each new character, we make sure that each character has a different effect on the final result. The "%" operator with a prime number tends to "mix up the bits" of the hash code. The prime is chosen to be large, but not so large that 127 * hashVal + key.charAt(i) will ever exceed the maximum possible value of an int.

The best way to understand good hash codes is to understand why bad hash codes are bad. Here are some examples of bad hash codes on Words.

[1] Sum up the ASCII values of the characters. Unfortunately, the sum will rarely exceed 500 or so, and most of the entries will be bunched up in a few hundred buckets. Moreover, anagrams like "pat," "tap," and "apt" will collide.
[2] Use the first three letters of a word, in a table with 26^3 buckets. Unfortunately, words beginning with "pre" are much more common than words beginning with "xzq", and the former will be bunched up in one long list. This does not approach our uniformly distributed ideal.
[3] Consider the "good" hashCode() function written out above. Suppose the prime modulus is 127 instead of 16908799. Then the return value is just the last character of the word, because (127 * hashVal) % 127 = 0. That's why 127 and 16908799 were chosen to have no common factors.

Why is the hashCode() function presented above good? Because we can find no obvious flaws, and it seems to work well in practice. (A black art indeed.)

Resizing Hash Tables

Sometimes we can't predict in advance how many entries we'll need to store. If the load factor n/N (entries per bucket) gets too large, we are in danger of losing constant-time performance.

One option is to enlarge the hash table when the load factor becomes too large (typically larger than 0.75). Allocate a new array (typically at least twice as long as the old), then walk through all the entries in the old array and _rehash_ them into the new.

Take note: you CANNOT just copy the linked lists to the same buckets in the new array, because the compression functions of the two arrays will certainly be incompatible. You have to rehash each entry individually.

You can also shrink hash tables (e.g., when n/N < 0.25) to free memory, if you think the memory will benefit something else. (In practice, it's only sometimes worth the effort.)

Obviously, an operation that causes a hash table to resize itself takes more than O(1) time; nevertheless, the _average_ over the long run is still O(1) time per operation.

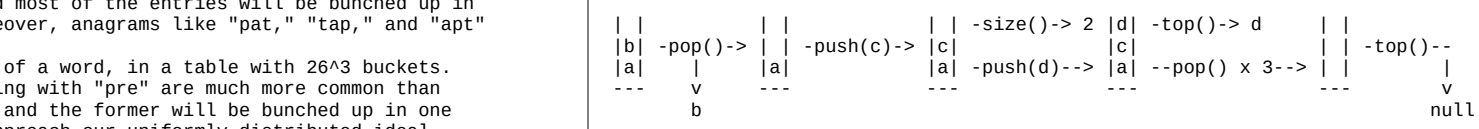
Transposition Tables: Using a Dictionary to Speed Game Trees

An inefficiency of unadorned game tree search is that some grids can be reached through many different sequences of moves, and so the same grid might be evaluated many times. To reduce this expense, maintain a hash table that records previously encountered grids. This dictionary is called a _transposition_table_. Each time you compute a grid's score, insert into the dictionary an entry whose key is the grid and whose value is the grid's score. Each time the minimax algorithm considers a grid, it should first check whether the grid is in the transposition table; if so, its score is returned immediately. Otherwise, its score is evaluated recursively and stored in the transposition table.

Transposition tables will only help you with your project if you can search to a depth of at least three ply (within the five second time limit). It takes three ply to reach the same grid two different ways.

After each move is taken, the transposition table should be emptied, because you will want to search grids to a greater depth than you did during the previous move.

STACKS
======
A _stack_ is a crippled list. You may manipulate only the item at the top of the stack. The main operations: you may "push" a new item onto the top of the stack; you may "pop" the top item off the stack; you may examine the "top" item of the stack. A stack can grow arbitrarily large.




```
public interface Stack {
  public int size();
  public boolean isEmpty();
  public void push(Object item);
  public Object pop();
  public Object top();
}
```

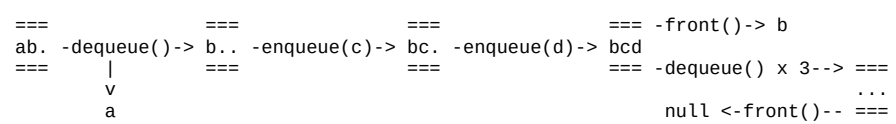
In any reasonable implementation, all these methods run in O(1) time.
A stack is easily implemented as a singly-linked list, using just the front(), insertFront(), and removeFront() methods.

Why talk about Stacks when we already have Lists? Mainly so you can carry on discussions with other computer programmers. If somebody tells you that an algorithm uses a stack, the limitations of a stack give you a hint how the algorithm works.

Sample application: Verifying matched parentheses in a String like "{[(){[]}](){}". Scan through the String, character by character.
- When you encounter a lefty--'{', '[', or '('--push it onto the stack.
- When you encounter a righty, pop its counterpart from atop the stack, and check that they match.

If there's a mismatch or null returned, or if the stack is not empty when you reach the end of the string, the parentheses are not properly matched.

QUEUES
======
A _queue_ is also a crippled list. You may read or remove only the item at the front of the queue, and you may add an item only to the back of the queue. The main operations: you may "enqueue" an item at the back of the queue; you may "dequeue" the item at the front; you may examine the "front" item. Don't be fooled by the diagram; a queue can grow arbitrarily long.



Sample Application: Printer queues. When you submit a job to be printed at a selected printer, your job goes into a queue. When the printer finishes printing a job, it dequeues the next job and prints it.

```
public interface Queue {
  public int size();
  public boolean isEmpty();
  public void enqueue(Object item);
  public Object dequeue();
  public Object front();
}
```

In any reasonable implementation, all these methods run in O(1) time. A queue is easily implemented as a singly-linked list with a tail pointer.

DEQUES
======
A _deque_ (pronounced "deck") is a Double-Ended QUEue. You can insert and remove items at both ends. You can easily build a fast deque using a doubly-linked list. You just have to add removeFront() and removeBack() methods, and deny applications direct access to listnodes. Obviously, deques are less powerful than lists whose listnodes are accessible.

Postscript: A Faster Hash Code (not examinable)

Here's another hash code for Strings, attributed to one P. J. Weinberger, which has been thoroughly tested and performs well in practice. It is faster than the one above, because it relies on bit operations (which are very fast) rather than the % operator (which is slow by comparison). You will learn about bit operations in CS 61C. Please don't ask me to explain them to you.

```
static int hashCode(String key) {
  int code = 0;

  for (int i = 0; i < key.length(); i++) {
    code = (code << 4) + key.charAt(i);
    code = (code & 0x0fffffff) ^ ((code & 0xf0000000) >> 24);
  }

  return code;
}
```

CS 61B: Lecture 23
Monday, March 17, 2014

ASYMPTOTIC ANALYSIS (continued): More Formalism

|---|
| Omega(f(n)) is the set of all functions T(n) that satisfy: |
| |
| There exist positive constants d and N such that, for all n >= N, |
T(n) >= d f(n)

^^^^^^^^^^ Compare with the definition of Big-Oh: T(n) <= c f(n). ^^^^^^^^^^

Omega is the reverse of Big-Oh. If T(n) is in O(f(n)), f(n) is in Omega(T(n)).

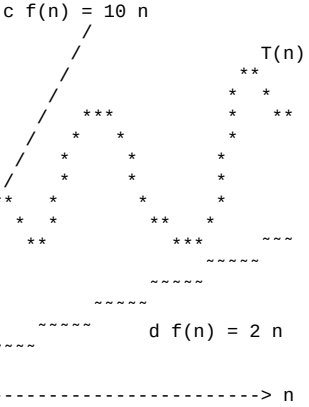
2n	is in Omega(n)	BECAUSE	n is in O(2n).
n^2	is in Omega(n)	BECAUSE	n is in O(n^2).
n^2	is in Omega(3 n^2 + n log n)	BECAUSE	3 n^2 + n log n is in O(n^2).

Big-Omega gives us a LOWER BOUND on a function, just as Big-Oh gives us an UPPER BOUND. Big-Oh says, "Your algorithm is at least this good." Big-Omega says, "Your algorithm is at least this bad."

Recall that Big-Oh notation can be misleading because, for instance, n is in O(n^8). If we know both a lower bound and an upper bound for a function, and they're both the same bound asymptotically (i.e. they differ only by a constant factor), we can use Big-Theta notation to precisely specify the function's asymptotic behavior.

|---|
| Theta(f(n)) is the set of all functions that are in both of |
| |
O(f(n)) and Omega(f(n)).

But how can a function be sandwiched between f(n) and f(n)?
Easy: we choose different constants (c and d) for the upper bound and lower bound. For instance, here is a function T(n) in Theta(n):



If we extend this graph infinitely far to the right, and find that T(n) remains always sandwiched between 2n and 10n, then T(n) is in Theta(n). If T(n) is an algorithm's worst-case running time, the algorithm will never exhibit worse than linear performance, but it can't be counted on to exhibit better than linear performance, either.

Theta is symmetric: if f(n) is in Theta(g(n)), then g(n) is in Theta(f(n)). For instance, n^3 is in Theta(3 n^3 - n^2), and 3 n^3 - n^2 is in Theta(n^3). n^3 is not in Theta(n), and n is not in Theta(n^3).

Big-Theta notation isn't potentially misleading in the way Big-Oh notation can be: n is NOT in Omega(n^8). If your algorithm's running time is in Theta(n^8), it IS slow.

However, some functions are not in "Theta" of anything simple. For example, the function f(n) = n (1 + sin n) is in O(n) and Omega(0), but it's not in Theta(n) nor Theta(0). f(n) keeps oscillating back and forth between zero and ever-larger numbers. We could say that f(n) is in Theta(2n (1 + sin n)), but that's not a simplification.

Remember that the choice of O, Omega, or Theta is _independent_ of whether we're talking about worst-case running time, best-case running time, average-case running time, memory use, annual beer consumption as a function of population, or some other function. The function has to be specified. "Big-Oh" is NOT a synonym for "worst-case running time," and Omega is not a synonym for "best-case running time."

ALGORITHM ANALYSIS
==========
Problem #1: Given a set of p points, find the pair closest to each other.
Algorithm #1: Calculate the distance between each pair; return the minimum.

There are p (p - 1) / 2 pairs, and each pair takes constant time to examine. Therefore, worst- and best-case running times are in Theta(p^2).

Often, you can figure out the running time of an algorithm just by looking at the loops--their loop bounds and how they are nested. For example, in the closest pair code below, the outer loop iterates p times, and the inner loop iterates an average of roughly p / 2 times, which multiply to Theta(p^2) time.

```
double minDistance = point[0].distance(point[1]);

/* Visit a pair (i, j) of points. */
for (int i = 0; i < numPoints; i++) {
  /* We require that j > i so that each pair is visited only once. */
  for (int j = i + 1; j < numPoints; j++) {
    double thisDistance = point[i].distance(point[j]);
    if (thisDistance < minDistance) {
      minDistance = thisDistance;
    }
  }
}
```

But doubly-nested loops don't always mean quadratic running time! The next example has the same loop structure, but runs in linear time.

Problem #2: Smooshing an array called "ints" to remove consecutive duplicates, from Homework 3.
Algorithm #2:

```
int i = 0, j = 0;

while (i < ints.length) {
  ints[j] = ints[i];
  do {
    i++;
  } while ((i < ints.length) && (ints[i] == ints[j]));
  j++;
}
// Code to fill in -1's at end of array omitted.
```

The outer loop can iterate up to ints.length times, and so can the inner loop. But the index "i" advances on _every_ iteration of the inner loop. It can't advance more than ints.length times before both loops end. So the worst-case running time of this algorithm is in Theta(ints.length). (So is the best-case time.)

Unfortunately, I can't give you a foolproof formula for determining the running time of any algorithm. You have to think! In fact, the problem of determining an algorithm's running time is, in general, as hard as proving _any_ mathematical theorem. For instance, I could give you an algorithm whose running time depends on whether the Riemann Hypothesis (one of the greatest unsolved questions in mathematics) is true or false.

Functions of Several Variables

Problem #3: Write a matchmaking program for w women and m men.
Algorithm #3: Compare each woman with each man. Decide if they're compatible.

If each comparison takes constant time then the running time, T(w, m), is in Theta(wm).

This means that there exist constants c, d, W, and M, such that d wm <= T(w, m) <= c wm for every w >= W and m >= M.

T is NOT in O(w^2), nor in O(m^2), nor in Omega(w^2), nor in Omega(m^2). Every one of these possibilities is eliminated either by choosing w >> m or m >> w. Conversely, w^2 is in neither O(wm) nor Omega(wm). You cannot asymptotically compare the functions wm, w^2, and m^2.

If we expand our service to help form women's volleyball teams as well, the running time is in Theta(w^6 + wm).

This expression cannot be simplified; neither term dominates the other. You cannot asymptotically compare the functions w^6 and wm.

Problem #4: Suppose you have an array containing n music albums, sorted by title. You request a list of all albums whose titles begin with "The Best of"; suppose there are k such albums.
Algorithm #4: Search for one matching album with binary search. Walk (in both directions) to find the other matching albums.

Binary search takes at most log n steps to find a matching album (if one exists). Next, the complete list of k matching albums is found, each in constant time. Thus, the worst-case running time is in

Theta(log n + k).

Because k can be as large as n, it is not dominated by the log n term. Because k can be as small as zero, it does not dominate the log n term. Hence, there is no simpler expression for the worst-case running time.

Algorithms like this are called _output-sensitive_, because their performance depends partly on the size k of the output, which can vary greatly.

Because binary search sometimes gets lucky and finds a match right away, the BEST-case running time is in

Theta(k).

Problem #5: Find the k-th item in an n-node doubly-linked list.
Algorithm #5: If k < 1 or k > n, report an error and return.
Otherwise, compare k with n - k.

If k <= n - k, start at the front of the list and walk forward k - 1 nodes.

Otherwise, start at the back of the list and walk backward n - k nodes.

If 1 <= k <= n, this algorithm takes Theta(min{k, n - k}) time (in all cases) This expression cannot be simplified: without knowing k and n, we cannot say that k dominates n - k or that n - k dominates k.

CS 61B: Lecture 24
Wednesday, March 19, 2014

Today's reading: Goodrich & Tamassia, Chapter 7.

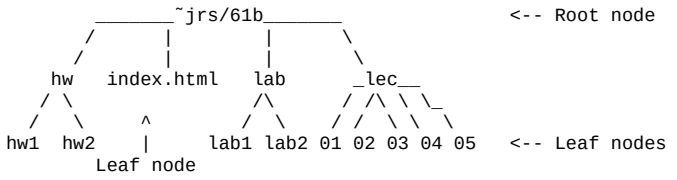
ROOTED TREES
============
A _tree_ consists of a set of nodes and a set of edges that connect pairs of nodes. A tree has the property that there is exactly one path (no more, no less) between any two nodes of the tree. A _path_ is a sequence of one or more nodes, each consecutive pair being connected by an edge.

In a _rooted_ tree, one distinguished node is called the _root_. Every node c, except the root, has exactly one _parent_ node p, which is the first node after c on the path from c to the root. c is p's _child_. The root has no parent. A node can have any number of children.

Some other definitions:

- A _leaf_ is a node with no children.
- An _internal_node_ is a non-leaf node (having one or more children).
- _Siblings_ are nodes with the same parent.
- The _ancestors_ of a node d are the nodes on the path from d to the root. These include d's parent, d's parent's parent, d's parent's parent's parent, and so forth up to the root. Technically, the ancestors of d also include d itself, which makes you wonder about d's sex life. The root is an ancestor of every node in the tree.
- If a is an ancestor of d, then d is a _descendant_ of a.
- The _length_ of a path is the number of edges in the path.
- The _depth_ of a node n is the length of the path from n to the root. (The depth of the root is zero.)
- The _height_ of a node n is the length of the path from n to its deepest descendant. (The height of a leaf node is zero.)
- The height of a tree is the depth of its deepest node = height of the root.
- The _subtree_ rooted at node n is the tree formed by n and its descendants.
- A _binary_tree_ is a tree in which no node has more than two children, and every child is either a _left_child_ or a _right_child_, even if it's the only child its parent has.

A commonly encountered application of trees is the directory structure of a file system.



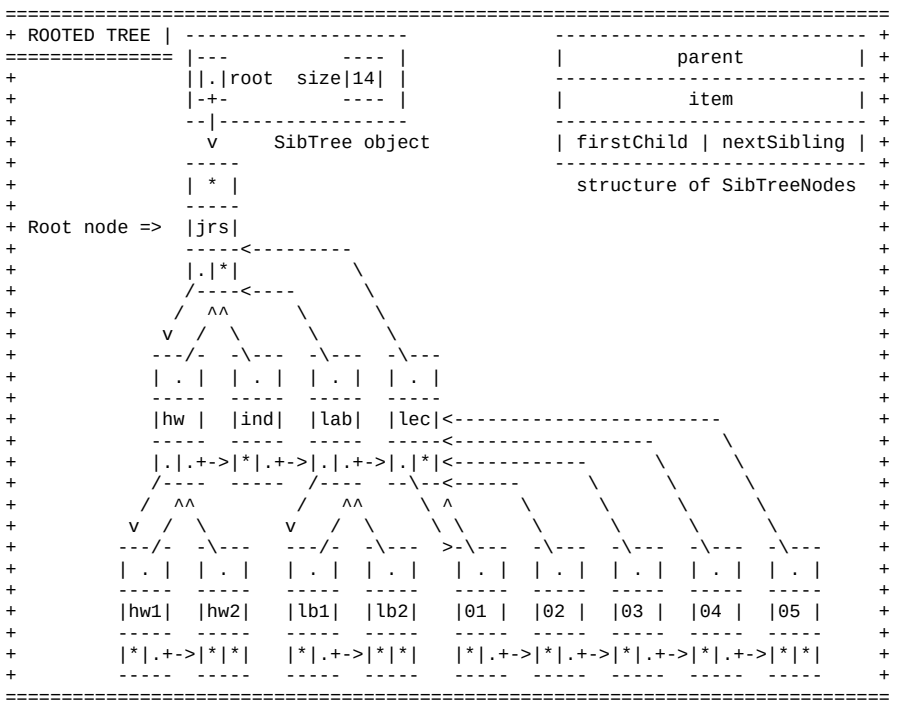
Representing Rooted Trees

Goodrich and Tamassia present a data structure in which each node has three references: one reference to an item, one reference to the node's parent, and one reference to the node's children, which can be stored in any reasonable data structure like a linked list. Directories are typically stored this way, but the lists they use are represented very differently than our list ADTs.

Another popular tree representation spurns separately encapsulated linked lists so that siblings are directly linked. It retains the "item" and "parent" references, but instead of referencing a list of children, each node references just its leftmost child. Each node also references its next sibling to the right. The "nextSibling" references are used to join the children of a node in a singly-linked list, whose head is the node's "firstChild".

I'll call this tree a "SibTree", since siblings are central to the representation. The nodes I call "SibTreeNodes".

```
class SibTreeNode {                  |  class SibTree {
  Object item;                       |    SibTreeNode root;
  SibTreeNode parent;                |    int size;
  SibTreeNode firstChild;            |  }
  SibTreeNode nextSibling;           |
}                                    |
```



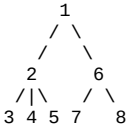
Tree Traversals

A _traversal_ is a manner of _visiting_ each node in a tree once. What you do when visiting any particular node depends on the application; for instance, you might print a node's value, or perform some calculation upon it. There are several different traversals, each of which orders the nodes differently.

Many traversals can be defined recursively. In a _preorder_ traversal, you visit each node before recursively visiting its children, which are visited from left to right. The root is visited first.

```
class SibTreeNode {
  public void preorder() {
    this.visit();
    if (firstChild != null) {
      firstChild.preorder();
    }
    if (nextSibling != null) {
      nextSibling.preorder();
    }
  }
}
```

Suppose your method visit() numbers the nodes in the order they're visited. A preorder traversal visits the nodes in this order.



Each node is visited only once, so a preorder traversal takes O(n) time, where n is the number of nodes in the tree. All the traversals we will consider take O(n) time.

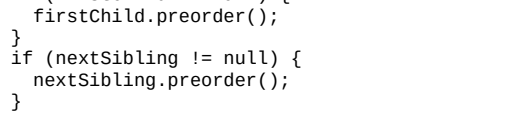
A preorder traversal is a natural way to print a directory's structure. Simply have the method visit() print each node of the tree.

```
~jrs/61b
   hw
      hw1
      hw2
   index.html
   lab
      lab1
      lab2
   lec
      01
      02
      03
      04
      05
```

In a _postorder_ traversal, you visit each node's children (in left-to-right order) before the node itself.

```
  public void postorder() {
    if (firstChild != null) {
      firstChild.postorder();
    }
    this.visit();
    if (nextSibling != null) {
      nextSibling.postorder();
    }
  }
```

A postorder traversal visits the nodes in this order.



The postorder() code is trickier than it looks. The best way to understand it is to draw a depth-two tree on paper, then pretend you're the computer and execute the algorithm carefully. Trust me on this. It's worth your time.

A postorder traversal is the natural way to sum the total disk space used in the root directory and its descendants. The method visit() sums "this" node's disk space with the disk space of all its children. In the example above, a postorder traversal would first sum the sizes of the files in hw1/ and hw2/; then it would visit hw/ and sum its two children. The last thing it would compute is the total disk space at the root ~jrs/61b/, which sums all the files in the tree.

Binary trees allow for an _inorder_ traversal: recursively traverse the root's left subtree (rooted at the left child), then the root itself, then the root's right subtree. The preorder, inorder, and postorder traversals of an expression tree will print a _prefix_, _infix_, or _postfix_ expression, respectively.

Prefix: + * 3 7 ^ 4 2

Infix: 3 * 7 + 4 ^ 2

Postfix: 3 7 * 4 2 ^ +

In a _level-order_ traversal, you visit the root, then all the depth-1 nodes (from left to right), then all the depth-2 nodes, et cetera. The level-order traversal of our expression tree is "+ * ^ 3 7 4 2" (which doesn't mean much).

Unlike the three previous traversals, a level-order traversal is not straightforward to define recursively. However, a level-order traversal can be done in O(n) time. Use a queue, which initially contains only the root. Then repeat the following steps:
- Dequeue a node.
- Visit it.
- Enqueue its children (in order from left to right).

Continue until the queue is empty.

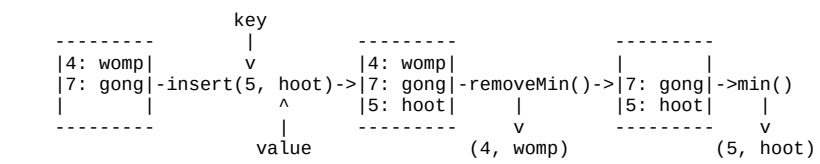
A final thought: if you use a stack instead of a queue, and push each node's children in reverse order--from right to left (so they pop off the stack in order from left to right)--you perform a preorder traversal. Think about why.

CS 61B: Lecture 25
Wednesday, March 19, 2014

Today's reading: Goodrich & Tamassia, Sections 8.1-8.3.

PRIORITY QUEUES
==========
A priority queue, like a dictionary, contains _entries_ that each consist of a key and an associated value. However, whereas a dictionary is used when we want to be able to look up arbitrary keys, a priority queue is used to prioritize entries. Define a total order on the keys (e.g. alphabetical order). You may identify or remove the entry whose key is the lowest (but no other entry). This limitation helps to make priority queues fast. However, an entry with any key may be inserted at any time.

