Introduction

- Name, major & year
- Favorite color
- If you could model something...

Arc of the course

We will be going over the toolbox of computer science. One part of that toolbox is a set of standard data structure concepts. Every program you write involves data structures. Data structures have different properties in terms of complexity, access time, ease of modification, and memory usage. No data structure is perfect for every situation, so proper selection of data structures can make a significant difference in system performance.

The second part of the toolbox is a set of algorithms for manipulating and organizing data. Most of these algorithms concern searching or sorting particular data structures. Given its importance in modeling and simulation, we will also be looking at the issue of nearest neighbor computations and how to accomplish them quickly.

To motivate learning the techniques, and to give you some practice in selecting appropriate data structures and evaluating their suitability, we will also be looking at some simple modeling and simulation problems. We’ll start with cellular automata on a grid, look at supermarket queues, and then do some simple ray-tracing in 2D to generate caustics. Beyond those applications, your interests should provide some guidance.

The overall goal of this course is to enable you to break down a real problem into manageable units, design algorithms and data structures appropriate to solve the problem, and implement a solution.

Administrivia

Language: Java or C++ (I don’t care, and I can provide assistance in either)

Web page: http://fangorn.colby.edu/cs/maxwell/courses/cs231

- Syllabus (subject to change)
- Assignments
- Readings
- Homeworks

Grading: Assignments and homeworks will account for half of your grade, with exams and class participation accounting for the remainder. My grading policy on assignments is that the tasks I specify explicitly will constitute about 85% of the assignment. If you complete the specified parts of the assignment properly, and produce a high-quality writeup, it’s worth up to a B+ grade. In addition, I will specify a variety of extensions to the assignment, or you can come up with your own. Completing one or more extensions, in addition to the specified parts of the assignment, will earn you some flavor of A.

Deadlines: Weekly homeworks will mostly be small programming assignments, which I expect to take less than an hour to complete. Homeworks must be emailed to me as a single zip file by class time on the given
due date. The homework deadline is a hard deadline, and you should complete them by yourself. You may discuss the homeworks among yourselves, but your code should be your own.

Assignments will be longer term projects, and I suggest you work in pairs. Collaboration on assignments is acceptable, but you need to understand everything you do, and your code should be your own group’s code. If you use ideas from another group, be sure to indicate that within your code.

**Handin:** For the assignments, you will need to put together a short web page writeup. The writeup should describe the assignment, explain any required theory, and present the results. To hand in your code, create a single zip file and email it to me along with the URL for the assignment.

**Survey**

What can you do?

What data structures have you seen?

For what are they useful?

Review of creating a class
/**
 * Creates a simple simulation of a 6-sided die
 * @Bruce Maxwell
 * @1.0
 */
public class Die
{
    // instance variables - replace the example below with your own
    private int x;

    /**
     * Constructor for objects of class Die
     */
    public Die()
    {
        // initialise instance variables
        x = 1;
    }

    /**
     * Gives the die a new value
     * @return new value of the die
     */
    public int roll()
    {
        x = ((int)(Math.random() * 6)) + 1;
        return x;
    }

    public int get()
    {
        return x;
    }

    public void set(int v) {