For concreteness, let's use Integer objects as our keys. The main operations:
- insert() adds an entry to the priority queue;
- min() returns the entry with the minimum key; and
- removeMin() both removes and returns the entry with the minimum key.



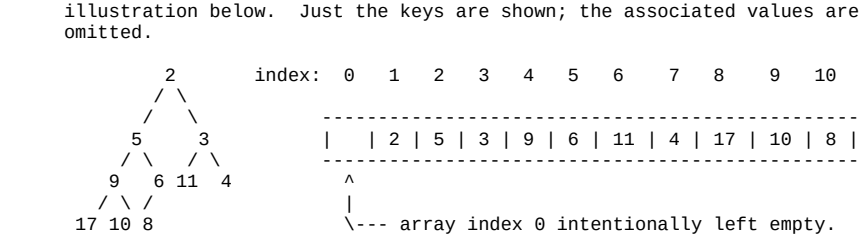
Priority queues are most commonly used as "event queues" in simulations. Each value on the queue is an event that is expected to take place, and each key is the time the event takes place. A simulation operates by removing successive events from the queue and simulating them. This is why most priority queues return the minimum, rather than maximum, key: we want to simulate the events that occur first first.

```
public interface PriorityQueue {
  public int size();
  public boolean isEmpty();
  Entry insert(Object key, Object value);
  Entry min();
  Entry removeMin();
}
```

See page 340 of Goodrich & Tamassia for how they implement an "Entry".

Binary Heaps: An Implementation of Priority Queues

A _complete_binary_tree_ is a binary tree in which every row is full, except possibly the bottom row, which is filled from left to right as in the illustration below. Just the keys are shown; the associated values are omitted.



A _binary_heap_ is a complete binary tree whose entries satisfy the _heap-order_property_: no child has a key less than its parent's key. Observe that every subtree of a binary heap is also a binary heap, because every subtree is complete and satisfies the heap-order property.

Because it is complete, a binary heap can be stored compactly as an array of entries. We map tree nodes to array indices with _level_numbering_, which places the root at index 1 and orders the remaining nodes by a level-order traversal of the tree.

Observe that if a node's index is i, its children's indices are 2i and 2i+1, and its parent's index is floor(i/2). Hence, no node needs to store explicit references to its parent or children. (Array index 0 is left empty to make the indexing work out nicely. If we instead put the root at index 0, node i's children are at indices 2i+1 and 2i+2, and its parent is at floor([i-1]/2).)

We can use either an array-based or a node-and-reference-based tree data structure, but the array representation tends to be faster (by a significant constant factor) because there is no need to read and write the references that connect nodes to each other, cache performance is better, and finding the last node in the level order is easier.

Just like in hash tables, either each tree node has two references (one for the key, and one for the value), or each node references an "Entry" object (see page 340 of Goodrich and Tamassia).

Let's look at how we implement priority queue operations with a binary heap.

[1] Entry min();

The heap-order property ensures that the entry with the minimum key is always at the top of the heap. Hence, we simply return the entry at the root node. If the heap is empty, return null or throw an exception.

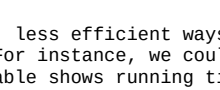
[2] Entry insert(Object k, Object v);

Let x be the new entry (k, v), whose key is k and whose value is v. We place the new entry x in the bottom level of the tree, at the first free spot from the left. (If the bottom level is full, start a new level with x at the far left.) In an array-based implementation, we place x in the first free location in the array (excepting index 0).

Of course, the new entry's key may violate the heap-order property. We correct this by having the entry bubble up the tree until the heap-order property is satisfied. More precisely, we compare x's key with its parent's key. While x's key is less, we exchange x with its parent, then repeat the test with x's new parent. Continue until x's key is greater than or equal to its parent, or x reaches the root. For instance, if we insert an entry whose key is 2:

As this example illustrates, a heap can contain multiple entries with the same key. (After all, in a typical simulation, we can't very well outlaw multiple events happening at the same time.)

When we finish, is the heap-order property satisfied?
Yes, if the heap-order property was satisfied before the insertion. Let's look at a typical exchange of x with a parent p (right) during the insertion operation. Since the heap-order property was satisfied before the insertion, we know that p <= s (where s is x's sibling), p <= l, and p <= r (where l and r are x's children). We swap only if x < p, which implies that x < s; after the swap, x is the parent of s. After the swap, p is the parent of l and r. All other relationships in the subtree rooted at x are unchanged, so after the swap, the tree rooted at x has the heap-order property.



For maximum speed, don't put x at the bottom of the tree and bubble it up. Instead, bubble a hole up the tree, then fill in x. This modification saves the time that would be spent setting a sequence of references to x that are going to change anyway.

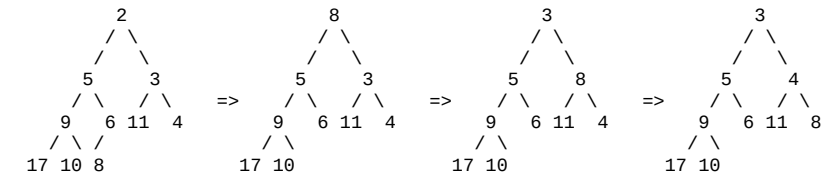
insert() returns an Entry object representing (k, v).

[3] Entry removeMin();

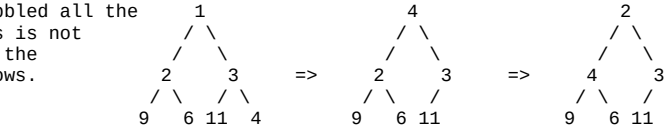
If the heap is empty, return null or throw an exception. Otherwise, begin by removing the entry at the root node and saving it for the return value. This leaves a gaping hole at the root. We fill the hole with the last entry in the tree (which we call "x"), so that the tree is still complete.

It is unlikely that x has the minimum key. Fortunately, both subtrees rooted at the root's children are heaps, and thus the new mimimum key is one of these two children. We bubble x down the heap until the heap-order property is satisfied, as follows. We compare x's key with both its children's keys. While x has a child whose key is smaller, swap x with the child having the minimum key, then repeat the test with x's new children. Continue until x is less than or equal to its children, or reaches a leaf.

Consider running removeMin() on our original tree.



Above, the entry bubbled all the way to a leaf. This is not always the case, as the example at right shows.



For maximum speed, don't put x at the root and bubble it down. Instead, bubble a hole down the tree, then fill in x.

Running Times

There are other, less efficient ways we could implement a priority queue than using a heap. For instance, we could use a list or array, sorted or unsorted. The following table shows running times for all, with n entries in the queue.

	Binary Heap	Sorted List/Array	Unsorted List/Array
min()	Theta(1)	Theta(1)	Theta(n)
insert()			
worst-case	Theta(log n) *	Theta(n)	Theta(1) *
best-case	Theta(1) *	Theta(1) *	Theta(1) *
removeMin()			
worst-case	Theta(log n)	Theta(1) **	Theta(n)
best-case	Theta(1)	Theta(1) **	Theta(n)

* If you're using an array-based data structure, these running times assume that you don't run out of room. If you do, it will take Theta(n) time to allocate a larger array and copy the entries into it. However, if you double the array size each time, the _average_ running time will still be as indicated.
** Removing the minimum from a sorted array in constant time is most easily done by keeping the array always sorted from largest to smallest.

In a binary heap, min's running time is clearly in Theta(1).

insert() puts an entry x at the bottom of the tree and bubbles it up. At each level of the tree, it takes O(1) time to compare x with its parent and swap if indicated. An n-node complete binary tree has height floor(log2 n). In the worst case, x will bubble all the way to the top, taking Theta(log n) time.

Similarly, removeMin() may cause an entry to bubble all the way down the heap, taking Theta(log n) worst-case time.

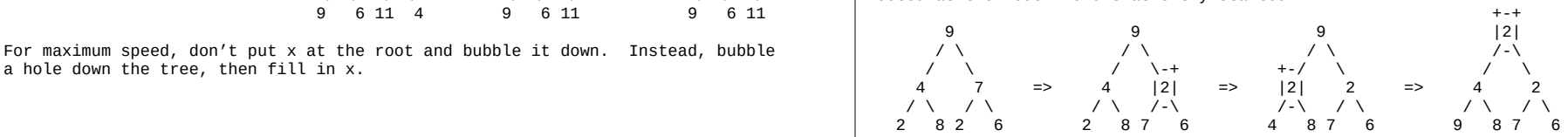
Bottom-Up Heap Construction

Suppose we are given a bunch of randomly ordered entries, and want to make a heap out of them. We could insert them one by one in O(n log n) time, but there's a faster way. We define one more heap operation.

[4] void bottomUpHeap();

First, we make a complete tree out of the entries, in any order. (If we're using an array representation, we just throw all the entries into an array.) Then we work backward from the last internal node (non-leaf node) to the root node, in reverse order in the array or the level-order traversal. When we visit a node this way, we bubble its entry down the heap as in removeMin().

Before we bubble an entry down, we know (inductively) that its two child subtrees are heaps. Hence, by bubbling the entry down, we create a larger heap rooted at the node where that entry started.



The running time of bottomUpHeap is tricky to derive. If each internal node bubbles all the way down, then the running time is proportional to the sum of the heights of all the nodes in the tree. Page 371 of Goodrich and Tamassia has a simple and elegant argument showing that this sum is less than n, where n is the number of entries being coalesced into a heap. Hence, the running time is in Theta(n), which beats inserting n entries into a heap individually.

Postscript: Other Types of Heaps (not examinable)

Binary heaps are not the only heaps in town. Several important variants are called "mergeable heaps", because it is relatively fast to combine two mergeable heaps together into a single mergeable heap. We will not describe these complicated heaps in CS 61B, but it's worthwhile for you to know they exist in case you ever need one.

The best-known mergeable heaps are called "binomial heaps," "Fibonacci heaps," "skew heaps," and "pairing heaps." Fibonacci heaps have another remarkable property: if you have a reference to an arbitrary node in a Fibonacci heap, you can decrease its key in constant time. (Pairing heaps are suspected of having the same property, but nobody knows for sure.) This operation is used frequently by Dijkstra's algorithm, an important algorithm for finding the shortest path in a graph. The following running times are all worst-case.

	Binary	Binomial	Skew	Pairing	Fibonacci
insert()	O(log n)	O(log n)	O(1)	O(log n) *	O(1)
removeMin()	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
merge()	O(n)	O(log n)	O(1)	O(log n) *	O(1)
decreaseKey()	O(log n)	O(log n)	O(log n)	O(log n) *	O(1)

* Conjectured to be O(1), but nobody has proven or disproven it.

The time bounds given here for skew heaps, pairing heaps, and Fibonacci heaps are "amortized" bounds, not worst case bounds. This means that, if you start from an empty heap, any sequence of operations will take no more than the given time bound on average, although individual operations may occasionally take longer. We'll discuss amortized analysis near the end of the semester.

CS 61B: Lecture 26
Monday, March 31, 2014

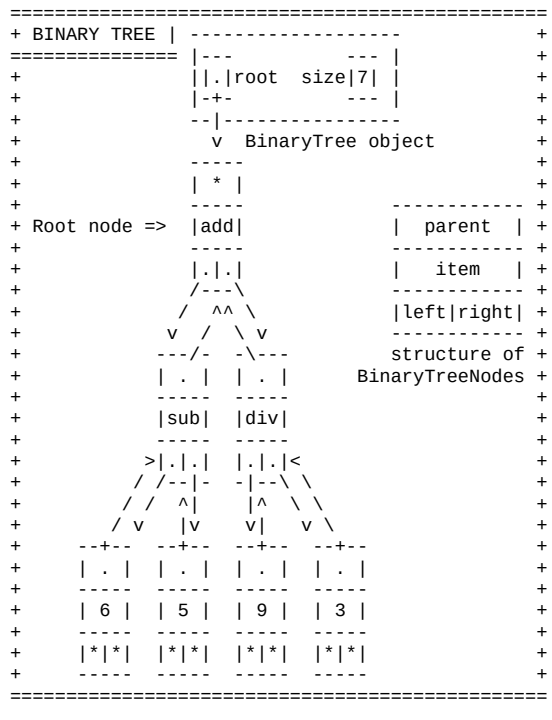
Today's reading: Goodrich & Tamassia, Section 10.1.

Representing Binary Trees

Recall that a binary tree is a rooted tree wherein no node has more than two children. Additionally, every child is either a _left_child_ or a _right_child_ of its parent, even if it is its parent's only child.

In the most popular binary tree representation, each tree node has three references to neighboring tree nodes: a "parent" reference, and "left" and "right" references for the two children. (For some algorithms, the "parent" references are unnecessary.) Each node also has an "item" reference.

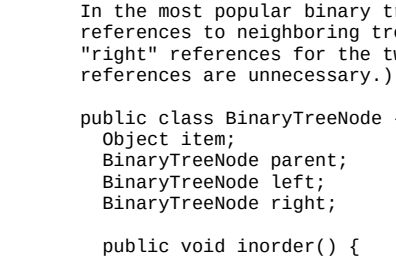
```
public class BinaryTreeNode {        |  public class BinaryTree {
  Object item;                       |    BinaryTreeNode root;
  BinaryTreeNode parent;             |    int size;
  BinaryTreeNode left;               |  }
  BinaryTreeNode right;              |

  public void inorder() {
    if (left != null) {
      left.inorder();
    }
    this.visit();
    if (right != null) {
      right.inorder();
    }
  }
}
```



BINARY SEARCH TREES

An _ordered_dictionary_ is a dictionary in which the keys have a total order, just like in a heap. You can insert, find, and remove entries, just as with a hash table. But unlike a hash table, you can quickly find the entry with minimum or maximum key, or the entry nearest another entry in the total order. An ordered dictionary does anything a dictionary or binary heap can do and more, albeit more slowly.

A simple implementation of an ordered dictionary is a binary search tree, wherein entries are maintained in a (somewhat) sorted order.



The _left_subtree_ of a node is the subtree rooted at the node's left child; the _right_subtree_ is defined similarly. A binary search tree satisfies the _binary_search_tree_ _invariant_: for any node X, every key in the left subtree of X is less than or equal to X's key, and every key in the right subtree of X is greater than or equal to X's key. You can verify this in the search tree at left: for instance, the root is 18, its left subtree (rooted at 12) contains numbers from 1 to 17, and its right subtree (rooted at 25) contains numbers from 25 to 30.

When a node has only one child, that child is either a left child or a right child, depending on whether its key is smaller or larger than its parent's key. (A key equal to the parent's key can go into either subtree.)

An inorder traversal of a binary search tree visits the nodes in sorted order. In this sense, a search tree maintains a sorted list of entries. However, operations on a search tree are usually more efficient than the same operations on a sorted linked list.

Let's replace the "Object item;" declaration in each node with "Entry entry;" where each Entry object stores a key and an associated value. The keys implement the Comparable interface, and the key.compareTo() method induces a total order on the keys (e.g. alphabetical or numerical order).

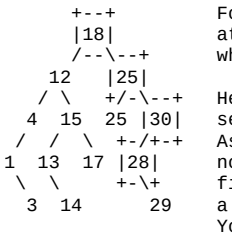
[1] Entry find(Object k);

```
public Entry find(Object k) {
  BinaryTreeNode node = root;                  // Start at the root.
  while (node != null) {
    int comp = ((Comparable) k).compareTo(node.entry.key());
    if (comp < 0) {                            // Repeatedly compare search
      node = node.left;                        // key k with current node; if
    } else if (comp > 0) {                     // k is smaller, go to the left
      node = node.right;                       // child; if k is larger, go to
    } else {    /* The keys are equal */       // the right child. Stop when
      return node.entry;                       // we find a match (success;
    }                                          // return the entry) or reach
  }                                            // a null pointer (failure;
  return null;                                 // return null).
}
```

This method only finds exact matches. What if we want to find the smallest key greater than or equal to k, or the largest key less than or equal to k? Fortunately, when searching downward through the tree for a key k that is not in the tree, we are certain to encounter both

- the node containing the smallest key greater than k (if any key is greater)
- the node containing the largest key less than k (if any key is less).

See Footnote 1 for an explanation why.



For instance, suppose we search for the key 27 in the tree at left. Along the way, we encounter the keys 25 and 28, which are the keys closest to 27 (below and above).

Here's how to implement a method smallestKeyNotSmaller(k): search for the key k in the tree, just like in find(). As you go down the tree, keep track of the smallest key not smaller than k that you've encountered so far. If you find the key k, you can return it immediately. If you reach a null pointer, return the best key you found on the path. You can implement largestKeyNotLarger(k) symmetrically.

[2] Entry min();
Entry max();

min() is very simple. If the tree is empty, return null. Otherwise, start at the root. Repeatedly go to the left child until you reach a node with no left child. That node has the minimum key.

max() is the same, except that you repeatedly go to the right child. In the tree above, observe the locations of the minimum (1) and maximum (30) keys.

[3] Entry insert(Object k, Object v);

insert() starts by following the same path through the tree as find(). (find() works _because_ it follows the same path as insert().) When it reaches a null reference, replace that null with a reference to a new node storing the entry (k, v).

Duplicate keys are allowed. If insert() finds a node that already has the key k, it puts the new entry in the left subtree of the older one. (We could just as easily choose the right subtree; it doesn't matter.)

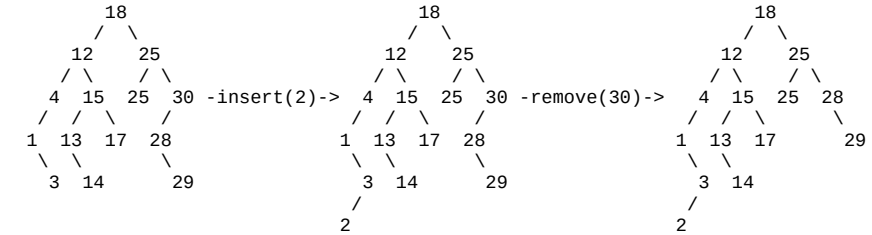
[4] Entry remove(Object k);

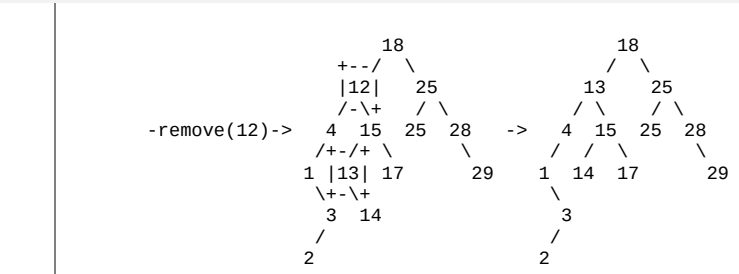
remove() is the most difficult operation. First, find a node with key k using the same algorithm as find(). Return null if k is not in the tree; otherwise, let n be the first node with key k.

If n has no children, we easily detach it from its parent and throw it away.

If n has one child, move n's child up to take n's place. n's parent becomes the parent of n's child, and n's child becomes the child of n's parent. Dispose of n.

If n has two children, however, we have to be a bit more clever. Let x be the node in n's right subtree with the smallest key. Remove x; since x has the minimum key in the subtree, x has no left child and is easily removed. Finally, replace n's entry with x's entry. x has the key closest to k that isn't smaller than k, so the binary search tree invariant still holds.





To ensure you understand the binary search tree operations, especially remove(), I recommend you inspect Goodrich and Tamassia's code on page 446. Be aware that Goodrich and Tamassia use sentinel nodes for the leaves of their binary trees; I think these waste an unjustifiably large amount of space.

Running Times of Binary Search Tree Operations

In a perfectly balanced binary tree (left) with height h, the number of nodes n is 2^(h+1) - 1. (See Footnote 2.) Therefore, no node has depth greater than log_2 n. The running times of find(), insert(), and remove() are all proportional to the depth of the last node encountered, so they all run in O(log n) worst-case time on a perfectly balanced tree.

On the other hand, it's easy to form a severely imbalanced tree like the one at right, wherein these operations will usually take linear time.

There's a vast middle ground of binary trees that are reasonably well-balanced, albeit certainly not perfectly balanced, for which search tree operations will run in O(log n) time. You may need to resort to experiment to determine whether any particular application will use binary search trees in a way that tends to generate somewhat balanced trees or not. If you create a binary search trees by inserting keys in a randomly chosen order, or if the keys are generated by a random process from the same distribution, then with high probability the tree will have height O(log n), and operations on the tree will take O(log n) time.

Unfortunately, there are occasions where you might fill a tree with entries that happen to be already sorted. In this circumstance, you'll create the disastrously imbalanced tree depicted at right. Technically, all operations on binary search trees have Theta(n) worst-case running time.

For this reason, researchers have developed a variety of algorithms for keeping search trees balanced. Prominent examples include 2-3-4 trees (which we'll cover next lecture), splay trees (in one month), and B-trees (in CS 186).

Footnote 1: When we search for a key k not in the binary search tree, why are we guaranteed to encounter the two keys that bracket it? Let x be the smallest key in the tree greater than k. Because k and x are "adjacent" keys, the result of comparing k with any other key y in the tree is the same as comparing x with y. Hence, find(k) will follow exactly the same path as find(x) until it reaches x. (After that, it may continue downward.) The same argument applies to the largest key less than k.

Footnote 2: A perfectly balanced binary tree has 2^i nodes at depth i, where $0 \le i \le h$. Hence, the total number of nodes is $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$.

CS 61B: Lecture 27
Wednesday, April 2, 2014

2-3-4 TREES

Last lecture, we learned about the Ordered Dictionary ADT, and we learned one data structure for implementing it: binary search trees. Today we learn a faster one.

A 2-3-4 tree is a perfectly balanced tree. It has a big advantage over regular binary search trees: because the tree is perfectly balanced, find, insert, and remove operations take O(log n) time, even in the worst case.

2-3-4 trees are thus named because every node has 2, 3, or 4 children, except leaves, which are all at the bottom level of the tree. Each node stores 1, 2, or 3 entries, which determine how other entries are distributed among its children's subtrees.

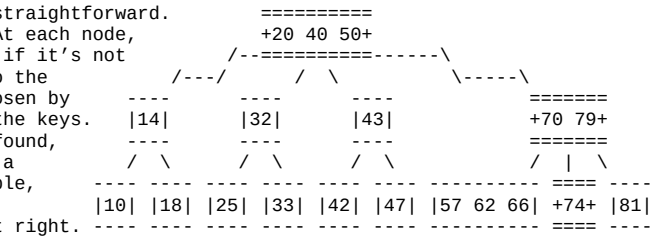
Each internal (non-leaf) node has one more child than entries. For example, a node with keys [20, 40, 50] has four children. Eack key k in the subtree rooted at the first child satisfies k <= 20; at the second child, 20 <= k <= 40; at the third child, 40 <= k <= 50; and at the fourth child, k >= 50.

WARNING: The algorithms for insertion and deletion I'll discuss today are different from those discussed by Goodrich and Tamassia. The text presents "bottom-up" 2-3-4 trees, so named because the effects of node splits at the bottom of the tree can work their way back up toward the root. I'll discuss "top-down" 2-3-4 trees, in which insertion and deletion finish at the leaves. Top-down 2-3-4 trees are usually faster than bottom-up ones, because both trees search down from the root to the leaves, but only the bottom-up trees sometimes go back up to the root. Goodrich and Tamassia call 2-3-4 trees "(2, 4) trees".

2-3-4 trees are a type of B-tree, which you may learn about someday in connection with fast disk access for database systems. B-trees on disks usually use the top-down insertion/deletion algorithms, because accessing a disk track is slow, so you'd rather not revisit it multiple times.

[1] Entry find(Object k);

Finding an entry is straightforward. Start at the root. At each node, check for the key k; if it's not present, move down to the appropriate child chosen by comparing k against the keys. Continue until k is found, or k is not found at a leaf node. For example, find(74) visits the double-lined boxes at right.

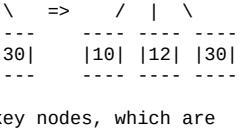


Incidentally, you can define an inorder traversal on 2-3-4 trees analogous to that on binary trees, and it visits the keys in sorted order.

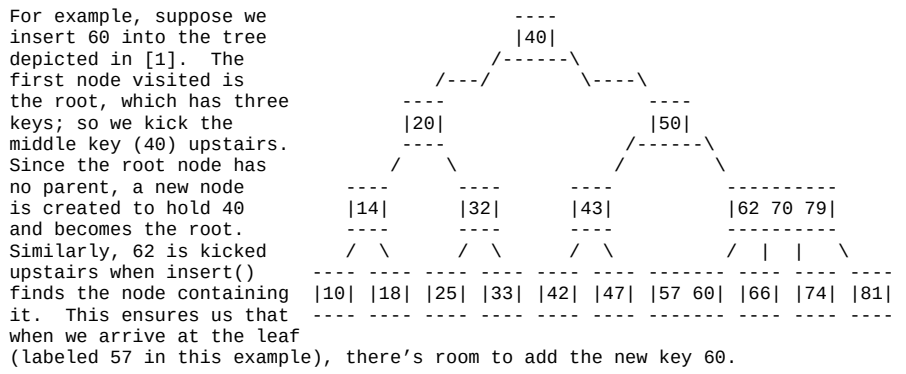
[2] Entry insert(Object k, Object e);

insert(), like find(), walks down the tree in search of the key k. If it finds an entry with key k, it proceeds to that entry's "left child" and continues.

Unlike find(), insert() sometimes modifies nodes of the tree as it walks down. Specifically, whenever insert() encounters a 3-key node, the middle key is ejected, and is placed in the parent node instead. Since the parent was previously treated the same way, the parent has at most two keys, and always has room for a third. The other two keys in the 3-key node are split into two separate 1-key nodes, which are divided underneath the old middle key (as the figure illustrates).



For example, suppose we insert 60 into the tree depicted in [1]. The first node visited is the root, which has three keys; so we kick the middle key (40) upstairs. Since the root node has no parent, a new node is created to hold 40 and becomes the root. Similarly, 62 is kicked upstairs when insert() finds the node containing it. This ensures us that when we arrive at the leaf (labeled 57 in this example), there's room to add the new key 60.



Observe that along the way, we created a new 3-key node "62 70 79". We do not kick its middle key upstairs until the next time it is visited.

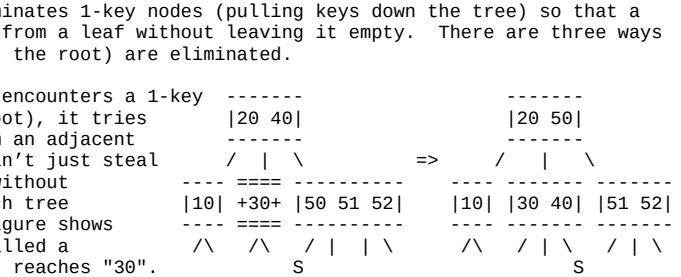
Again, the reasons why we split every 3-key node we encounter (and move its middle key up one level) are (1) to make sure there's room for the new key in the leaf node, and (2) to make sure that above the leaves, there's room for any key that gets kicked upstairs. Sometimes, an insertion operation increases the height of the tree by one by creating a new root.

[3] Entry remove(Object k);

2-3-4 tree remove() is similar to remove() on binary search trees: you find the entry you want to remove (having key k). If it's in a leaf, you remove it. If it's in an internal node, you replace it with the entry with the next higher key. That entry is always in a leaf. In either case, you remove an entry from a leaf in the end.

Like insert(), remove() changes nodes of the tree as it walks down. Whereas insert() eliminates 3-key nodes (moving keys up the tree) to make room for new keys, remove() eliminates 1-key nodes (pulling keys down the tree) so that a key can be removed from a leaf without leaving it empty. There are three ways 1-key nodes (except the root) are eliminated.

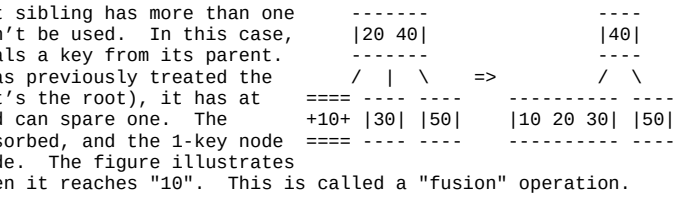
(1) When remove() encounters a 1-key node (except the root), it tries to steal a key from an adjacent sibling. But we can't just steal the sibling's key without violating the search tree invariant. This figure shows remove's action, called a "rotation", when it reaches "30".



We move a key from the sibling to the parent, and we move a key from the parent to the 1-key node. We also move a subtree S from the sibling to the 1-key node (now a 2-key node).

Goodrich & Tamassia call rotations "transfer" operations. Note that we can't steal a key from a non-adjacent sibling.

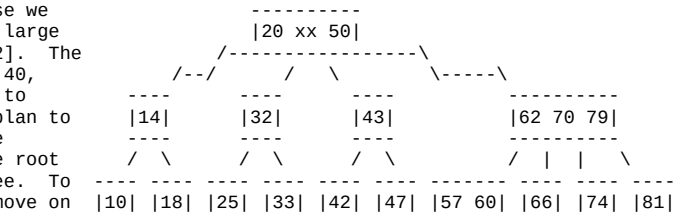
(2) If no adjacent sibling has more than one key, a rotation can't be used. In this case, the 1-key node steals a key from its parent. Since the parent was previously treated the same way (unless it's the root), it has at least two keys, and can spare one. The sibling is also absorbed, and the 1-key node becomes a 3-key node. The figure illustrates remove's action when it reaches "10". This is called a "fusion" operation.



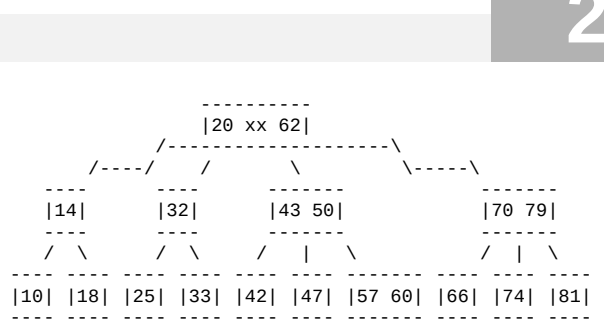
(3) If the parent is the root and contains only one key, and the sibling contains only one key, then the current 1-key node, its 1-key sibling, and the 1-key root are fused into one 3-key node that serves as the new root. The height of the tree decreases by one.

Eventually we reach a leaf. After we process the leaf, it has at least two keys (if there are at least two keys in the tree), so we can delete the key and still have one key in the leaf.

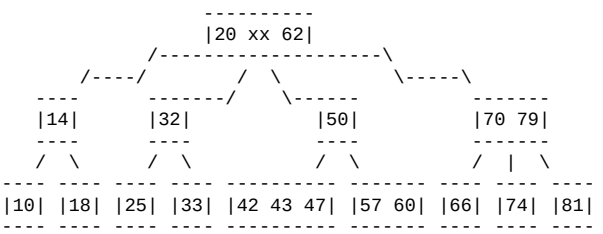
For example, suppose we remove 40 from the large tree depicted in [2]. The root node contains 40, which we mark "xx" to remind us that we plan to replace it with the smallest key in the root node's right subtree. To find that key, we move on to the 1-key node labeled 50. Following our rules for 1-key nodes, we fuse 50 with its sibling and parent to create a new 3-key root labeled "20 xx 50".



Next, we visit the node labeled 43. Again following our rules for 1-key nodes, we rotate 62 from a sibling to the root, and move 50 from the root to the node containing 43.



Finally, we move down to the node labeled 42. A different rule for 1-key nodes requires us to fuse the nodes labeled 42 and 47 into a 3-key node, stealing 43 from the parent node.



The last step is to remove 42 from the leaf and replace "xx" with 42.

Running Times

A 2-3-4 tree with height h has between 2^h and 4^h leaves. If n is the total number of entries (including entries in internal nodes), then n >= 2^(h+1) - 1. By taking the logarithm of both sides, we find that h is in O(log n).

The time spent visiting a 2-3-4 node is typically longer than in a binary search tree (because the nodes and the rotation and fusion operations are complicated), but the time per node is still in O(1).

The number of nodes visited is proportional to the height of the tree. Hence, the running times of the find(), insert(), and remove() operations are in O(h) and hence in O(log n), even in the worst case.

Compare this with the Theta(n) worst-case time of ordinary binary search trees.

Another Approach to Duplicate Keys

Rather than have a separate node for each entry, we might wish to collect all the entries that share a common key in one node. In this case, each node's entry becomes a list of entries. This simplifies the implementation of findAll(), which finds all the entries with a specified key. It also speeds up other operations by leaving fewer nodes in the tree data structure. Obviously, this is a change in the implementation, but not a change in the dictionary ADT.

This idea can be used with hash tables, binary search trees, and 2-3-4 trees.

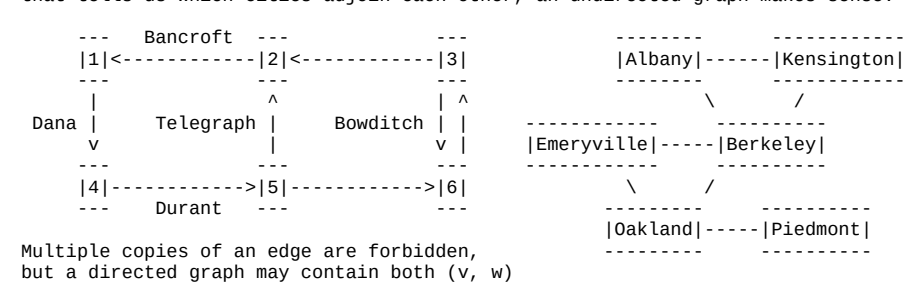
CS 61B: Lecture 28
Wednesday, April 2, 2014

GRAPHS
======
A graph G is a set V of vertices (sometimes called nodes), and a set E of edges (sometimes called arcs) that each connect two vertices together. To confuse you, mathematicians often use the notation G = (V, E). Here, "(V, E)" is an _ordered_pair_ of sets. This isn't as deep and meaningful as it sounds; some people just love formalism. The notation is convenient when you want to discuss several graphs with the same vertices; e.g. G = (V, E) and T = (V, F).

Graphs come in two types: _directed_ and _undirected_. In a directed graph (or _digraph_ for short), every edge e is directed from some vertex v to some vertex w. We write "e = (v, w)" (also an ordered pair), and draw an arrow pointing from v to w. The vertex v is called the _origin_ of e, and w is the _destination_ of e.

In an undirected graph, edges have no favored direction, so we draw a curve connecting v and w. We still write e = (v, w), but now it's an unordered pair, which means that (v, w) = (w, v).

One application of a graph is to model a street map. For each intersection, define a vertex that represents it. If two intersections are connected by a length of street with no intervening intersection, define an edge connecting them. We might use an undirected graph, but if there are one-way streets, a directed graph is more appropriate. We can model a two-way street with two edges, one pointing in each direction. On the other hand, if we want a graph that tells us which cities adjoin each other, an undirected graph makes sense.



Multiple copies of an edge are forbidden, but a directed graph may contain both (v, w) and (w, v). Both types of graph can have _self-edges_ of the form (v, v), which connect a vertex to itself. (Many applications, like the two illustrated above, don't use these.)

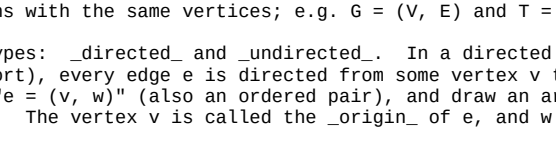
A _path_ is a sequence of vertices such that each adjacent pair of vertices is connected by an edge. If the graph is directed, the edges that form the path must all be aligned with the direction of the path. The _length_ of a path is the number of edges it traverses. Above, <4, 5, 6, 3> is a path of length 3. It is perfectly respectable to talk about a path of length zero, such as <2>. The _distance_ from one vertex to another is the length of the shortest path from one to the other.

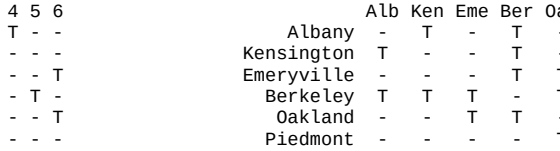
A graph is _strongly_connected_ if there is a path from every vertex to every other vertex. (This is just called _connected_ in undirected graphs.) Both graphs above are strongly connected.

The _degree_ of a vertex is the number of edges incident on that vertex. (Self-edges count just once in 61B.) Hence, Berkeley has degree 4, and Piedmont has degree 1. A vertex in a directed graph has an _indegree_ (the number of edges directed toward it) and an _outdegree_ (the number of edges directed away). Intersection 6 above has indegree 2 and outdegree 1.

Graph Representations

There are two popular ways to represent a graph. The first is an _adjacency_ _matrix_, a |V|-by-|V| array of boolean values (where |V| is the number of vertices in the graph). Each row and column represents a vertex of the graph. Set the value at row i, column j to true if (i, j) is an edge of the graph. If the graph is undirected (below right), the adjacency matrix is _symmetric_: row i, column j has the same value as row j, column i.





It should be clear that the maximum possible number of edges is |V|^2 for a directed graph, and slightly more than half that for an undirected graph. In many applications, however, the number of edges is much less than Theta(|V|^2). For instance, our maps above are _planar_graphs_ (graphs that can be drawn without edges crossing), and all planar graphs have O(|V|) edges. A graph is called _sparse_ if it has far fewer edges than the maximum possible number. ("Sparse" has no precise definition, but it usually implies that the number of edges is asymptotically smaller than |V|^2.)

For a sparse graph, an adjacency matrix representation is very wasteful. A more memory-efficient data structure for sparse graphs is the _adjacency_ _list_. An adjacency list is actually a collection of lists. Each vertex v maintains a list of the edges directed out from v. The standard list representations all work--arrays (below left), linked lists (below right), even search trees (because you can traverse one in linear time).

The total memory used by all the lists is Theta(|V| + |E|).

If your vertices have consecutive integer names, you can declare an array of lists, and find any vertex's list in O(1) time. If your vertices have names like "Albany," you can use a hash table to map names to lists. Each entry in the hash table uses a vertex name as a key, and a List object as the associated value. You can find the list for any label in O(1) average time.

An adjacency list is more space- and time-efficient than an adjacency matrix for a sparse graph, but less efficient for a _complete_graph_. A complete graph is a graph having every possible edge; that is, for every vertex u and every vertex v, (u, v) is an edge of the graph.

Graph Traversals

We'll look at two types of graph traversals, which can be used on either directed or undirected graphs to visit each vertex once. Depth-first search (DFS) starts at an arbitrary vertex and searches a graph as "deeply" as possible as early as possible. For example, if your graph is an undirected tree, DFS performs a preorder (or if you prefer, postorder) tree traversal.

Breadth-first search (BFS) starts by visiting an arbitrary vertex, then visits all vertices whose distance from the starting vertex is one, then all vertices whose distance from the starting vertex is two, and so on. If your graph is an undirected tree, BFS performs a level-order tree traversal.

In a graph, unlike a tree, there may be several ways to get from one vertex to another. Therefore, each vertex has a boolean field called "visited" that tells us if we have visited the vertex before, so we don't visit it twice. When we say we are "marking a vertex visited", we are setting its "visited" field to true.

Assume that we are traversing a strongly connected graph, thus there is a path from the starting vertex to every other vertex.

When DFS visits a vertex u, it checks every vertex v that can be reached by some edge (u, v). If v has not yet been visited, DFS visits it recursively.

```
public class Graph {
  // Before calling dfs(), set every "visited" flag to false; takes O(|V|) time
  public void dfs(Vertex u) {
    u.visit();                                // Do some unspecified thing to u
    u.visited = true;                              // Mark the vertex u visited
    for (each vertex v such that (u, v) is an edge in E) {
      if (!v.visited) {
        dfs(v);
      }
    }
  }
}
```

In this DFS pseudocode, a "visit()" method is defined that performs some action on a specified vertex. For instance, if we want to count the total population of the city graph above, "visit()" might add the population of the visited city to the grand total. The order in which cities are visited depends partly on their order in the adjacency lists.

The sequence of figures below shows the behavior of DFS on our street map, starting at vertex 1. A "V" is currently visited; an "x" is marked visited; a "*" is a vertex which we try to visit but discover has already been visited.

```
V<-2<-3  x<-2<-3  x<-2<-3  x<-V<-3  *<-V<-3  x<-x<-3  x<-x<-V  x<-*<-V  x<-x<-V
|  ^  ^  |  ^  ^  |  ^  ^  |  ^  ^  |  ^  ^  |  ^  ^  |  ^  ^  |  ^  ^  |  ^  ^
v  |  v  v  |  v  v  |  v  v  |  v  v  |  v  v  |  v  v  |  v  v  |  v  v  |  v
4->5->6  V->5->6  x->V->6  x->x->6  x->x->6  x->x->V  x->x->x  x->x->x  x->x->*
```

DFS runs in O(|V| + |E|) time if you use an adjacency list; O(|V|^2) time if you use an adjacency matrix. Hence, an adjacency list is asymptotically faster if the graph is sparse.

What's an application of DFS? Suppose you want to determine whether there is a path from a vertex u to another vertex v. Just do DFS from u, and see if v gets visited. (If not, you can't there from here.)

I'll discuss BFS in the next lecture.

More on the Running Time of DFS

Why does DFS on an adjacency list run in O(|V| + |E|) time?

The O(|V|) component comes up solely because we have to initialize all the "visited" flags to false (or at least construct an array of flags) before we start.

The O(|E|) component is trickier. Take a look at the "for" loop of the dfs() pseudocode above. How many times does it iterate? If the vertex u has outdegree d(u), then the loop iterates d(u) times. Different vertices have different degrees. Let the total degree D be the sum of the outdegrees of all the vertices in V.

$$D = \sum_{v \text{ in } V} d(v).$$

A call to dfs(u) takes O(d(u)) time, NOT counting the time for the recursive calls it makes to dfs(). A depth-first search never calls dfs() more than once on the same vertex, so the total running time of all the calls to dfs() is in O(D). How large is D?

- If G is a directed graph, then D = |E|, the number of edges.
- If G is an undirected graph with no self-edges, then D = 2|E|, because each edge offers a path out of two vertices.
- If G is an undirected graph with one or more self-edges, then D < 2|E|.

In all three cases, the running time of depth-first search is in O(|E|), NOT counting the time required to initialize the "visited" flags.

CS 61B: Lecture 29
Monday, April 7, 2014

GRAPHS (continued)
======
Breadth-first search (BFS) is a little more complicated than depth-first search, because it's not naturally recursive. We use a queue so that vertices are visited in order according to their distance from the starting vertex.

```
  public void bfs(Vertex u) {
    for (each vertex v in V) {                                  // O(|V|) time
      v.visited = false;
    }
    u.visit(null);                              // Do some unspecified thing to u
    u.visited = true;                                // Mark the vertex u visited
    q = new Queue();                                              // New queue...
    q.enqueue(u);                                    // ...initially containing u
    while (q is not empty) {                  // With adjacency list, O(|E|) time
      v = q.dequeue();
      for (each vertex w such that (v, w) is an edge in E) {
        if (!w.visited) {
          w.visit(v);                           // Do some unspecified thing to w
          w.visited = true;                          // Mark the vertex w visited
          q.enqueue(w);
        }
      }
    }
  }
```

```
public class Vertex {
  protected Vertex parent;
  protected int depth;
  protected boolean visited;

  public void visit(Vertex origin) {
    this.parent = origin;
    if (origin == null) {
      this.depth = 0;
    } else {
      this.depth = origin.depth + 1;
    }
  }
}
```

Notice that when we visit a vertex, we pass the edge's origin vertex as a parameter. This allows us to do a computation such as finding the distance of the vertex from the starting vertex, or finding the shortest path between them. The visit() method at right accomplishes both these tasks.

When an edge (v, w) is traversed to visit a Vertex w, the depth of w is set to the depth of v plus one, and v is set to become the _parent_ of w.

The sequence of figures below shows BFS running on the city adjacency graph (Albany, Kensington, Emeryville, Berkeley, Oakland, Piedmont) from last lecture, starting from Albany. A "V" is currently visited; a digit shows the depth of a vertex that is marked visited; a "*" is a vertex which we try to visit but discover has already been visited. Underneath each figure of the graph, I depict the queue and the current value of the variable "v" in bfs().

```
V-K  0-V  0-1  *-1  0-1  *-1  0-*  0-1  0-1  0-1  0-1  0-1  0-1  0-1  0-1  0-1  0-1
 \|   \|   \|   \|   \|   \|   \|   \|   \|   \|   \|   \|   \|   \|   \|   \|   \|
E-B  E-B  E-V  E-1  E-*  E-1  E-1  V-1  2-1  2-*  2-1  *-1  2-*  2-1  2-1  2-1
|/   |/   |/   |/   |/   |/   |/   |/   |/   |/   |/   |/   |/   |/   |/   |/
O-P  O-P  O-P  O-P  O-P  O-P  O-P  O-P  V-P  2-P  *-P  2-P  2-P  2-V  *-3  2-3

===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===
A    K    KB   B    B              E    EO   O    O              P
===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===  ===
     v=A  v=A  v=K  v=K  v=B  v=B  v=B  v=B  v=E  v=E  v=O  v=O  v=O  v=P
```

After we finish, we can find the shortest path from any vertex to the starting vertex by following the parent pointers (right). These pointers form a tree rooted at the starting vertex. Note that they point in the direction _opposite_ the search direction that got us there.

```
0<--1
     ^
      \
       \
2-->1
     ^
      /
       /
2<--3
```

Why does this work? The starting vertex is enqueued first, then all the vertices at a distance of 1 from the start, then all the vertices at a distance of 2, and so on. Why? When the starting vertex is dequeued, all the vertices at a distance of 1 are enqueued, but no other vertex is. When the depth-1 vertices are dequeued and processed, all the vertices at a distance of 2 are enqueued, because every vertex at a distance of 2 must be reachable by a single edge from some vertex at a distance of 1. No other vertex is enqueued, because every vertex at a distance less than 2 has been marked, and every vertex at a distance greater than 2 is not reachable by a single edge from some vertex at a distance of 1.

Recommendation: pull out a piece of paper, draw a graph and a program stack, and simulate BFS, with you acting as the computer and executing bfs() line by line. You will understand it much better after taking the time to do this.

BFS, like DFS, runs in O(|V| + |E|) time if you use an adjacency list; O(|V|^2) time if you use an adjacency matrix.

Weighted Graphs

A weighted graph is a graph in which each edge is labeled with a numerical weight. A weight might express the distance between two nodes, the cost of moving from one to the other, the resistance between two points in an electrical circuit, or many other things.

In an adjacency matrix, each weight is stored in the matrix. Whereas an unweighted graph uses an array of booleans, a weighted graph uses an array of ints, doubles, or some other numerical type. Edges missing from the graph can be represented by a special number like Integer.MIN_VALUE, at the cost of declaring that number invalid as an edge weight. (If you want to permit every int to be a valid edge weight, you might use an additional array of booleans as well.)

In an adjacency list, recall that each edge is represented by a listnode. Each listnode must be enlarged to include a weight, in addition to the reference to the destination vertex. (If you're using an array implementation of lists, you'll need two separate arrays: one for weights, and one for destinations.)

There are two particularly common problems involving weighted graphs. One is the _shortest_path_problem_. Suppose a graph represents a highway map, and each road is labeled with the amount of time it takes to drive from one interchange to the next. What's the fastest way to drive from Berkeley to Los Angeles? A shortest path algorithm will tell us. You'll learn several of these algorithms if you take CS 170.

The second problem is constructing a _minimum_spanning_tree_. Suppose that you're wiring a house for electricity. Each node of the graph represents an outlet, or the source of electricity. Every outlet needs to be connected to the source, but not necessarily directly--possibly routed via another outlet. The edges of the graph are labeled with the length of wire you'll need to connect one node to another. How do you connect all the nodes together with the shortest length of wire?

Kruskal's Algorithm for Finding Mimumum Spanning Trees

Let G = (V, E) be an undirected graph. A _spanning_tree_ T = (V, F) of G is a graph containing the same vertices as G, and |V| - 1 edges of G that form a tree. (Hence, there is exactly one path between any two vertices of T.)

If G is not connected, it has no spanning tree, but we can instead compute a _spanning_forest_, or collection of trees, having one tree for each connected component of G.

If G is weighted, then a _minimum_spanning_tree_ T of G is a spanning tree of G whose total weight (summed over all edges of T) is minimal. In other words, no other spanning tree of G has a smaller total weight.

Kruskal's algorithm computes the mimimum spanning tree of G as follows.

[1] Create a new graph T with the same vertices as G, but no edges (yet).
[2] Make a list of all the edges in G.
[3] Sort the edges by weight, from least to greatest.
[4] Iterate through the edges in sorted order.
    For each edge (u, w):
[4a] If u and w are not connected by a path in T, add (u, w) to T.

Because this algorithm never adds (u, w) if some path already connects u and w, T is guaranteed to be a tree (if G is connected) or a forest (if G is not).

Why is T a minimum spanning tree in the end? Suppose the algorithm is considering adding an edge (u, w) to T, and there is not yet a path connecting u to w. Let U be the set of vertices in T that are connected (so far) to u, and let W be a set containing all the other vertices, including w. Let the _bridge_edges_ be any edges in G that have one end vertex in U and one end vertex in W. Any spanning tree must contain at least one of these bridge edges. As long as we choose a bridge edge with the least weight, we are safe. (There may be several bridge edges with the same least weight, in which case it doesn't matter which one we choose.)

Because we go through the edges of G in order by weight, (u, w) must have the least weight, because it's the first edge we encountered connecting U to W. (See Goodrich and Tamassia page 649 for a proof that choosing the bridge edge with least weight is always the right thing to do.)

What is the running time of Kruskal's algorithm? As we'll discover in the next two lectures, sorting |E| edges takes O(|E| log |E|) time. The tricky part is, in [4a], determining whether u and w are already connected by a path. The simplest way to do this is by doing a depth-first search on T starting at u, and seeing if we visit w. But if we do that, Kruskal's algorithm might take Theta(|E| |V|)) time.

We can do better. In Lecture 33, we'll learn how to solve that problem quickly, so that all the iterations of [4a] together take less than O(|E| log |E|) time.

If we use an adjacency list, the running time is in O(|V| + |E| log |E|). But |E| < |V|^2, so log |E| < 2 log |V|. Therefore, Kruskal's algorithm runs in O(|V| + |E| log |V|) time.

If we use an adjacency matrix, the running time is in O(|V|^2 + |E| log |E|), because it takes Theta(|V|^2) time simply to make a list of all the edges.

CS 61B: Lecture 30
Wednesday, April 9, 2014

SORTING
=======
The need to sort numbers, strings, and other records arises frequently. The entries in any modern phone book were sorted by a computer. Databases have features that sort the records returned by a query, ordered according to any field the user desires. Google sorts your query results by their "relevance". We've seen that Kruskal's algorithm uses sorting. So do hundreds of other algorithms.

Sorting is perhaps the simplest fundamental problem that offers a huge variety of algorithms, each with its own inherent advantages and disadvantages. We'll study and compare eight sorting algorithms.

Insertion Sort

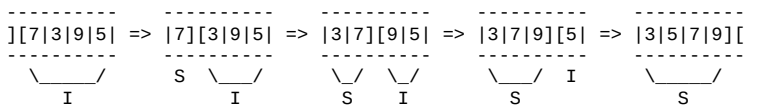
Insertion sort is very simple and runs in O(n^2) time. We employ a list S, and maintain the invariant that S is sorted.

```
  Start with an empty list S and the unsorted list I of n input items.
  for (each item x in I) {
    insert x into the list S, positioned so that S remains in sorted order.
  }
```

S may be an array or a linked list. If S is a linked list, then it takes Theta(n) worst-case time to find the right position to insert each item. If S is an array, we can find the right position in O(log n) time by binary search, but it takes Theta(n) worst-case time to shift the larger items over to make room for the new item. In either case, insertion sort runs in Theta(n^2) worst-case time--but for a different reason in each case.

If S is an array, one of the nice things about insertion sort is that it's an in-place sort. An _in-place_sort_ is a sorting algorithm that keeps the sorted items in the same array that initially held the input items. Besides the input array, it uses only O(1) or perhaps O(log n) additional memory.

To do an in-place insertion sort, we partition the array into two pieces: the left portion (initially empty) holds S, and the right portion holds I. With each iteration, the dividing line between S and I moves one step to the right.



If the input list I is "almost" sorted, insertion sort can be as fast as Theta(n)--if the algorithm starts its search from the _end_ of S. In this case, the running time is proportional to n plus the number of _inversions_. An inversion is a pair of keys j < k such that j appears after k in I. I could have anywhere from zero to n (n - 1) / 2 inversions.

If S is a balanced search tree (like a 2-3-4 tree or splay tree), then the running time is in O(n log n); but that's not what computer scientists mean when they discuss "insertion sort." This is our first O(n log n) sorting algorithm, but we'll pass it by for others that use less memory and have smaller constants hidden in the asymptotic running time bounds.

Selection Sort

Selection sort is equally simple, and also runs in quadratic time. Again we employ a list S, and maintain the invariant that S is sorted. Now, however, we walk through I and pick out the smallest item, which we append to the end of S.

```
  Start with an empty list S and the unsorted list I of n input items.
  for (i = 0; i < n; i++) {
    Let x be the item in I having smallest key.
    Remove x from I.
    Append x to the end of S.
  }
```

Whether S is an array or linked list, finding the smallest item takes Theta(n) time, so selection sort takes Theta(n^2) time, even in the best case! Hence, it's even worse than insertion sort.

If S is an array, we can do an in-place selection sort. After finding the item in I having smallest key, swap it with the first item in I, as shown here.

If I is a data structure faster than an array, we call it...

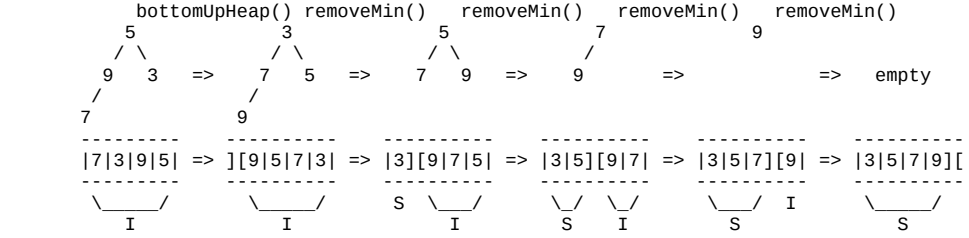
Heapsort

Heapsort is a selection sort in which I is a heap.

```
  Start with an empty list S and an unsorted list I of n input items.
  toss all the items in I onto a heap h (ignoring the heap-order property).
  h.bottomUpHeap();                          // Enforces the heap-order property
  for (i = 0; i < n; i++) {
    x = h.removeMin();
    Append x to the end of S.
  }
```

bottomUpHeap() runs in linear time, and each removeMin() takes O(log n) time. Hence, heapsort is an O(n log n)-time sorting algorithm.

There are several ways to do heapsort in place; I'll describe just one. Maintain the heap _backward_ at the _end_ of the array. This makes the indexing a little more complicated, but not substantially so. As items are removed from the heap, the heap shrinks toward the end of the array, making room to add items to the end of S.



Heapsort is excellent for sorting arrays, but it is an awkward choice for linked lists. The easiest way to heapsort a linked list is to create a new array of n references to the listnodes. Sort the array of references (using the keys in the listnodes for comparisons). When the array is sorted, link all the listnodes together into a sorted list.

The array of references uses extra memory. There is another O(n log n) algorithm that can sort linked lists using very little additional memory.

Mergesort

Mergesort is based on the observation that it's possible to merge two sorted lists into one sorted list in linear time. In fact, we can do it with queues:

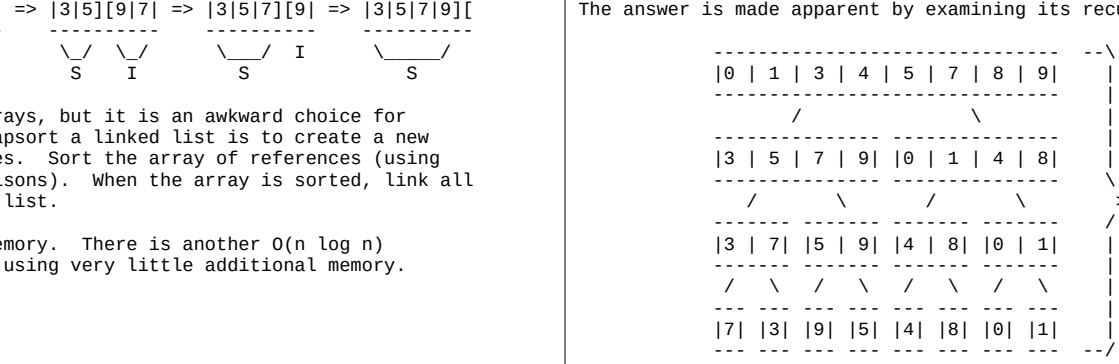
```
  Let Q1 and Q2 be two sorted queues.  Let Q be an empty queue.
  while (neither Q1 nor Q2 is empty) {
    item1 = Q1.front();
    item2 = Q2.front();
    move the smaller of item1 and item2 from its present queue to end of Q.
  }
  concatenate the remaining non-empty queue (Q1 or Q2) to the end of Q.
```

The merge routine is a kind of selection sort. At each iteration, it chooses the item having smallest key from the two input lists, and appends it to the output list. Since the two input lists are sorted, there are only two items to test, so each iteration takes constant time. Hence, merging takes O(n) time.

Mergesort is a recursive divide-and-conquer algorithm, in which the merge routine is what allows us to reunite what we divided:

```
  Start with the unsorted list I of n input items.
  Break I into two halves I1 and I2, having ceiling(n/2) and floor(n/2) items.
  Sort I1 recursively, yielding the sorted list S1.
  Sort I2 recursively, yielding the sorted list S2.
  Merge S1 and S2 into a sorted list S.
```

The recursion bottoms out at one-item lists. How long does mergesort take? The answer is made apparent by examining its recursion tree.



(Note that this tree is not a data structure. It's the structure of a sequence of recursive calls, like a game tree.)

Each level of the tree involves O(n) operations, and there are O(log n) levels. Hence, mergesort runs in O(n log n) time.

What makes mergesort a memory-efficient algorithm for sorting linked lists makes it a memory-inefficient algorithm for sorting arrays. Unlike the other sorting algorithms we've considered, mergesort is not an in-place algorithm. There is no reasonably efficient way to merge two arrays in place. Instead, use an extra array of O(n) size to temporarily hold the result of a merge.

CS61B: Lecture 31
Wednesday, April 9, 2014

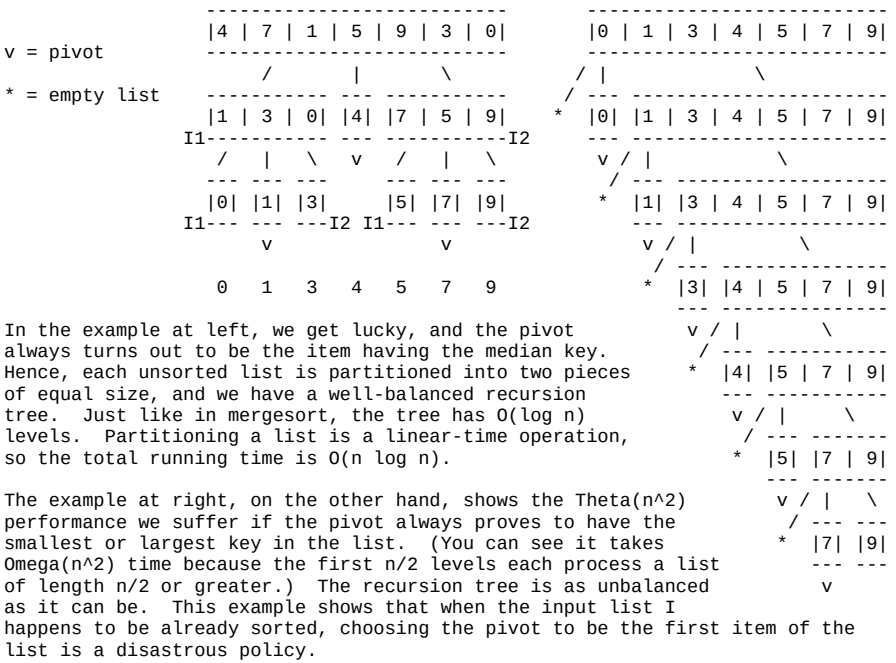
QUICKSORT
=========
Quicksort is a recursive divide-and-conquer algorithm, like mergesort. Quicksort is in practice the fastest known comparison-based sort for arrays, even though it has a Theta(n^2) worst-case running time. If properly designed, however, it virtually always runs in O(n log n) time. On arrays, this asymptotic bound hides a constant smaller than mergesort's, but mergesort is often slightly faster for sorting linked lists.

Given an unsorted list I of items, quicksort chooses a "pivot" item v from I, then puts each item of I into one of two unsorted lists, depending on whether its key is less or greater than v's key. (Items whose keys are equal to v's key can go into either list; we'll discuss this issue later.)

Start with the unsorted list I of n input items.
Choose a pivot item v from I.
Partition I into two unsorted lists I1 and I2.
- I1 contains all items whose keys are smaller than v's key.
- I2 contains all items whose keys are larger than v's key.
- Items with the same key as v can go into either list.
- The pivot v, however, does not go into either list.

Sort I1 recursively, yielding the sorted list S1.
Sort I2 recursively, yielding the sorted list S2.
Concatenate S1, v, and S2 together, yielding a sorted list S.

The recursion bottoms out at one-item and zero-item lists. (Zero-item lists can arise when the pivot is the smallest or largest item in its list.) How long does quicksort take? The answer is made apparent by examining several possible recursion trees. In the illustrations below, the pivot v is always chosen to be the first item in the list.



In the example at left, we get lucky, and the pivot always turns out to be the item having the median key. Hence, each unsorted list is partitioned into two pieces of equal size, and we have a well-balanced recursion tree. Just like in mergesort, the tree has O(log n) levels. Partitioning a list is a linear-time operation, so the total running time is O(n log n).

The example at right, on the other hand, shows the Theta(n^2) performance we suffer if the pivot always proves to have the smallest or largest key in the list. (You can see it takes Omega(n^2) time because the first n/2 levels each process a list of length n/2 or greater.) The recursion tree is as unbalanced as it can be. This example shows that when the input list I happens to be already sorted, choosing the pivot to be the first item of the list is a disastrous policy.

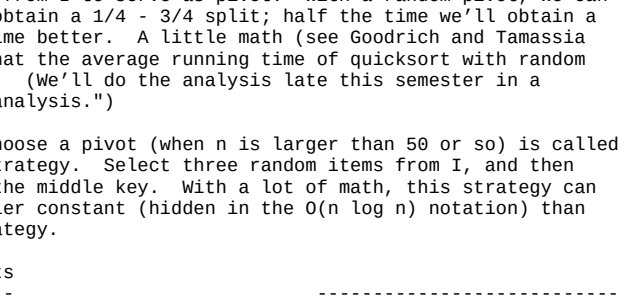
Choosing a Pivot

We need a better way to choose a pivot. A respected, time-tested method is to randomly select an item from I to serve as pivot. With a random pivot, we can expect "on average" to obtain a 1/4 - 3/4 split; half the time we'll obtain a worse split, half the time better. A little math (see Goodrich and Tamassia Section 11.2.1) shows that the average running time of quicksort with random pivots is in O(n log n). (We'll do the analysis late this semester in a lecture on "Randomized analysis.")

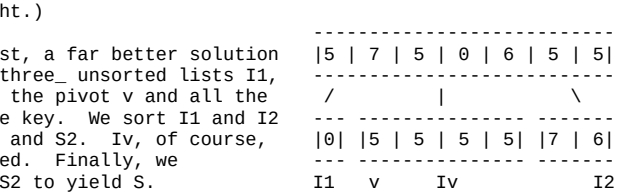
An even better way to choose a pivot (when n is larger than 50 or so) is called the "median-of-three" strategy. Select three random items from I, and then choose the item having the middle key. With a lot of math, this strategy can be shown to have a smaller constant (hidden in the O(n log n) notation) than the one-random-item strategy.

Quicksort on Linked Lists

I deliberately left unresolved the question of what to do with items that have the same key as the pivot. Suppose we put all the items having the same key as v into the list I1. If we try to sort a list in which every single item has the same key, then _every_ item will go into list I1, and quicksort will have quadratic running time! (See illustration at right.)



When sorting a linked list, a far better solution is to partition I into _three_ unsorted lists I1, I2, and Iv. Iv contains the pivot v and all the other items with the same key. We sort I1 and I2 recursively, yielding S1 and S2. Iv, of course, does not need to be sorted. Finally, we concatenate S1, Iv, and S2 to yield S.



This strategy is quite fast if there are a large number of duplicate keys, because the lists called "Iv" (at each level of the recursion tree) require no further sorting or manipulation.

Unfortunately, with linked lists, selecting a pivot is annoying. With an array, we can read a randomly chosen pivot in constant time; with a linked list we must walk half-way through the list on average, increasing the constant in our running time. However, if we restrict ourselves to pivots near the beginning of the linked list, we risk quadratic running time (for instance, if I is already in sorted order, or nearly so), so we have to pay the price. (If you are clever, you can speed up your implementation by choosing random pivots during the partitioning step for the _next_ round of partitioning.)

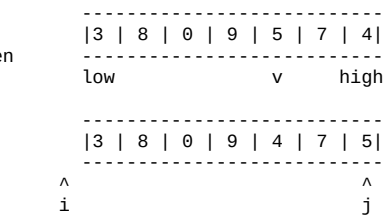
Quicksort on Arrays

Quicksort shines for sorting arrays. In-place quicksort is very fast. But a fast in-place quicksort is tricky to code. It's easy to write a buggy or quadratic version by mistake. Goodrich and Tamassia did.

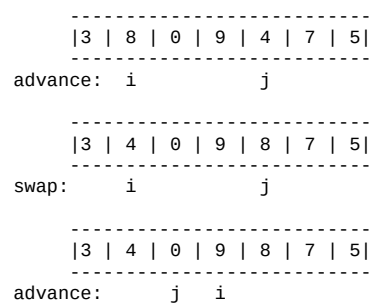
Suppose we have an array a in which we want to sort the items starting at a[low] and ending at a[high]. We choose a pivot v and move it out of the way by swapping it with the last item, a[high].

We employ two array indices, i and j. i is initially "low - 1", and j is initially "high", so that i and j sandwich the items to be sorted (not including the pivot). We will enforce the following invariants.
- All items at or left of index i have a key <= the pivot's key.
- All items at or right of index j have a key >= the pivot's key.

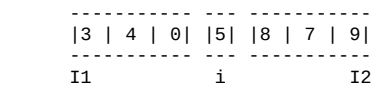
To partition the array, we advance the index i until it encounters an item whose key is greater than or equal to the pivot's key; then we decrement the index j until it encounters an item whose key is less than or equal to the pivot's key. Then, we swap the items at i and j. We repeat this sequence until the indices i and j meet in the middle. Then, we move the pivot back into the middle (by swapping it with the item at index i).



An example is given at right. The randomly selected pivot, whose key is 5, is moved to the end of the array by swapping it with the last item. The indices i and j are created. i advances until it reaches an item whose key is >= 5, and j retreats until it reaches an item whose key is <= 5. The two items are swapped, and i advances and j retreats again. After the second advance/retreat, i and j have crossed paths, so we do not swap their items. Instead, we swap the pivot with the item at index i, putting it between the lists I1 and I2 where it belongs.



What about items having the same key as the pivot? Handling these is particularly tricky. We'd like to put them on a separate list (as we did for linked lists), but doing that in place is too complicated. As I noted previously, if we put all these items into the list I1, we'll have quadratic running time when all the keys in the array are equal, so we don't want to do that either.



The solution is to make sure each index, i and j, stops whenever it reaches a key equal to the pivot. Every key equal to the pivot (except perhaps one, if we end with i = j) takes part in one swap. Swapping an item equal to the pivot may seem unnecessary, but it has an excellent side effect: if all the items in the array have the same key, half these items will go into I1, and half into I2, giving us a well-balanced recursion tree. (To see why, try running the pseudocode below on paper with an array of equal keys.) WARNING: The code on page 530 of Goodrich and Tamassia gets this WRONG. Their implementation has quadratic running time when all the keys are equal.

```
public static void quicksort(Comparable[] a, int low, int high) {
  // If there's fewer than two items, do nothing.
  if (low < high) {
    int pivotIndex = random number from low to high;
    Comparable pivot = a[pivotIndex];
    a[pivotIndex] = a[high];                        // Swap pivot with last item
    a[high] = pivot;

    int i = low - 1;
    int j = high;
    do {
      do { i++; } while (a[i].compareTo(pivot) < 0);
      do { j--; } while ((a[j].compareTo(pivot) > 0) && (j > low));
      if (i < j) {
        swap a[i] and a[j];
      }
    } while (i < j);

    a[high] = a[i];
    a[i] = pivot;                   // Put pivot in the middle where it belongs
    quicksort(a, low, i - 1);                    // Recursively sort left list
    quicksort(a, i + 1, high);                  // Recursively sort right list
  }
}
```

Can the "do { i++ }" loop walk off the end of the array and generate an out-of-bounds exception? No, because a[high] contains the pivot, so i will stop advancing when i == high (if not sooner). There is no such assurance for j, though, so the "do { j-- }" loop explicitly tests whether "j > low" before retreating.

Postscript

The journal "Computing in Science & Engineering" did a poll of experts to make a list of the ten most important and influential algorithms of the twentieth century, and it published a separate article on each of the ten algorithms. Quicksort is one of the ten, and it is surely the simplest algorithm on the list. Quicksort's inventor, Sir C. A. R. "Tony" Hoare, received the ACM Turing Award in 1980 for his work on programming languages, and was conferred the title of Knight Bachelor in March 2000 by Queen Elizabeth II for his contributions to "Computing Science."

CS61B: Lecture 33
Wednesday, April 16, 2014

Today's reading: Goodrich & Tamassia, Section 11.4.

DISJOINT SETS
=============
A _disjoint_sets_ data structure represents a collection of sets that are _disjoint_: that is, no item is found in more than one set. The collection of disjoint sets is called a _partition_, because the items are partitioned among the sets.

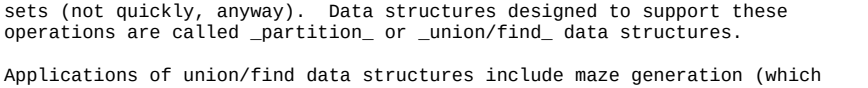
Moreover, we work with a _universe_ of items. The universe is made up of all of the items that can be a member of a set. Every item is a member of exactly one set.

For example, suppose the items in our universe are corporations that still exist today or were acquired by other corporations. Our sets are corporations that still exist under their own name. For instance, "Microsoft," "Forethought," and "Web TV" are all members of the "Microsoft" set.

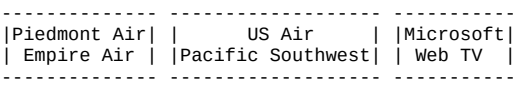
We will limit ourselves to two operations. The first is called a _union_ operation, in which we merge two sets into one. The second is called a _find_ query, in which we ask a question like, "What corporation does Web TV belong to today?" More generally, a "find" query takes an item and tells us which set it is in. We will not support operations that break a set up into two or more sets (not quickly, anyway). Data structures designed to support these operations are called _partition_ or _union/find_ data structures.

Applications of union/find data structures include maze generation (which you'll do in Homework 9) and Kruskal's algorithm for computing the minimum spanning tree of a graph (which you'll implement in Project 3).

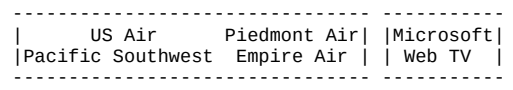
Union/find data structures begin with every item in a separate set.



The query "find(Empire Air)" returns "Empire Air". Suppose we take the union of Piedmont Air and Empire Air and called the resulting corporation Piedmont Air. Similarly, we unite Microsoft with Web TV and US Air with Pacific SW.



The query "find(Empire Air)" now returns "Piedmont Air". Suppose we further unite US Air with Piedmont Air.



The query "find(Empire Air)" now returns "US Air". When Microsoft takes over US Air, everything will be in one set and no further mergers will be possible.

List-Based Disjoint Sets and the Quick-Find Algorithm

The obvious data structure for disjoint sets looks like this.
- Each set references a list of the items in that set.
- Each item references the set that contains it.

With this data structure, find operations take O(1) time; hence, we say that list-based disjoint sets use the _quick-find_ algorithm. However, union operations are slow, because when two sets are united, we must walk through one set and relabel all the items so that they reference the other set.

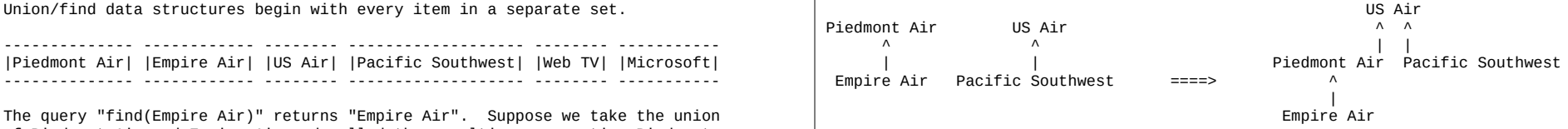
Time prevents us from analyzing this algorithm in detail (but see Goodrich and Tamassia, Section 11.4.3). Instead, let's move on to the less obvious but flatly superior _quick-union_ algorithm.

Tree-Based Disjoint Sets and the Quick-Union Algorithm

In tree-based disjoint sets, union operations take O(1) time, but find operations are slower. However, for any sequence of union and find operations, the quick-union algorithm is faster overall than the quick-find algorithm.

To support fast unions, each set is stored as a general tree. The quick-union data structure comprises a _forest_ (a collection of trees), in which each item is initially the root of its own tree; then trees are merged by union operations. The quick-union data structure is simpler than the general tree structures you have studied so far, because there are no child or sibling references. Every node knows only its parent, and you can only walk up the tree. The true identity of each set is recorded at its root.

Union is a simple O(1) time operation: we simply make the root of one set become a child of the root of the other set. For example, when we form the union of US Air and Piedmont Air:



US Air becomes a set containing four members. However, finding the set to which a given item belongs is not a constant-time operation.

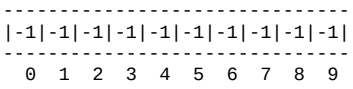
The find operation is performed by following the chain of parent references from an item to the root of its tree. For example, find(Empire Air) will follow the path of references until it reaches US Air. The cost of this operation is proportional to the item's depth in the tree.

These are the basic union and find algorithms, but we'll consider two optimizations that make finds faster. One strategy, called union-by-size, helps the union operation to build shorter trees. The second strategy, called path compression, gives the find operation the power to shorten trees.

Union-by-size is a strategy to keep items from getting too deep by uniting sets intelligently. At each root, we record the size of its tree (i.e. the number of nodes in the tree). When we unite two trees, we make the smaller tree a subtree of the larger one (breaking ties arbitrarily).

Implementing Quick-Union with an Array

Suppose the items are non-negative integers, numbered from zero. We'll use an array to record the parent of each item. If an item has no parent, we'll record the size of its tree. To distinguish it from a parent reference, we'll record the size s as the negative number -s. Initially, every item is the root of its own tree, so we set every array element to -1.



The forest illustrated at left below is represented by the array at right.

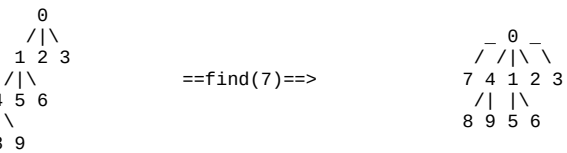
This is a slightly kludgy way to implement tree-based disjoint sets, but it's fast (in terms of the constant hidden in the asymptotic notation).

Let root1 and root2 be two items that are roots of their respective trees. Here is code for the union operation with the union-by-size strategy.

```
  public void union(int root1, int root2) {
    if (array[root2] < array[root1]) {               // root2 has larger tree
      array[root2] += array[root1];        // update # of items in root2's tree
      array[root1] = root2;                        // make root2 parent of root1
    } else {                                      // root1 has equal or larger tree
      array[root1] += array[root2];        // update # of items in root1's tree
      array[root2] = root1;                        // make root1 parent of root2
    }
  }
```

Path Compression

The find() method is equally simple, but we need one more trick to obtain the best possible speed. Suppose a sequence of union operations creates a tall tree, and we perform find() repeatedly on its deepest leaf. Each time we perform find(), we walk up the tree from leaf to root, perhaps at considerable expense. When we perform find() the first time, why not move the leaf up the tree so that it becomes a child of the root? That way, next time we perform find() on the same leaf, it will run much more quickly. Furthermore, why not do the same for _every_ node we encounter as we walk up to the root?



In the example above, find(7) walks up the tree from 7, discovers that 0 is the root, and then makes 0 be the parent of 4 and 7, so that future find operations on 4, 7, or their descendants will be faster. This technique is called _path_compression_.

Let x be an item whose set we wish to identify. Here is code for find, which returns the identity of the item at the root of the tree.

```
  public int find(int x) {
    if (array[x] < 0) {
      return x;                          // x is the root of the tree; return it
    } else {
      // Find out who the root is; compress path by making the root x's parent
      array[x] = find(array[x]);
      return array[x];                                    // Return the root
    }
  }
```

Naming Sets

Union-by-size means that if Microsoft acquires US Air, US Air will be the root of the tree, even though the new conglomorate might still be called Microsoft. What if we want some control over the names of the sets when we perform union() operations?

The solution is to maintain an additional array that maps root items to set names (and perhaps vice versa, depending on the application's needs). For instance, the array "name" might map 0 to Microsoft. We must modify the union() method so that when it unites two sets, it assigns the union an appropriate name.

For many applications, however, we don't care about the name of a set at all; we only want to know if two items x and y are in the same set. This is true in both Homework 9 and Project 3. You only need to run find(x), run find(y), and check if the two roots are the same.

Running Time of Quick-Union

Union operations obviously take Theta(1) time. (Look at the code--no loops or recursion.)

If we use union-by-size, a single find operation can take Theta(log u) worst-case time, where u is the number of union operations that took place prior to the find. Path compression does not improve this worst-case time, but it improves the _average_ running time substantially--although a find operation can take Theta(log u) time, path compression will make that operation fast if you do it again. The average running time of find and union operations in the quick-union data structure is so close to a constant that it's hardly worth mentioning that, in a rigorous asymptotic sense, it's slightly slower.

The bottom line: a sequence of f find and u union operations (in any order and possibly interleaved) takes Theta(u + f alpha(f + u, u)) time in the worst case. alpha is an extremely slowly-growing function known as the _inverse_ _Ackermann_function_.

When I say "extremely slowly-growing function", I mean "comically slowly-growing function." The inverse Ackermann function is never larger than 4 for any values of f and u you could ever use (though it does grow arbitrarily large--for unimaginably gigantic values of f and u). Hence, for all practical purposes (but not on the Final Exam), you should think of quick-union as having find operations that run, on average, in constant time.

CS61B: Lecture 34
Wednesday, April 16, 2014

Today's reading: Goodrich & Tamassia, Sections 11.3.1 & 11.5.

SELECTION

Suppose that we want to find the kth smallest key in a list. In other words, we want to know which item has index j if the list is sorted (where j = k - 1). We could simply sort the list, then look up the item at index j. But if we don't actually need to sort the list, is there a faster way? This problem is called _selection_.

One example is finding the median of a set of keys. In an array of n items, we are looking for the item whose index is j = floor(n / 2) in the sorted list.

Quickselect

We can modify quicksort to perform selection for us. Observe that when we choose a pivot v and use it to partition the list into three lists I1, Iv, and I2, we know which of the three lists contains index j, because we know the lengths of I1 and I2. Therefore, we only need to search one of the three lists.

Here's the quickselect algorithm for finding the item at index j - that is, having the (j + 1)th smallest key.

```
  Start with an unsorted list I of n input items.
  Choose a pivot item v from I.
  Partition I into three unsorted lists I1, Iv, and I2.
    - I1 contains all items whose keys are smaller than v's key.
    - I2 contains all items whose keys are larger than v's.
    - Iv contains the pivot v.
    - Items with the same key as v can go into any of the three lists.
      (In list-based quickselect, they go into Iv; in array-based quickselect,
      they go into I1 and I2, just like in array-based quicksort.)
  if (j < |I1|) {
    Recursively find the item with index j in I1; return it.
  } else if (j < |I1| + |Iv|) {
    Return the pivot v.
  } else {   // j >= |I1| + |Iv|.
    Recursively find the item with index j - |I1| - |Iv| in I2; return it.
  }
```

The advantage of quickselect over quicksort is that we only have to make one recursive call, instead of two. Since we make at most _one_ recursive call at _every_ level of the recursion tree, quickselect is much faster than quicksort. I won't analyze quickselect here, but it runs in Theta(n) average time if we select pivots randomly.

We can easily modify the code for quicksort on arrays, presented in Lecture 31, to do selection. The partitioning step is done exactly according to the Lecture 31 pseudocode for array quicksort. Recall that when the partitioning stage finishes, the pivot is stored at index "i" (see the variable "i" in the array quicksort pseudocode). In the quickselect pseudocode above, just replace |I1| with i and |Iv| with 1.

A LOWER BOUND ON COMPARISON-BASED SORTING

Suppose we have a scrambled array of n numbers, with each number from 1...n occurring once. How many possible orders can the numbers be in?

The answer is n!, where n! = 1 * 2 * 3 * ... * (n-2) * (n-1) * n. Here's why: the first number in the array can be anything from 1...n, yielding n possibilities. Once the first number is chosen, the second number can be any one of the remaining n-1 numbers, so there are n * (n-1) possible choices of the first two numbers. The third number can be any one of the remaining n-2 numbers, yielding n * (n-1) * (n-2) possibilities for the first three numbers. Continue this reasoning to its logical conclusion.

Each different order is called a _permutation_ of the numbers, and there are n! possible permutations. (For Homework 9, you are asked to create a random permutation of maze walls.)

Observe that if n > 0,

$$n! = 1 * 2 * \ldots * (n-1) * n \leq \underbrace{n * n * n * \ldots * n * n * n}_{n} = n$$

and (supposing n is even)

$$n! = 1 * 2 * \ldots * (n-1) * n \geq \underbrace{\frac{n}{2} * \left(\frac{n}{2} + 1\right) * \ldots * (n-1) * n}_{n/2} \geq (n/2)$$

so n! is between (n/2)^(n/2) and n^n. Let's look at the logarithms of both these numbers: log((n/2)^(n/2)) = (n/2) log (n/2), which is in Theta(n log n), and log(n^n) = n log n. Hence, log(n!) is also in Theta(n log n).

A _comparison-based_sort_ is one in which all decisions are based on comparing keys (generally done by "if" statements). All actions taken by the sorting algorithm are based on the results of a sequence of true/false questions. All of the sorting algorithms we have studied are comparison-based.

Suppose that two computers run the _same_ sorting algorithm at the same time on two _different_ inputs. Suppose that every time one computer executes an "if" statement and finds it true, the other computer executes the same "if" statement and also finds it true; likewise, when one computer executes an "if" and finds it false, so does the other. Then both computers perform exactly the same data movements (e.g. swapping the numbers at indices i and j) in exactly the same order, so they both permute their inputs in _exactly_ the same way.

A correct sorting algorithm must generate a _different_ sequence of true/false answers for each different permutation of 1...n, because it takes a different sequence of data movements to sort each permutation. There are n! different permutations, thus n! different sequences of true/false answers.

If a sorting algorithm asks d true/false questions, it generates <= 2^d different sequences of true/false answers. If it correctly sorts every permutation of 1...n, then n! <= 2^d, so log_2 (n!) <= d, and d is in Omega(n log n). The algorithm spends Omega(d) time asking these d questions. Hence,

EVERY comparison-based sorting algorithm takes Omega(n log n) worst-case time.

This is an amazing claim, because it doesn't just analyze one algorithm. It says that of the thousands of comparison-based sorting algorithms that haven't even been invented yet, not one of them has any hope of beating O(n log n) time for all inputs of length n.

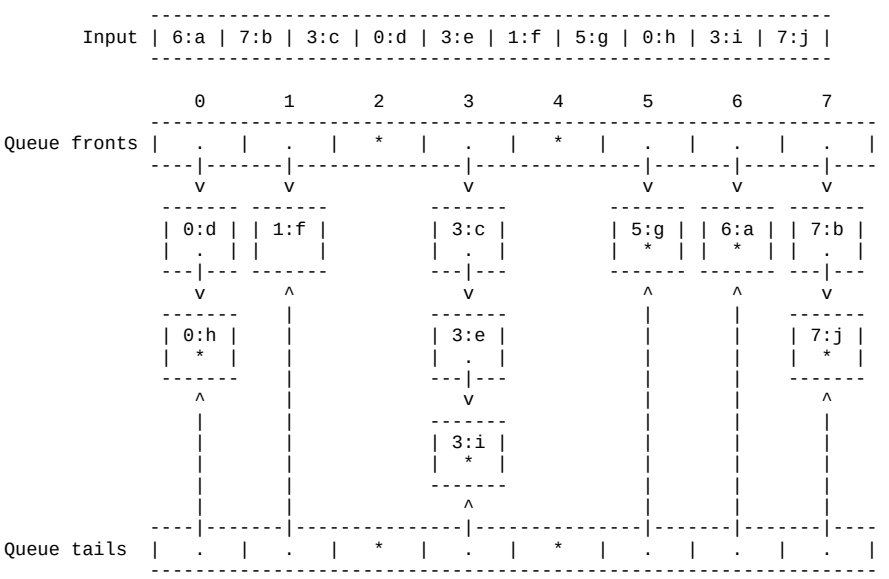
LINEAR-TIME SORTING
==========
However, there are faster sorting algorithms that can make q-way decisions for large values of q, instead of true/false (2-way) decisions. Some of these algorithms run in linear time.

Bucket Sort

_Bucket_sort_ works well when keys are distributed in a small range, e.g. from 0 to q - 1, and the number of items n is larger than, or nearly as large as, q. In other words, when q is in O(n).

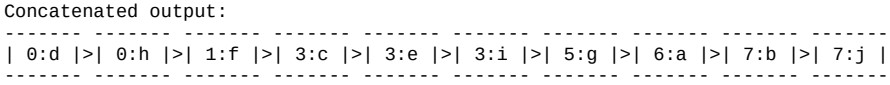
We allocate an array of q queues (or singly-linked lists with tail references, which are basically the same thing, but we only need the queue operations), numbered from 0 to q - 1. The queues are called _buckets_. We walk through the list of input items, and enqueue each item in the appropriate queue: an item with key i goes into queue i.

Each item illustrated here has a numerical key and an associated value.



When we're done, we concatenate all the queues together in order.

Concatenated output:



This data structure is _exactly_ like a hash table (plus tail references), but the hash code just maps the key i to bucket i, and there is no compression function because there is no need for compression.

Bucket sort takes Theta(q + n) time--in the best case and in the worst case. It takes Theta(q) time to initialize the buckets in the beginning and to concatenate them together in the end. It takes Theta(n) time to put all the items in their buckets.

If q is in O(n)--that is, the number of possible keys isn't much larger than the number of items we're sorting--then bucket sort takes Theta(n) time. How did we get around the Omega(n log n) lower bound on comparison-based sorting? Bucket sort is not comparison-based. We are making a q-way decision every time we decide which queue to put an item into, instead of the true/false decisions provided by comparisons and "if" statements.

Bucket sort (as I've described it here) is said to be _stable_. A sort is stable if items with equal keys come out in the same order they went in. For example, observe that 3:c, 3:e, and 3:i appear in the same order in the output above as they appeared in the input. Bucket sort is not the only stable sort we have seen; insertion sort, selection sort, and mergesort can all be implemented so that they are stable. The linked list version of quicksort we have seen can be stable, but the array version is decidedly not. Heapsort is never stable. (Actually, we can _make_ heapsort stable using a simple trick called a _secondary_key_, which I might describe later in the semester.)

Take note that bucket sort is ONLY appropriate when keys are distributed in a small range; i.e. q is in O(n). On Monday we'll study a sorting algorithm called _radix_sort_ that will fix that limitation. The stability of bucket sort will be important for radix sort.

CS 61B: Lecture 35
Monday, April 21, 2014

Today's reading: Goodrich & Tamassia, Sections 11.3.2.

Counting Sort

If the items we sort are naked keys, with no associated values, bucket sort can be simplified to become _counting_sort_. In counting sort, we use no queues at all; we need merely keep a count of how many copies of each key we have encountered. Suppose we sort 6 7 3 0 3 1 5 0 3 7:

	0	1	2	3	4	5	6	7
counts	2	1	0	3	0	1	1	2

When we are finished counting, it is straightforward to reconstruct the sorted keys from the counts: 0 0 1 3 3 3 5 6 7 7.

Counting Sort with Complete Items

Now let's go back to the case where we have complete items (key plus associated value). We can use a more elaborate version of counting sort. The trick is to use the counts to find the right index to move each item to.

Let x be an input array of objects with keys (and perhaps other information).

	0	1	2	3	4	5	6	7	8	9
x	.	.	.	.	.	.	.	.	.	.
	v	v	v	v	v	v	v	v	v	v
	6	7	3	0	3	1	5	0	3	7

Begin by counting the keys in x.

```
for (i = 0; i < x.length; i++) {
  counts[x[i].key]++;
}
```

Next, do a _scan_ of the "counts" array so that counts[i] contains the number of keys _less_than_ i.

	0	1	2	3	4	5	6	7
counts	0	2	3	3	6	6	7	8

```
total = 0;
for (j = 0; j < counts.length; j++) {
  c = counts[j];
  counts[j] = total;
  total = total + c;
}
```

Let y be the output array, where we will put the sorted objects. counts[i] tells us the first index of y where we should put items with key i. Walk through the array x and copy each item to its final position in y. When you copy an item with key k, you must increment counts[k] to make sure that the next item with key k goes into the next slot.

```
for (i = 0; i < x.length; i++) {
  y[counts[x[i].key]] = x[i];
  counts[x[i].key]++;
}
```

```
y |.|.|.|.|.|.|.|.|.|.|    counts | 0 | 2 | 3 | 3 | 6 | 6 | 8 | 8 |
  --------------|-----             ---------------------------------
                v
                6

y |.|.|.|.|.|.|.|.|.|.|    counts | 0 | 2 | 3 | 3 | 6 | 6 | 8 | 9 |
  --------------|-|---              ---------------------------------
                v v
                6 7

y |.|.|.|.|.|.|.|.|.|.|    counts | 0 | 2 | 3 | 4 | 6 | 6 | 8 | 9 |
  -------|-------|-|---              ---------------------------------
         v       v v
         3       6 7

y |.|.|.|.|.|.|.|.|.|.|    counts | 1 | 2 | 3 | 4 | 6 | 6 | 8 | 9 |
  -|-----|-------|-|---              ---------------------------------
   v     v       v v
   0     3       6 7

y |.|.|.|.|.|.|.|.|.|.|    counts | 1 | 2 | 3 | 5 | 6 | 6 | 8 | 9 |
  -|-----|-|-----|-|---              ---------------------------------
   v     v v     v v
   0     3 3     6 7

y |.|.|.|.|.|.|.|.|.|.|    counts | 1 | 3 | 3 | 5 | 6 | 6 | 8 | 9 |
  -|---|-|-|-----|-|---              ---------------------------------
   v   v v v     v v
   0   1 3 3     6 7

...

y |.|.|.|.|.|.|.|.|.|.|    counts | 2 | 3 | 3 | 6 | 6 | 7 | 8 | 10 |
  -|-|-|-|-|-|-|-|-|-|-            ---------------------------------
   v v v v v v v v v v
   0 0 1 3 3 3 5 6 7 7
```

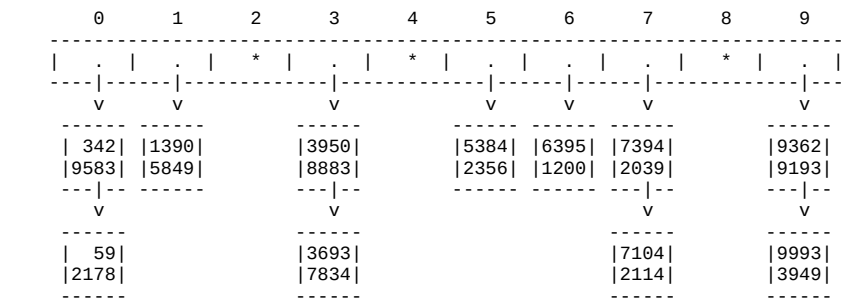
Bucket sort and counting sort both take O(q + n) time. If q is in O(n), then they take O(n) time. If you're sorting an array, counting sort is slightly faster and takes less memory than bucket sort, though it's a little harder to understand. If you're sorting a linked list, bucket sort is more natural, because you've already got listnodes ready to put into the buckets.

However, if q is not in O(n)--there are many more _possible_values_ for keys than keys--we need a more aggressive method to get linear-time performance.

Radix Sort

Suppose we want to sort 1,000 items in the range from 0 to 99,999,999. If we use bucket sort, we'll spend so much time initializing and concatenating empty queues we'll wish we'd used selection sort instead.

Instead of providing 100 million buckets, let's provide q = 10 buckets and sort on the first digit only. (A number less than 10 million is said to have a first digit of zero.) We use bucket sort or counting sort, treating each item as if its key is the first digit of its true key.



Once we've dealt all 1,000 items into ten queues, we could sort each queue recursively on the second digit; then sort the resulting queues on the third digit, and so on. Unfortunately, this tends to break the set of input items into smaller and smaller subsets, each of which will be sorted relatively inefficiently.

Instead, we use a clever but counterintuitive idea: we'll keep all the numbers together in one big pile throughout the sort; but we'll sort on the _last_ digit first, then the next-to-last, and so on up to the most significant digit.

The reason this idea works is because bucket sort and counting sort are stable. Hence, once we've sorted on the last digit, the numbers 55,555,552 and 55,555,558 will remain ever after in sorted order, because their other digits will be sorted stably. Consider an example with three-digit numbers:

```
Sort on 1s:    771 721 822 955 405   5 925 825 777  28 829
Sort on 10s:   405   5 721 822 925 825  28 829 955 771 777
Sort on 100s:    5  28 405 721 771 777 822 825 829 925 955
```

After we sort on the middle digit, observe that the numbers are sorted by their last two digits. After we sort on the most significant digit, the numbers are completely sorted.

Returning to our eight-digit example, we can do better than sorting on one decimal digit at a time. With 1,000 keys, sorting would likely be faster if we sort on two digits at a time (using a base, or _radix_, of q = 100) or even three (using a radix of q = 1,000). Furthermore, there's no need to use decimal digits at all; on computers, it's more natural to choose a power-of-two radix like q = 256. Base-256 digits are easier to extract from a key, because we can quickly pull out the eight bits that we need by using bit operators (which you'll study in detail in CS 61C).

Note that q is both the number of buckets we're using to sort, and the radix of the digit we use as a sort key during one pass of bucket or counting sort. "Radix" is a synonym for the base of a number, hence the name "radix sort."

How many passes must we perform? Each pass inspects log2 q bits of each key. If all the keys can be represented in b bits, the number of passes is ceiling(b / log2 q). So the running time of radix sort is in

$$O\left((n + q) \text{ ceiling}\left(\frac{b}{\log_2 q} \right) \right).$$

How should we choose the number of queues q? Let's choose q to be in O(n), so each pass of bucket sort or counting sort takes O(n) time. However, we want q to be large enough to keep the number of passes small. Therefore, let's choose q to be approximately n. With this choice, the number of passes is in O(1 + b / log2 n), and radix sort takes

$$O\left(n + \frac{b}{\log n} n\right) \text{ time.}$$

For many kinds of keys we might sort (like ints), b is technically a constant, and radix sort takes linear time. Even if the key length b tends to grow logarithmically with n (a reasonable model in many applications), radix sort runs in time linear in the total number of bits in all the keys together.

A practical, efficient choice is to make q equal to n rounded down to the next power of two. If we want to keep memory use low, however, we can make q equal to the square root of n, rounded to the nearest power of two. With this choice, the number of buckets is far smaller, but we only double the number of passes.

Postscript: Radix Sort Rocks (not examinable)

Linear-time sorts tend to get less attention than comparison-based sorts in most computer science classes and textbooks. Perhaps this is because the theory behind linear-time sorts isn't as interesting as for other algorithms. Nevertheless, the library sort routines for machines like Crays use radix sort, because it kicks major ass in the speed department.

Radix sort can be used not only with integers, but with almost any data that can be compared bitwise, like strings. The IEEE standard for floating-point numbers is designed to work with radix sort combined with a simple prepass and postpass (to flip the bits, except the sign bit, of each negative number).

Strings of different lengths can be sorted in time proportional to the total length of the strings. A first stage sorts the strings by their lengths. A second stage sorts the strings character by character (or several characters at a time), starting with the last character of the longest string and working backward to the first character of every string. We don't sort every string during every pass of the second stage; instead, a string is included in a pass only if it has a character in the appropriate place.

For instance, suppose we're sorting the strings CC, BA, CCAAA, BAACA, and BAABA. After we sort them by length, the next three passes sort only the last three strings by their last three characters, yielding CCAAA BAABA BAACA. The fifth pass is on the second character of each string, so we prepend the two-character strings to our list, yielding CC BA CCAAA BAABA BAACA. After sorting on the second and first characters, we end with

```
  BA BAABA BAACA CC CCAAA.
```

Observe that BA precedes BAABA and CC precedes CCAAA because of the stability of the sort. That's why we put the two-character strings before the five-character strings when we began the fifth pass.

CS61B: Lecture 36
Wednesday, April 23, 2014

Today's reading: Goodrich & Tamassia, Section 10.3.

SPLAY TREES

A splay tree is a type of balanced binary search tree. Structurally, it is identical to an ordinary binary search tree; the only difference is in the algorithms for finding, inserting, and deleting entries.

All splay tree operations run in O(log n) time _on_average_, where n is the number of entries in the tree. Any single operation can take Theta(n) time in the worst case. But any sequence of k splay tree operations, with the tree initially empty and never exceeding n items, takes O(k log n) worst-case time.

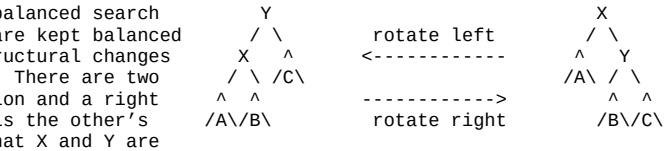
Although 2-3-4 trees make a stronger guarantee (_every_ operation on a 2-3-4 tree takes O(log n) time), splay trees have several advantages. Splay trees are simpler and easier to program. Because of their simplicity, splay tree insertions and deletions are typically faster in practice (sometimes by a constant factor, sometimes asymptotically). Find operations can be faster or slower, depending on circumstances.

Splay trees are designed to give especially fast access to entries that have been accessed recently, so they really excel in applications where a small fraction of the entries are the targets of most of the find operations.

Splay trees have become the most widely used basic data structure invented in the last 30 years, because they're the fastest type of balanced search tree for many applications.

Tree Rotations

Like many types of balanced search trees, splay trees are kept balanced with the help of structural changes called _rotations_. There are two types--a left rotation and a right rotation--and each is the other's reverse. Suppose that X and Y are binary tree nodes, and A, B, and C are subtrees. A rotation transforms either of the configurations illustrated above to the other. Observe that the binary search tree invariant is preserved: keys in A are less than or equal to X; keys in C are greater than or equal to Y; and keys in B are >= X and <= Y.



Rotations are also used in AVL trees and red-black trees, which are discussed by Goodrich and Tamassia, but are not covered in this course.

Unlike 2-3-4 trees, splay trees are not kept perfectly balanced, but they tend to stay reasonably well-balanced most of the time, thereby averaging O(log n) time per operation in the worst case (and sometimes achieving O(1) average running time in special cases).

Splay Tree Operations

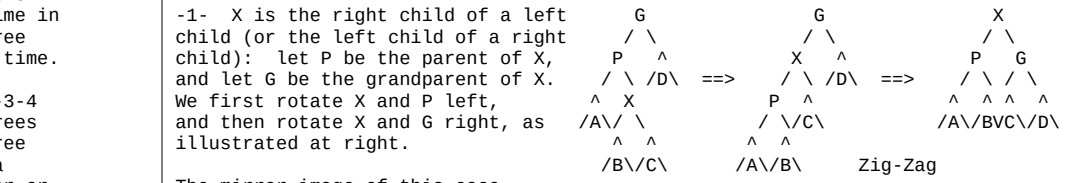
[1] Entry find(Object k);

The find() operation in a splay tree begins just like the find() operation in an ordinary binary search tree: we walk down the tree until we find the entry with key k, or reach a dead end (a node from which the next logical step leads to a null pointer).

However, a splay tree isn't finished its job. Let X be the node where the search ended, whether it contains the key k or not. We _splay_ X up the tree through a sequence of rotations, so that X becomes the root of the tree. Why? One reason is so that recently accessed entries are near the root of the tree, and if we access the same few entries repeatedly, accesses will be very fast. Another reason is because if X lies deeply down an unbalanced branch of the tree, the splay operation will improve the balance along that branch.

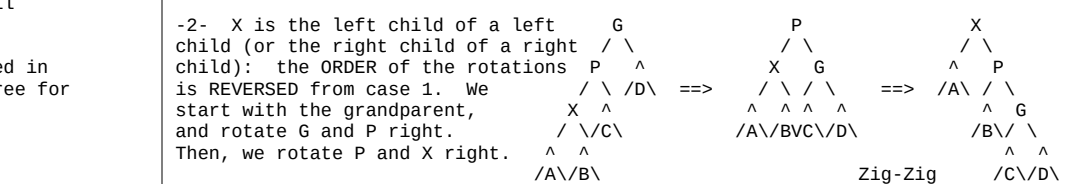
When we splay a node to the root of the tree, there are three cases that determine the rotations we use.

-1- X is the right child of a left child (or the left child of a right child): let P be the parent of X, and let G be the grandparent of X. We first rotate X and P left, and then rotate X and G right, as illustrated at right.



The mirror image of this case--where X is a left child and P is a right child--uses the same rotations in mirror image: rotate X and P right, then X and G left. Both the case illustrated above and its mirror image are called the "zig-zag" case.

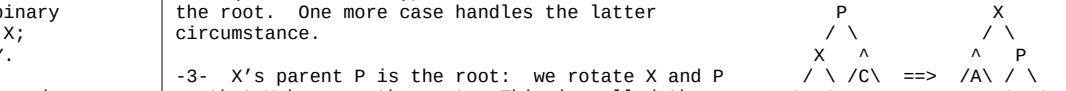
-2- X is the left child of a left child (or the right child of a right child): the ORDER of the rotations is REVERSED from case 1. We start with the grandparent, and rotate G and P right. Then, we rotate P and X right.



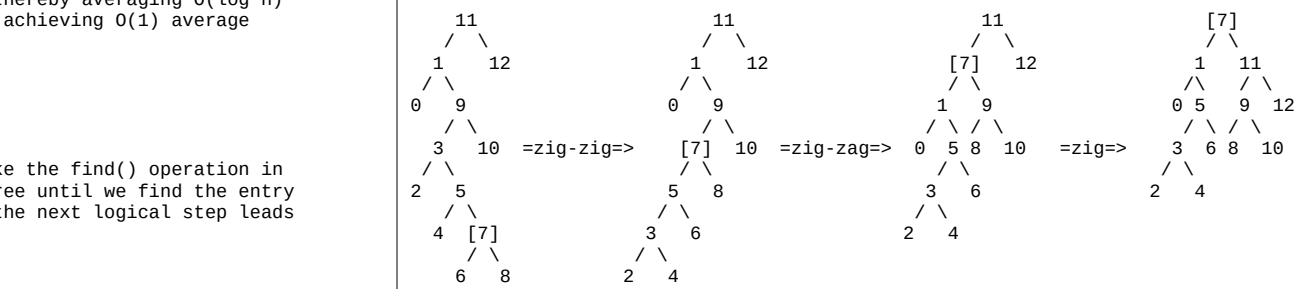
The mirror image of this case--where X and P are both right children--uses the same rotations in mirror image: rotate G and P left, then P and X left. Both the case illustrated above and its mirror image are called the "zig-zig" case.

We repeatedly apply zig-zag and zig-zig rotations to X; each pair of rotations raises X two levels higher in the tree. Eventually, either X will reach the root (and we're done), or X will become the child of the root. One more case handles the latter circumstance.

-3- X's parent P is the root: we rotate X and P so that X becomes the root. This is called the "zig" case.

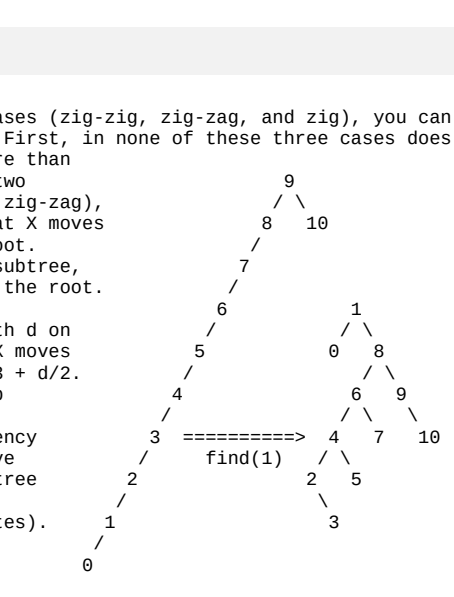


Here's an example of "find(7)". Note how the tree's balance improves.



By inspecting each of the three cases (zig-zig, zig-zag, and zig), you can observe a few interesting facts. First, in none of these three cases does the depth of a subtree increase by more than two. Second, every time X takes two steps toward the root (zig-zig or zig-zag), every node in the subtree rooted at X moves at least one step closer to the root. As more and more nodes enter X's subtree, more of them get pulled closer to the root.

A node that initially lies at depth d on the access path from the root to X moves to a final depth no greater than 3 + d/2. In other words, all the nodes deep down the search path have their depths roughly halved. This tendency of nodes on the access path to move toward the root prevents a splay tree from staying unbalanced for long (as the example at right illustrates).



[2] Entry min();
Entry max();

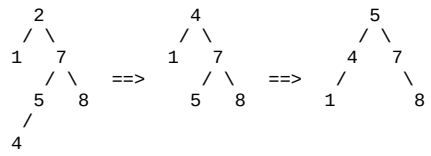
These methods begin by finding the entry with minimum or maximum key, just like in an ordinary binary search tree. Then, they splay the node containing the minimum or maximum key to the root.

[3] Entry insert(Object k, Object v);

insert() begins by inserting the new entry (k, v), just like in an ordinary binary search tree. Then, it splays the new node to the root.

[4] Entry remove(Object k);

An entry having key k is removed from the tree, just as with ordinary binary search trees. Recall that the node containing k is removed if it has zero or one children. If it has two children, the node with the next higher key is removed instead. In either case, let X be the node removed from the tree. After X is removed, splay X's parent to the root. Here's a sequence illustrating the operation remove(2).



In this example, the key 4 moves up to replace the key 2 at the root. After the node containing 4 is removed, its parent (containing 5) splays to the root.

If the key k is not in the tree, splay the node where the search ended to the root, just like in a find() operation.

Postscript: Babble about Splay Trees (not examinable, but good for you)

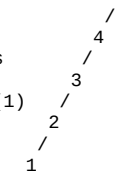
It may improve your understanding to watch the splay tree animation at http://www.ibr.cs.tu-bs.de/courses/ss98/audii/applets/BST/SplayTree-Example.html .

Splay trees can be rigorously shown to run in O(log n) average time per operation, over any sequence of operations (assuming we start from an empty tree), where n is the largest size the tree grows to. However, the proof is quite elaborate. It relies on an interesting algorithm analysis technique called _amortized_analysis_, which uses a _potential_function_ to account for the time saved by operations that execute more quickly than expected. This "saved-up time" can later be spent on the rare operations that take longer than O(log n) time to execute. By proving that the potential function is never negative (that is, our "bank account" full of saved-up time never goes into the red), we prove that the operations take O(log n) time on average.

The proof is given in Goodrich & Tamassia Section 10.3.3 and in the brilliant original paper in the Journal of the Association for Computing Machinery, volume 32, number 3, pages 652-686, July 1985. Unfortunately, there's not much intuition for why the proof works. You crunch the equations and the result comes out.

In 2000, Danny Sleator and Robert Tarjan won the ACM Kanellakis Theory and Practice Award for their papers on splay trees and amortized analysis. Splay trees are used in Windows NT (in the virtual memory, networking, and file system code), the gcc compiler and GNU C++ library, the sed string editor, Fore Systems network routers, the most popular implementation of Unix malloc, Linux loadable kernel modules, and in much other software.

When do operations occur that take more than O(log n) time? Consider inserting a long sequence of numbers in order: 1, 2, 3, etc. The splay tree will become a long chain of left children (as illustrated at right). Now, find(1) will take Theta(n) time. However, each of the n insert() operations before the find took O(1) time, so the average for this example is O(1) time per operation.



The fastest implementations of splay trees don't use the bottom-up splaying strategy discussed here. Splay trees, like 2-3-4 trees, come in bottom-up and top-down versions. Instead of doing one pass down the tree and another pass up, top-down splay trees do just one pass down. This saves a constant factor in the running time.

There is an interesting conjecture about splay trees called the _dynamic_ _optimality_conjecture_: that splay trees are as asymptotically fast on _any_ sequence of operations as _any_ other type of search tree with rotations. What does this mean? Any sequence of splay tree operations takes amortized O(log n) time per operation, but sometimes there are sequences of operations that can be processed faster by a sufficiently smart data structure. One example is accessing the same ten keys over and over again (which a splay tree can do in amortized O(1) time per access). The dynamic optimality conjecture guesses that if _any_ search tree can exploit the structure of a sequence of accesses to achieve asymptotically faster running time, so can splay trees.

The conjecture has never been proven, but it's not clear whether it's been disproven, either.

One special case that has been proven is that if you perform the find operation on each key in a splay tree in order from the smallest key to the largest key, the total time for all n operations is O(n), and not O(n log n) as you might expect.

CS61B: Lecture 37
Wednesday, April 23, 2014

AMORTIZED ANALYSIS

We've seen several data structures for which I claimed that the average time for certain operations is always better than the worst-case time: hash tables, tree-based disjoint sets, and splay trees.

The mathematics that proves these claims is called _amortized_analysis_. Amortized analysis is a way of proving that even if an operation is occasionally expensive, its cost is made up for by earlier, cheaper operations.

The Averaging Method

Most hash table operations take O(1) time, but sometimes an operation forces a hash table to resize itself, at great expense. What is the average time to insert an item into a hash table with resizing? Assume that the chains never grow longer than O(1), so any operation that doesn't resize the table takes O(1) time--more precisely, suppose it takes at most one second.

Let n be the number of items in the hash table, and N the number of buckets. Suppose it takes one second for the insert operation to insert the new item, increment n, and then check if n = N. If so, it doubles the size of the table from N to 2N, taking 2N additional seconds. This resizing scheme ensures that the load factor n/N is always less than one.

Suppose every newly constructed hash table is empty and has just one bucket--that is, initially n = 0 and N = 1. After i insert operations, n = i. The number of buckets N must be a power of two, and we never allow it to be less than or equal to n; so N is the smallest power of two > n, which is <= 2n.

The total time in seconds for _all_ the table resizing operations is

$$2 + 4 + 8 + \ldots + N/4 + N/2 + N = 2N - 2.$$

So the cost of i insert operations is at most i + 2N - 2 seconds. Because N <= 2n = 2i, the i insert operations take <= 5i - 2 seconds. Therefore, the _average_ running time of an insertion operation is (5i - 2)/i = 5 - 2/i seconds, which is in O(1) time.

We say that the _amortized_running_time_ of insertion is in O(1), even though the worst-case running time is in Theta(n).

For almost any application, the amortized running time is more important than the worst-case running time, because the amortized running time determines the total running time of the application. The main exceptions are some applications that require fast interaction (like video games), for which one really slow operation might cause a noticeable glitch in response time.

The Accounting Method

Consider hash tables that resize in both directions: not only do they expand as the number of items increases, but they also shrink as the number of items decreases. You can't analyze them with the averaging method, because you don't know what sequence of insert and remove operations an application might perform.

Let's try a more sophisticated method. In the _accounting_method_, we "charge" each operation a certain amount of time. Usually we overcharge. When we charge more time than the operation actually takes, we can save the excess time in a bank to spend on later operations.

Before we start, let's stop using seconds as our unit of running time. We don't actually know how many seconds any computation takes, because it varies from computer to computer. However, everything a computer does can be broken down into a sequence of constant-time computations. Let a _dollar_ be a unit of time that's long enough to execute the slowest constant-time computation that comes up in the algorithm we're analyzing. A dollar is a real unit of time, but it's different for different computers.

Each hash table operation has
- an _amortized_cost_, which is the number of dollars that we "charge" to do that operation, and
- an _actual_cost_, which is the actual number of constant-time computations the operation performs.

The amortized cost is usually a fixed function of n (e.g. $5 for insertion into a hash table, or $2 log n for insertion into a splay tree), but the actual cost may vary wildly from operation to operation. For example, insertion into a hash table takes a long, long time when the table is resized.

When an operation's amortized cost exceeds its actual cost, the extra dollars are saved in the bank to be spent on later operations. When an operation's actual cost exceeds its amortized cost, dollars are withdrawn from the bank to pay for an unusually expensive operation.

If the bank balance goes into surplus, it means that the actual total running time is even faster than the total amortized costs imply.

THE BANK BALANCE MUST NEVER FALL BELOW ZERO. If it does, you are spending more total dollars than your budget claims, and you have failed to prove anything about the amortized running time of the algorithm.

Think of amortized costs as an allowance. If your dad gives you $500 a month allowance, and you only spend $100 of it each month, you can save up the difference and eventually buy a car. The car may cost $30,000, but if you saved that money and don't go into debt, your _average_ spending obviously wasn't more than $500 a month.

Accounting of Hash Tables

Suppose every operation (insert, find, remove) takes one dollar of actual running time unless the hash table is resized. We resize the table in two circumstances.

- An insert operation doubles the table size if n = N AFTER the new item is inserted and n is incremented, taking 2N additional dollars of time for resizing to 2N buckets. Thus, the load factor is always less than one.
- The remove operation halves the table size if n = N/4 AFTER the item is deleted and n is decremented, taking N additional dollars of time for resizing to N/2 buckets. Thus, the load factor is always greater than 0.25 (except when n = 0, i.e. the table is empty).

Either way, a hash table that has _just_ been resized has n = N/2.
A newly constructed hash table has n = 0 items and N = 1 buckets.

By trial and error, I came up with the following amortized costs.

```
    insert:  5 dollars
    remove:  5 dollars
    find:    1 dollar
```

Is this accounting valid, or will we go broke?

The crucial insight is that at any time, we can look at a hash table and know a lower bound for how many dollars are in the bank from the values of n and N. We know that the last time the hash table was resized, the number of items n was exactly N/2. So if n != N/2, there have been subsequent insert/remove operations, and these have put money in the bank.

We charge an amortized $5 for an insert or remove operation. Every insert or remove operation costs one actual dollar (not counting resizing) and puts the remaining $4 in the bank to pay for resizing. For every step n takes away from N/2, we accumulate another $4. So there must be at least 4|n - N/2| dollars saved (or 4n dollars for a never-resized one-bucket hash table).

IMPORTANT: Note that 4|n - N/2| is a function of the data structure, and does NOT depend on the history of hash table operations performed. In general, the accounting method only works if you can tell how much money is in the bank (or, more commonly, a minimum bound on that bank balance) just by looking at the current state of the data structure--without knowing how the data structure reached that state.

An insert operation only resizes the table if the number of items n reaches N. According to the formula 4|n - N/2|, there are at least 2N dollars in the bank. Resizing the hash table from N to 2N buckets costs 2N dollars, so we can afford it. After we resize, the bank balance might be zero again, but it isn't negative.

A remove operation only resizes the table if the number of items n drops to N/4. According to the formula 4|n - N/2|, there are at least N dollars in the bank. Resizing the hash table from N to N/2 buckets costs N dollars, so we can afford it.

The bank balance never drops below zero, so my amortized costs above are valid. Therefore, the amortized cost of all three operations is in O(1).

Observe that if we alternate between inserting and deleting the same item over and over, the hash table is never resized, so we save up a lot of money in the bank. This isn't a problem; it just means the algorithm is faster (spends fewer dollars) than my amortized costs indicate.

Why Does Amortized Analysis Work?

Why does this metaphor about putting money in the bank tell us anything about the actual running time of an algorithm?

Suppose our accountant keeps a ledger with two columns: the total amortized cost of all operations so far, and the total actual cost of all operations so far. Our bank balance is the sum of all the amortized costs in the left column, minus the sum of all the actual costs in the right column. If the bank balance never drops below zero, the total actual cost is less than or equal to the total amortized cost.

Total amortized cost	Total actual cost
$5	$1
$1	$1
$5	$3
.	.
.	.
.	.
$5	$1
$5	$2,049
$1	$1
$12,327 >=	$10,333

Therefore, the total running time of all the actual operations is never longer than the total amortized cost of all the operations.

Amortized analysis (as presented here) only tells us an upper bound (big-Oh) on the actual running time, and not a lower bound (big-Omega). It might happen that we accumulate a big bank balance and never spend it, and the total actual running time might be much less than the amortized cost. For example, splay tree operations take amortized O(log n) time, where n is the number of items in the tree, but if your only operation is to find the same item n times in a row, the actual average running time is in O(1).

If you want to see the amortized analysis of splay trees, Goodrich and Tamassia have it. If you take CS 170, you'll see an amortized analysis of disjoint sets. I am saddened to report that both analyses are too complicated to provide much intuition about their running times. (Especially the inverse Ackermann function, which is ridiculously nonintuitive, though cool nonetheless.)

CS61B: Lecture 38
Monday, April 29, 2013

RANDOMIZED ANALYSIS
==========

_Randomized_algorithms_ are algorithms that make decisions based on rolls of the dice. The random numbers actually help to keep the running time low. Examples are quicksort, quickselect, and hash tables with random hash codes.

Randomized analysis, like amortized analysis, is a mathematically rigorous way of saying, "The average running time of this operation is fast, even though the worst-case running time is slow." Unlike amortized analysis, the "average" is taken over an infinite number of runs of the program. A randomized algorithm will sometimes run more slowly than the average, but the probability that it will run _asymptotically_ slower is extremely low.

Randomized analysis requires a little bit of probability theory.

Expectation

Suppose a method x() flips a coin. If the coin comes up heads, x() takes one second to execute. If it comes up tails, x() takes three seconds.

Let X be the exact running time of one call to x(). With probability 0.5, X is 1, and with probability 0.5, X is 3. For obvious reasons, X is called a _random_variable_.

The _expected_ value of X is the average value X assumes in an infinite sequence of coin flips,

```
E[X] = 0.5 * 1 + 0.5 * 3 = 2 seconds expected time.
```

Suppose we run the code sequence

```
x();     // takes time X
x();     // takes time Y
```

and let Y be the running time of the _second_ call. The total running time is T = X + Y. (Y and T are also random variables.) What is the expected total running time E[T]?

The main idea from probability we need is called _linearity_of_expectation_, which says that expected running times sum linearly.

```
E[X + Y] = E[X] + E[Y]
         = 2 + 2
         = 4 seconds expected time.
```

The interesting thing is that linearity of expectation holds true whether or not X and Y are _independent_. Independence means that the first coin flip has no effect on the outcome of the second. If X and Y are independent, the code will take four seconds on average. But what if they're not? Suppose the second coin flip always matches the first--we always get two heads, or two tails. Then the code still takes four seconds on average. If the second coin flip is always the opposite of the first--we always get one head and one tail--the code still takes four seconds on average.

So if we determine the expected running time of each individual operation, we can determine the expected running time of a whole program by adding up the expected costs of all the operations.

Hash Tables

The implementations of hash tables we have studied don't use random numbers, but we can model the effects of collisions on running time by pretending we have a random hash code.

A _random_hash_code_ maps each possible key to a number that's chosen randomly. This does _not_ mean we roll dice every time we hash a key. A hash table can only work if a key maps to the same bucket every time. Each key hashes to a randomly chosen bucket in the table, but a key's random hash code never changes.

Unfortunately, it's hard to choose a hash code randomly from all possible hash codes, because you need to remember a random number for each key, and that would seem to require another hash table. However, random hash codes are a good _model_ for how a good hash code will perform. The model isn't perfect, and it doesn't apply to bad hash codes, but for a hash code that proves effective in experiments, it's a good rough guess. Moreover, there is a sneaky number-theoretical trick called _universal_hashing_ that generates random hash codes. These random hash codes are chosen from a relatively small set of possibilities, yet they perform just as well as if they were chosen from the set of all possible hash codes. (If you're interested, you can read about it in the textbook "Algorithms" by Cormen, Leiserson, Rivest, and Stein.)

Assume our hash table uses chaining and does not allow duplicate keys. If an entry is inserted whose key matches an existing entry, the old entry is replaced.

Suppose we perform the operation find(k), and the key k hashes to a bucket b. Bucket b contains at most one entry with key k, so the cost of the search is one dollar, plus an additional dollar for every entry stored in bucket b whose key is not k. (Recall from last lecture that a _dollar_ is a unit of time chosen large enough to make this statement true.)

Suppose there are n keys in the table besides k. Let V1, V2, ..., Vn be random variables such that for each key ki, the variable Vi = 1 if key ki hashes to bucket b, and Vi is zero otherwise. Then the cost of find(k) is

```
T = 1 + V1 + V2 + ... + Vn.
```

The expected cost of find(k) is (by linearity of expectation)

```
E[T] = 1 + E[V1] + E[V2] + ... + E[Vn].
```

What is E[Vi]? Since there are N buckets, and the hash code is random, each key has a 1/N probability of hashing to bucket b. So E[Vi] = 1/N, and

```
E[T] = 1 + n/N,
```

which is one plus the load factor! If we keep the load factor n/N below some constant c as n grows, find operations cost expected O(1) time.

The same analysis applies to insert and remove operations. All three hash table operations take O(1) expected amortized time. (The word "amortized" accounts for table resizing, as discussed last lecture.)

Observe that the running times of hash table operations are _not_ independent. If key k1 and key k2 both hash to the same bucket, it increases the running time of both find(k1) and find(k2). Linearity of expectation is important because it implies that we can add the expected costs of individual operations, and obtain the expected total cost of all the operations an algorithm performs.

Quicksort

Recall that mergesort sorts n items in O(n log n) time because the recursion tree has 1 + ceiling(log_2 n) levels, and each level involves O(n) time spent merging lists. Quicksort also spends linear time at each level (partitioning the lists), but it is trickier to analyze because the recursion tree is not perfectly balanced, and some keys survive to deeper levels than others.

To analyze quicksort, let's analyze the expected depth one input key k will reach in the tree. (In effect, we're measuring a vertical slice of the recursion tree instead of a horizontal slice.) Assume no two keys are equal, since that is the slowest case.

Quicksort chooses a random pivot. The pivot is equally likely to be the smallest key, the second smallest, the third smallest, ..., or the largest. For each case, the probability is 1/n. Since we want a roughly balanced partition, let's say that the least floor(n/4) keys and the greatest floor(n/4) keys are "bad" pivots, and the other keys are "good" pivots. Since there are at most n/2 bad pivots, the probability of choosing a bad pivot is <= 0.5.

If we choose a good pivot, we'll have a 1/4-3/4 split or better, and our chosen key k will go into a subset containing at most three quarters of the keys, which is sorted recursively. If we choose a bad pivot, k might go into a subset with nearly all the other keys.

Let D(n) be a random variable equal to the deepest depth at which key k appears when we sort n keys. D(n) varies from run to run, but we can reason about its expected value. Since we choose a bad key no more than half the time,

  E[D(n)] <= 1 + 0.5 E[D(n)] + 0.5 E[D(3n / 4)].

Multiplying by two and subtracting E[D(n)] from both sides gives

  E[D(n)] <= 2 + E[D(3n / 4)].

This inequality is called a _recurrence_, and you'll learn how to solve them in CS 170. (No, recurrences won't be on the CS 61B final exam.) The base cases for this recurrence are D(0) = 0 and D(1) = 0. It's easy to check by substitution that a solution is

  E[D(n)] <= 2 log   n.
               4/3

So any arbitrary key k appears in expected O(log n) levels of the recursion tree, and causes O(log n) partitioning work. By linearity of expectation, we can sum the expected O(log n) work for each of the n keys, and we find that quicksort runs in expected O(n log n) time.

Quickselect

For quickselect, we can analyze the expected running time more directly. Suppose we run quickselect on n keys. Let P(n) be a random variable equal to the total number of keys partitioned, summed over all the partitioning steps. Then the running time is in Theta(P(n)).

Quickselect is like quicksort, but when we choose a good pivot, at least one quarter of the keys are discarded. We choose a good pivot at least half the time, so

  E[P(n)] <= n + 0.5 E[P(n)] + 0.5 E[P(3n / 4)],

which is solved by E[P(n)] <= 8n. Therefore, the expected running time of quickselect on n keys is in O(n).

Amortized Time vs. Expected Time

There's a subtle but important difference between amortized running time and expected running time.

Quicksort with random pivots takes O(n log n) expected running time, but its worst-case running time is in Theta(n^2). This means that there is a small possibility that quicksort will cost Omega(n^2) dollars, but the probability of that happening approaches zero as n approaches infinity.

A splay tree operation takes O(log n) amortized time, but the worst-case running time for a splay tree operation is in Theta(n). Splay trees are not randomized, and the "probability" of an Omega(n)-time splay tree operation is not a meaningful concept. If you take an empty splay tree, insert the items 1...n in order, then run find(1), the find operation _will_ cost n dollars. But a sequence of n splay tree operations, starting from an empty tree, _never_ costs more than O(n log n) actual running time. Ever.

Hash tables are an interesting case, because they use both amortization and randomization. Resizing takes Theta(n) time. With a random hash code, there is a tiny probability that every item will hash to the same bucket, so the worst-case running time of an operation is Theta(n)--even without resizing.

To account for resizing, we use amortized analysis. To account for collisions, we use randomized analysis. So when we say that hash table operations run in O(1) time, we mean they run in O(1) _expected_, _amortized_ time.

Splay trees	O(log n) amortized time / operation *
Disjoint sets (tree-based)	O(alpha(f + u, u)) amortized time / find op **
Quicksort	O(n log n) expected time ***
Quickselect	Theta(n) expected time ****
Hash tables	Theta(1) expected amortized time / op *****

If you take CS 170, you will learn an amortized analysis of disjoint sets there. Unfortunately, the analyses of both disjoint sets and splay trees are complicated. Goodrich & Tamassia give the amortized analysis of splay trees, but you're not required to read or understand it for this class.

* Worst-case time is in Theta(n), worst-case amortized time is in Theta(log n), best-case time is in Theta(1).

** For find operations, worst-case time is in Theta(log u), worst-case amortized time is in Theta(alpha(f + u, u)), best-case time is in Theta(1). All union operations take Theta(1) time.

*** Worst-case time is in Theta(n^2)--if we get worst-case input AND worst-case random numbers. "Worst-case expected" time is in Theta(n log n)--meaning when the _input_ is worst-case, but we take the average over all possible sequences of random numbers. Recall that quicksort can be implemented so that keys equal to the pivot go into a separate list, in which case the best-case time is in Theta(n), because the best-case input is one where all the keys are equal. If quicksort is implemented so that keys equal to the pivot are divided between lists I1 and I2, as is the norm for array-based quicksort, then the best-case time is in Theta(n log n).

**** Worst-case time is in Theta(n^2)--if we get worst-case input AND worst-case random numbers. Worst-case expected time, best-case time, and best-case expected time are in Theta(n).

***** Worst-case time is in Theta(n), expected worst-case time is in Theta(n) (worst case is when table is resized), amortized worst-case time is in Theta(n) (worst case is when every item is in one bucket), worst-case expected amortized time is in Theta(1), best-case time is in Theta(1). Confused yet?

CS61B: Lecture 39
Wednesday, April 30, 2014

Today's reading: Goodrich & Tamassia, Sections 14.1.2-14.1.3.

GARBAGE COLLECTION

Objects take up space in memory. If your program creates lots of objects, throws them away, and creates more, you might eventually run out of memory. To reduce the chance that this will happen, Java has garbage collection.

While the Java Virtual Machine (JVM) runs your program, it also spends little bits of time searching for objects that you're no longer using, so it can reclaim their memory for use by other objects.

You don't have to know anything about garbage collection to be an effective Java programmer. But garbage collection is interesting because the JVM uses a lot of hidden data structures to manage memory. These data structures are hidden from your Java program--after all, the JVM, just like any other encapsulated module, should hide the details of how it is implemented. Here's a peak at what's going on under the hood.

Roots and Reachability

Garbage collection's prime directive is to never sweep up an object your program might possibly use or inspect again. These objects are said to be _live_. The opposite of "live" is _garbage_--objects that your program cannot reference again. Java's design makes it possible for the JVM to determine whether an object can ever be used again by your program or not.

Garbage collection begins at the roots. A _root_ is any object reference your program can access directly, without going through another object. There are two kinds (that we know about). First, every local variable (including parameters) in every stack frame on the program stack is a root if it is a reference. (Primitive types like ints are not roots; only references are.) Second, every class variable (aka "static" field) that is a reference is a root.

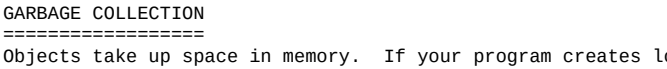
An object is live, and should not be garbage collected, if
- it is referenced by a root (obviously), or
- it is referenced by a field in another live object.

Note that this definition is recursive. Another way to say it is that an object is live if it is _reachable_ from the roots. If you run depth-first search starting at all the roots, you will visit all the live objects and none of the garbage. And that's exactly what garbage collectors do: run depth-first search from the roots.

Each object has a small tag that allows the garbage collector to mark whether the object has been visited or not. The tag is invisible to your Java program, but it takes a few bits of the object's memory. (This is not the only "hidden" memory Java associates with an object--for example, every object has a hidden label that tells Java what class it's in.)

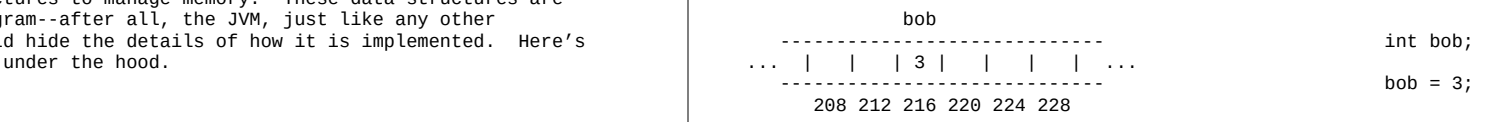
Memory Addresses

In any modern computer, memory is one huge array of bytes with addresses.

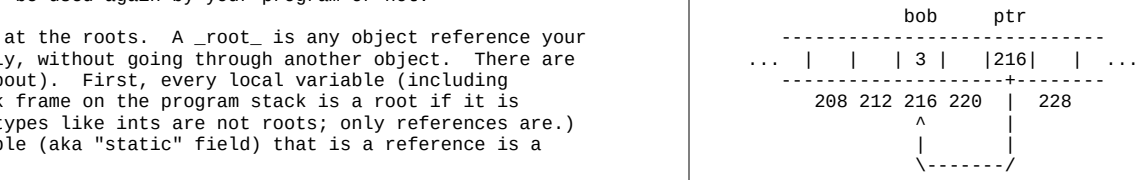


However, modern computers prefer to read or write four bytes at a time, and they do this much faster if the four bytes start at an address divisible by four, so that's how things like ints and floats are stored.

Every time you declare a local variable, you are naming a memory location. (You pick the name, Java picks the address.) An assignment statement writes something into a memory location.



Computers can store memory addresses in memory. To reduce the number of syllables, memory addresses are called _pointers_ for short. Some languages like C allow you to declare variables that are pointers. A pointer field is a memory location that points to another memory location.



Java uses pointers too, but it considers them confidential information, and won't let you print them or look at the numbers directly. Java references are a little bit like pointers, but as we'll learn soon, they're actually more complicated than pointers.

The important point is that your computer's memory is just one giant array that has no structure except the structure you impose on it. Java saves you a huge amount of time and effort by structuring memory for you. Java does this by using hidden pointer-based data structures that you can't access from a Java program.

Mark and Sweep Garbage Collection

A mark-and-sweep garbage collector runs in two separate phases. The _mark_ phase does a DFS from every root, and marks all the live objects. The _sweep_ phase does a pass over all the objects in memory. Each object that was not marked as being live is garbage, and its memory is reclaimed.

How does the sweep phase do a pass over all the objects in memory? The JVM has an elaborate internal data structure for managing the heap. This hidden data structure keeps track of free memory and allocated memory so that new objects can be allocated without overwriting live ones. Time prevents my describing Java's heap data structure here, but it allows the garbage collector to do a pass over every object, even the ones that are not live. It's roughly like an invisible linked list that links _everything_.

Similarly, the stack frames on the stack are data structures that make it possible for the garbage collector to determine which data on the stack are references, and which are not.

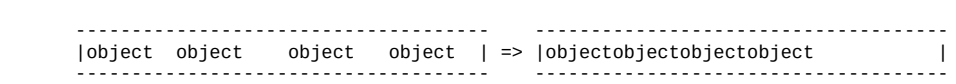
When a mark-and-sweep collector runs, your program stops running for an instant while the garbage collector does a mark pass and a sweep pass. The garbage collector is typically started when the JVM tries to create a new object but doesn't have enough memory for it.

Compaction

Typical programs allocate and forget a good many objects. One problem that arises is _fragmentation_, the tendency of the free memory to get broken up into lots of small pieces. Fragmentation can render Java unable to allocate a large object despite having lots of free memory available.

(Fragmentation also means that the memory caches and virtual memory don't perform as well. If you don't know why, wait until CS 61C or CS 152.)

To solve this problem, a compacting garbage collector actually picks up the objects and moves them to different locations in memory, thereby removing the space between the objects. This is easily done during the sweep phase.

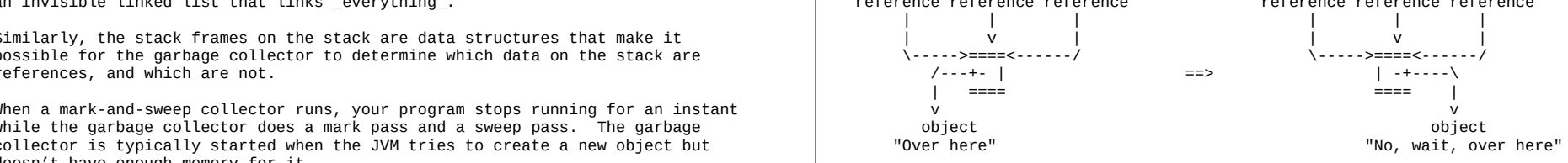


References

There's a problem here: if we pick up an object and move it, what about all the references to that object? Aren't those references wrong now?

Interestingly, in the Oracle JVM, a reference isn't a pointer. A reference is a handle. A _handle_ is a pointer to a pointer.

When an object moves, Java corrects the second pointer so it points to the object's new address. That way, even if there are a million references to the object, they're all corrected in one fell swoop. The "second pointers" are kept in a special table, since they don't take as much memory as objects.



The special table of "second pointers" does not suffer from fragmentation because all pointers have exactly the same size. Objects suffer from fragmentation because when a small object is garbage collected, the space it leaves behind might not be large enough to accommodate a larger object. But a garbage-collected object's "second pointer" can simply be reused by any newly constructed object that comes along, because all "second pointers" have the same size.

Copying Garbage Collection

Copying garbage collection is an alternative to mark and sweep. It does compaction, but it is faster than mark and sweep with compaction because there is only one phase, rather than a mark phase and a sweep phase.

Memory is divided into two distinct spaces, called the old space and the new space. A copying garbage collector finds the live objects by DFS as usual, but when it encounters an object in the old space, it _immediately_ moves it to the new space. The object is moved to the first available memory location in the new space, so compaction is part of the deal. After all the objects are moved to the new space, the garbage objects that remain in the old space are simply forgotten. There is no need for a sweep phase.

Next time the garbage collector runs, the new space is relabeled the "old space" and the old space is relabeled the "new space". Long-lived objects may be copied back and forth between the two spaces many times.

While your program is running (between garbage collections), all your objects are in one space, while the other space sits empty.

The advantage of copying garbage collection is that it's fast. The disadvantage is that you effectively cut in half the amount of heap memory available to your program.

CS61B: Lecture 40
Wednesday, April 30, 2014

Generational Garbage Collection

Studies of memory allocation have shown that most objects allocated by most programs have short lifetimes, while a few go on to survive through many garbage collections. This observation has inspired generational garbage collectors, which separate old from new objects.

A generational collector has two or more generations, which are like the separate spaces used by copying collectors, except that the generations can be of different sizes, and can change size during a program's lifetime.

Sun's 1.3 JVM divides objects into an old generation and a young generation. Because old objects tend to last longer, the old generation doesn't need to be garbage collected nearly as often. Hence, the old generation uses a compacting mark-and-sweep collector, because speed is not critical, but memory efficiency might be. Because old objects are long-lived, and because mark and sweep only uses one memory space, the old generation tends to remain compact.

The young generation is itself divided into three areas. The largest area is called "Eden", and it is the space where all objects are born, and most die. Eden is large enough that most objects in it will become garbage long before it gets full. When Eden fills up, it is garbage collected and the surviving objects are copied into one of two _survivor_spaces_. The survivor spaces are just the two spaces of a copying garbage collector.

If an unexpectedly large number of objects survive Eden, the survivor spaces can expand if necessary to make room for additional objects.

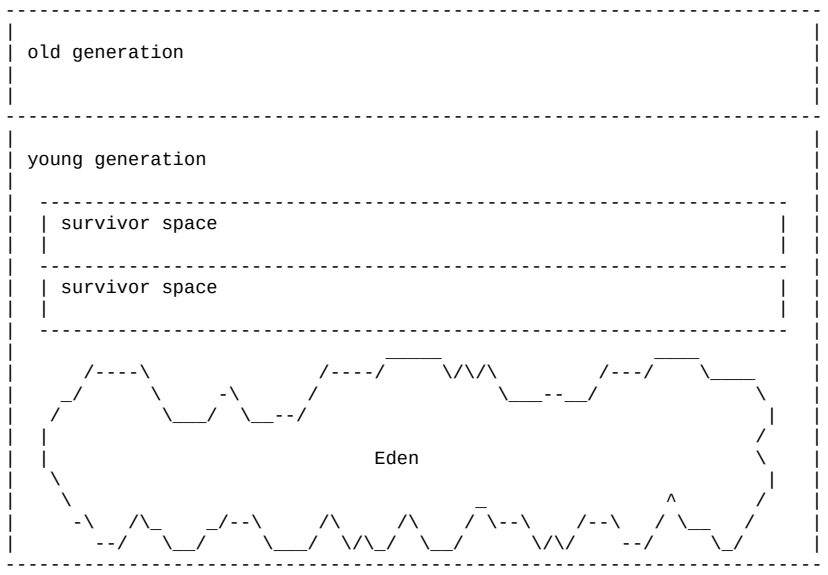
Objects move back and forth between the two survivor spaces until they age enough to be _tenured_ - moved to the old generation. Young objects benefit from the speed of the copying collector while they're still wild and prone to die young.

Thus, the Sun JVM takes advantage of the best features of both the mark-and-sweep and copying garbage collection methods.

There are two types of garbage collection: minor collections, which happen frequently but only affect the young generation - thereby saving lots of time - and major collections, which happen much less often but cover all the objects in memory.

This introduces a problem. Suppose a young object is live only because an old object references it. How does the minor collection find this out, if it doesn't search the old generation?

References from old objects to young objects tend to be rare, because old objects are set in their ways and don't change much. Since references from old objects to young are so rare, the JVM keeps a special table of them, which it updates whenever such a reference is created. The table of references is added to the roots of the young generation's copying collector.



AUGMENTING DATA STRUCTURES

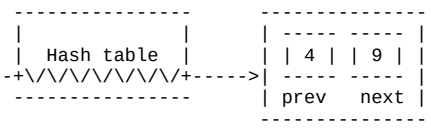
Once you know how to design one of the data structures taught in this class, it's sometimes easy to augment it to have "extra" abilities.

You've already augmented data structures in Project 3. For example, the set E of edges is stored as both a hash table and an adjacency list. The hash table allows you to test set membership in O(1) time, unlike the adjacency list. The adjacency list tells you the edges adjoining a vertex in O(degree) time, unlike the hash table.

2-3-4 Trees with Fast Neighbors

Suppose you have a 2-3-4 tree with no duplicate keys. Given a key k, you want to be able to determine whether k is in the tree, and what the next smaller and larger keys are, in O(1) time. You are allowed to change the insert() and remove() operations, but they still must take O(log n) time. Can you do it?

It's easy if you combine the 2-3-4 tree with a hash table. The hash table maps each key to a record that stores the next smaller and next larger keys in the tree.



The trick is that when you insert a key into the tree, you can determine by tree search in O(log n) time what the next smaller and larger keys are. Then, you update all three keys' records in the hash table in O(1) time.

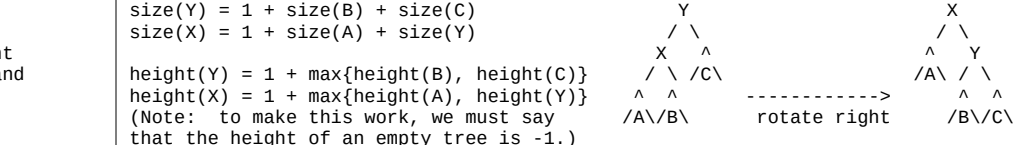
When you remove a key from the tree, you can learn its two neighboring keys from the hash table, then update the neighbor records for those two keys so they list each other instead of the removed key. You also remove the key's record from the hash table. The hash table updates take O(1) time, and it takes O(log n) time to remove the key from the 2-3-4 tree itself.

Splay Trees with Node Information

Sometimes it's useful for a binary search tree to record extra information in each node, like the size and height of each subtree at each node.

In splay trees, this is easy to maintain. Splaying is just a sequence of tree rotations. Each rotation changes the sizes of only two subtrees, and we can easily compute their new sizes after the rotation. Let size(Y) be the number of nodes in the subtree rooted at node Y. After a right rotation (for instance) you can recompute the information as follows:

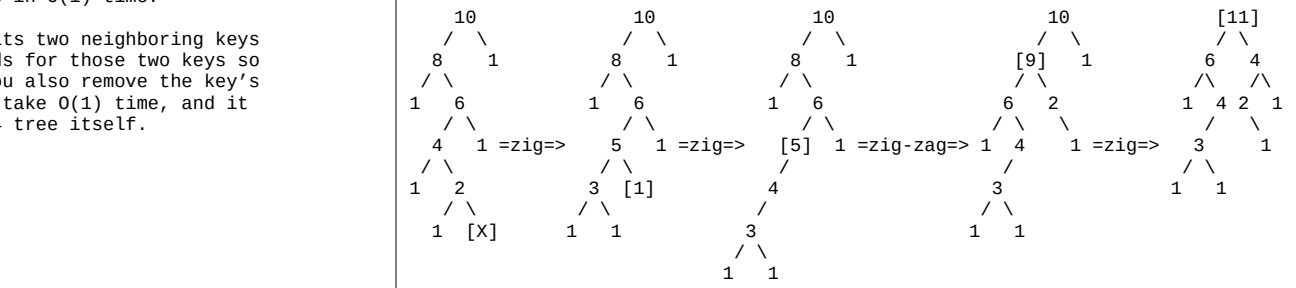
```
size(Y) = 1 + size(B) + size(C)
size(X) = 1 + size(A) + size(Y)

height(Y) = 1 + max{height(B), height(C)}
height(X) = 1 + max{height(A), height(Y)}
(Note:  to make this work, we must say
that the height of an empty tree is -1.)
```



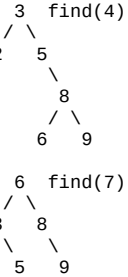
Be forwarned that a rotation does not just change the heights of X and Y--it also can change the heights of all their ancestors. But X gets splayed all the way to the root, so all the ancestors' heights get fixed on the way up.

Likewise, inserting or removing an item changes the subtree sizes of all the ancestors of the affected item, and possibly their heights as well. But a newly inserted item gets splayed to the top; and a removed node's parent is splayed to the top. So again, all the sizes and heights will get fixed during the rotations. Let's watch the size fields as we insert a new node X into a splay tree. (The following numbers are sizes, _not_ keys.)

Note that the very first rotation is at the grandparent of node X (zig-zig).



How can we use this information? We can answer the query "How many keys are there between x and y?" in O(log n) amortized time if the splay tree has no duplicate keys and we label every subtree with its size. Our strategy is to set c = n, then deduct from c the number of keys outside the range [x, y].




```
find(x);  // After the splaying, the keys in the root's left
// subtree are all less than x, so subtract their number from c.
c = c - size(root's left subtree);
if (root key < x)  // Only possible if x is not in the tree--
  c--;             // otherwise x was splayed to the root.

find(y);  // After the splaying, the keys in the root's
          // right subtree all exceed y.
c = c - size(root's right subtree);
if (root key > y) c--;
```

Now, c is the number of keys in [x, y].

Keys in [4, 7] =
7 - 2 - 1 - 2 = 2.

CS 61B: Practice for the Final Exam

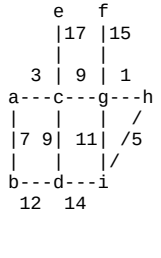
Please try to answer these questions. We'll release solutions two or three days before the exam. Starred problems are particularly difficult--much more difficult than any exam question would be.

Warning: Midterm 1 topics are absent here, but they can reappear on the Final.

[1] Given an array containing the digits 71808294, show how the order of the digits changes during each step of [a] insertion sort, [b] selection sort, [c] mergesort, [d] quicksort (using the array-based quicksort of Lecture 31, and always choosing the last element of any subarray to be the pivot), and [e] heapsort (using the backward min-heap version discussed in Lecture 30). Show the array after each swap, except in insertion sort. For insertion sort, show the array after each insertion.

[2] Some sorting methods, like heapsort and array-based quicksort, are not naturally stable. Suggest a way to make _any_ sorting algorithm stable by extending the keys (making them longer and adding extra information).

[3] Consider the graph at right.



[a] In what order are the vertices visited using DFS starting from vertex a? Where a choice exists, use alphabetical order. What if you use BFS?

[b] A vertex x is "finished" when the recursive call DFS(x) terminates. In what order are the vertices finished? (This is different from the order in which they are visited, when DFS(x) is called.)

[c] In what order are edges added to the minimum spanning tree by Kruskal's algorithm? List the edges by giving their endpoints.

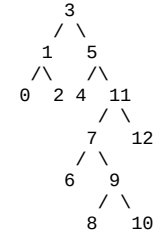
[4] [a] How long does it take to determine if an undirected graph contains a vertex that is connected to no other vertex [i] if you use an adjacency matrix; [ii] if you use an adjacency list.

[b] Suppose we use DFS on a binary search tree, starting from the root. The edge to a left child is always traversed before an edge to the right child. In what order are the nodes visited? Finished?

[c] An undirected graph contains a "cycle" (i.e., loop) if there are two different simple paths by which we can get from one vertex to another. Using breadth-first search (not DFS), how can we tell if an undirected graph contains a cycle?

[d] Recall that an undirected graph is "connected" if there is a path from any vertex to any other vertex. If an undirected graph is not connected, it has multiple connected components. A "connected component" consists of all the vertices reachable from a given vertex, and the edges incident on those vertices. Suggest an algorithm based on DFS (possibly multiple invocations of DFS) that counts the number of connected components in a graph.

[5] What does the splay tree at right look like after:



[a] max() [the operation that finds the maximum item]

[b] insert(4.5)

[c] find(10)

[d] remove(9)

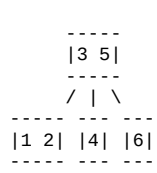
Start from the _original_ tree, not the tree resulting from the previous operation.

[6] Consider the quick-union algorithm for disjoint sets. We know that a sequence of n operations (unions and finds) can take asymptotically slightly more than linear time in the worst case.

[a] Explain why if all the finds are done before all the unions, a sequence of n operations is guaranteed to take O(n) time.

[b] * Explain why if all the unions are done before all the finds, a sequence of n operations is guaranteed to take O(n) time. Hint: you can tell the number of dollars in the bank just by looking at the forest.

[7] [a] Suggest a sequence of insertion operations that would create the binary tree at right.



[b] Suggest a sequence of operations that would create the 2-3-4 tree at right. You are allowed to use removal as well as insertion.

[8] Suppose an application uses only three operations: insert(), find(), and remove().

[a] Under what circumstances would you use a splay tree instead of a hash table?

[b] Under what circumstances would you use a 2-3-4 tree instead of a splay tree?

[c] Under what circumstances would you use an unordered array instead of a 2-3-4 tree?

[d] Under what circumstances would you use a binary heap instead of an unordered array?

[e] Under what circumstances would you use a hash tabel instead of a binary heap?

[9] [a] * Suppose we are implementing a binary heap, based on reference-based binary trees (_not_ arrays). We want to implement a deleteRef() operation which, given a _reference_ to a node in the tree, can delete that node (and the item it contains) from the heap while maintaining the heap-order property--even if the node isn't the root and its item isn't the minimum. deleteRef() should run in O(log n) time. How do we do it?

[b] Building on your answer to the previous question, explain how to combine a min-heap and max-heap (both using reference-based binary trees) to yield a data structure that implements insert(), deleteMin(), and deleteMax() in O(log n) time. Hint: You will need inter-heap pointers. Think of how you deleted edges in Project 3, for example.

[c] How can we accomplish the same thing if we use array-based heaps? Hint: Add an extra field to the items stored in each array.

[10] Suppose we wish to create a binary heap containing the keys D A T A S T R U C T U R E. (All comparisons use alphabetical order.)

[a] Show the resulting min-heap if we build it using successive insert() operations (starting from D).

[b] Show the resulting min-heap if we build it using bottomUpHeap().

[11] [a] In Lecture 26, we told you how to implement a method smallestKeyNotSmaller(k) that returns the smallest key not less than k in a binary search tree. If the search tree contains an entry with key k, then an entry with key k is returned.

Describe how to implement a method smallestKeyGreater(k) that returns the smallest key strictly greater than k in a binary search tree. Hint: write a slightly modified version of find() that acts as if it were searching for k + epsilon, where epsilon > 0 is an infinitesimally small number. Therefore, it is never an exact match with any key in the tree. (This "hint" is actually a very useful general-purpose technique worth remembering.)

For extra practice, code it in Java. Use the BinaryTree data structure from Lecture 26.

[b] * You are given a binary search tree that is NOT a splay tree and does not rebalance itself. However, every node of the tree stores a field that specifies the number of items/nodes in the subtree rooted at that node (as described at the end of Lecture 40).

Given two search keys x and y, describe an algorithm that computes the number of keys in the range [x, y] (inclusive) in O(h) time, where h is the height of the binary search tree.

For extra practice, code it in Java. Assume that every BinaryTreeNode has an extra int field named "size" that stores the size of the subtree rooted at that node.

[12] Suppose we modify the array-based quicksort() implementation in the Lecture 31 notes to yield an array-based quickselect() algorithm, as described in Lecture 34. Show the steps it would use to find the median letter in D A T A S T R U C T U R E. (The median in this case is the 7th letter, which would appear at array index 6 if we sorted the letters.) As in Question [1], choose the last element of any subarray to be the pivot, and show the array after each swap.

[13] Suppose our radix-sort algorithm takes exactly n+r microseconds per pass, where n is the number of keys to sort, and r is the radix (number of queues). To sort 493 keys, what radix r will give us the best running time? With this radix, how many passes will it take to sort 420-bit keys? To answer this question, you'll need to use calculus (and a calculator), and you'll need to remember that log2 r = (ln r) / (ln 2).

[14] Suppose that while your computer is sorting an array of objects, its memory is struck by a cosmic ray that changes exactly one of the keys to something completely different. For each of the following sorting algorithms, what is the _worst-case_ possibility? For each, answer [x] the final array won't even be close to sorted, [y] the final array will have just one or two keys out of place, or [z] the final array will consist of two separate sorted subsets, one following the other, plus perhaps one or two additional keys out of place.

[a] Insertion sort
[b] Selection sort
[c] Mergesort
[d] Radix sort

[15] [Note: this is included for those who want some programming practice. You are not responsible on the Final Exam for knowing anything about the video Sorting Out Sorting.]

Implement tree sort (as described in the sorting video) in Java. Assume your treeSort() method's only input parameters are the number of items and a complete (perfectly balanced) BinaryTree of depth d in which each leaf has an item; hence, there are 2^d items to sort. All internal nodes begin with their item field set to null. Use the data structures below (in which each node knows its left and right child), not a general tree.

Your algorithm should never change a node reference; only the items move. The centerpiece of your algorithm will be a method that fills an empty node by (i) recursively filling its left and right children if they're empty, and (ii) choosing the smaller of its children's items, which is moved up into the empty node. treeSort() will repeatedly: (i) apply this method to the root node to find the smallest item remaining in the tree, (ii) pluck that item out of the root node, leaving the root empty again, and (iii) put the item into an array of type Comparable[]. Your treeSort() should allocate and return that array.

```
public class BinaryTreeNode {
  Comparable item;
  BinaryNode leftChild;
  BinaryNode rightChild;
}
```

```
public class BinaryTree {
  BinaryTreeNode root;
  int size;
}
```

CS 61B: Selected Solutions (Practice for the Final Exam)

[1] [a] insertion sort

```
71808294
17808294
01788294
01278894
01247889
```

[b] selection sort

```
71808294
01878294
01278894
01248897
01247898
01247889
```

[c] mergesort

```
71808294
17,80,8294
17,08,8294
17,08,28,94
17,08,28,49
0178,28,49
0178,2489
01247889
```

[d] quicksort

```
71808294
21808794
21088794
210|4|8798
0|12|4|8798
012|4|7898
012|4|7|8|98
01247889
```

[e] heapsort

```
71808294 \
81807294  |
82807194  | bottomUpHeap()
82897104  |
82897140 /
08897241
01897842
01298874
01249887
01247898
01247898
01247889
```

[2] Extend each item so that it has a "secondary key," which is the index of the item in the initial array/list. If two items have the same primary key, the tie is broken using the secondary key, so no two items are ever considered equal.

```
5 8 7 5 8 8 3 7     =>     5/0 8/1 7/2 5/3 8/4 8/5 3/6 7/7
```

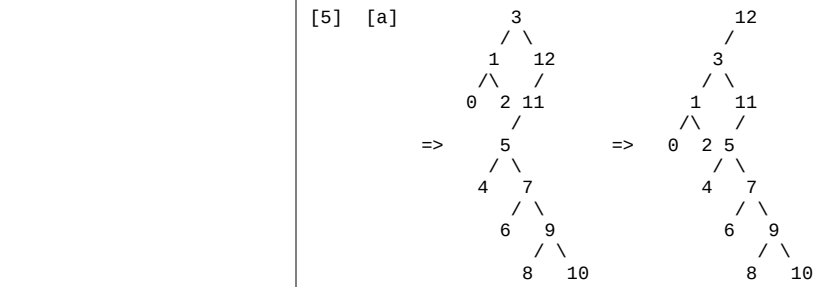
[3] [a] DFS: abdcegfhi
BFS: abcdegifh
[b] DFS: efihgcdba
[c] gh, ac, hi, ab, cd, cg, fg, ce (cg may come before cd instead)

[4] [a] [i] O(|V|^2) [ii] O(|V|)
[b] Visited in preorder. Finished in postorder.
[c] With BFS, it's done exactly the same as with DFS.
(See Homework 9 for a description of how it's done with the latter.)
[d]

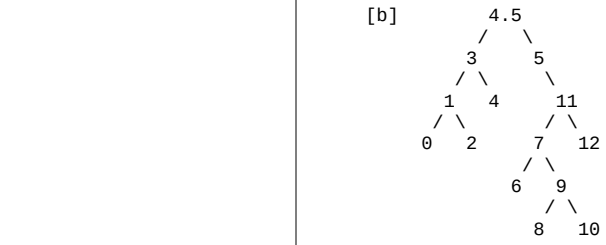
```
for (each vertex v in the graph) {
  if (v has not been visited) {
    increment the count of connected components
    perform DFS on v, thereby marking all vertices in its
                              connected component as having been visited
  }
}
```

This algorithm requires that you don't "unmark" the marked vertices between calls to DFS.

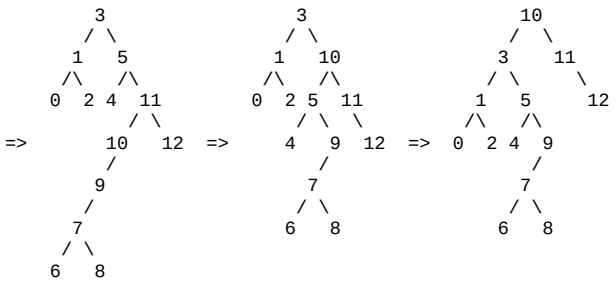
[5] [a]



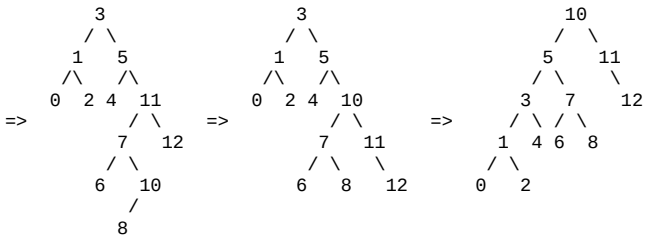
[b]



[c]



[d]



[6] [a] If the find operations are done first, then the find operations take O(1) time each because every item is the root of its own tree. No item has a parent, so finding the set an item is in takes a fixed number of operations.

Union operations always take O(1) time. Hence, a sequence of n operations with all the finds before the unions takes O(n) time.

[b] This question requires amortized analysis. Find operations can be expensive, but an expensive find operation is balanced out by lots of cheap union operations. The accounting is as follows.

Union operations always take O(1) time, so let's say they have an actual cost of $1. Assign each union operation an amortized cost of $2, so every union operation puts $1 in the bank.

Each union operation creates a new child. (Some node that was not a child of any other node before is a child now.) When all the union operations are done, there is $1 in the bank for every child, or in other words, for every node with a depth of one or greater.

Let's say that a find(u) operation costs $1 if u is a root. For any other node, the find operation costs an additional $1 for each parent pointer the find operation traverses. So the actual cost is $(1 + d), where d is the depth of u.

Assign each find operation an amortized cost of $2. This covers the case where u is a root or a child of a root. For each additional parent pointer traversed, $1 is withdrawn from the bank to pay for it.

Fortunately, path compression changes the parent pointers of all the nodes we pay $1 to traverse, so these nodes become children of the root. All of the traversed nodes whose depths are 2 or greater move up, so their depths are now 1. We will never have to pay to traverse these nodes again.

Say that a node is a grandchild if its depth is 2 or greater. Every time find(u) visits a grandchild, $1 is withdrawn from the bank, but the grandchild is no longer a grandchild. So the maximum number of dollars that can ever be withdrawn from the bank is the number of grandchildren. But we initially put $1 in the bank for every child, and every grandchild is a child, so the bank balance will never drop below zero. Therefore, the amortization works out.

Union and find operations both have amortized costs of $2, so any sequence of n operations where all the unions are done first takes O(n) time.

[7] [a] insert 4, 2, 1, 3, and 6 in that order.
[b] insert 4, 5, 6, 1, 3, and 2 in that order.

[8] [a] If you need inexact matches. For example, if you want to find the item less than or equal to 5 that's closest to 5. Hash tables can only do exact matches. (If exact matches are all you need, however, hash tables are faster.)
[b] If each single operation absolutely must run in O(log n) time. OR If most operations are find()s, and the data access patterns are uniformly random. (2-3-4 trees are faster for these operations because they don't restructure the tree. But splay trees do better if a small proportion of the items are targets of most of the finds.)
[c] If memory use is the primary consideration (especially if a 2-3-4 tree holding all the items won't fit in memory).
[d] None. find() and remove() on a heap take worst-case Theta(n) time, and they're more complicated than in an unordered array. insert() on a heap takes worst-case Theta(log n) time, versus Theta(1) for an unordered array.
[e] When you don't need to find the minimum key.

[10] [a]
```
          A
        A       E
      C   S   R   T
     U D T U T R
```

[b]
```
          A
        A       E
      C   S   R   R
     U D T U T T
```
(Note that two nodes are different than in [a].)

[12]
```
DATASTRUCTURE
DACASTRUTTURE
DACA|E|TRUTTURS
DACA|E|RRUTTUTS
DACA|E|RR|S|TTUTU
DACA|E|R|R|S|TTUTU
        ^
```

[13] Radix sort takes b/log2 r passes, so the overall running time of radix sort is

$$t = b (\ln 2) \frac{n + r}{\ln r}$$

To find the value of r that minimizes t, set dt/dr to zero.

$$\frac{dt}{dr} = b (\ln 2) \frac{\ln r - (n + r)/r}{(\ln r)^2} = 0$$

Therefore, ln r = (n + r)/r. Given that n = 493, with a calculator and some trial-and-error you can determine that r = 128 is the optimal radix.

[14] [a] z: consider the case where, half-way through the sort, the last key in the "sorted" list is changed to a very low number.

```
      1 3 5 7 9|8 4 2 6 10
 zap! 1 3 5 7 0|8 4 2 6 10
```

Using an in-place insertion sort implementation that searches from the end of the sorted array, the remaining keys will never get past the zero.

```
      1 3 5 7 0 2 4 6 8 10
```

Note that if a key in the "unsorted list" is zapped, no harm is done at all.

[b] y: If an item in the "sorted list" is zapped, only that one item is affected. If an item in the "unsorted list" is zapped to a value lower than the last item in the sorted list, that item will be out-of-place, but other items are still sorted.

[c] z: Consider merging two lists, where the first item in one of the lists gets zapped to a very high value. You'll wind up with two consecutive sorted portions. (After further merge operations, there will still be two consecutive sorted portions.)

```
                      /== 100 3 5 7 9 11
                merge  \== 2 4 6 8 10 12
```

[d] y: Radix sort uses no comparisons at all, so the zapped item doesn't affect how the others are ordered.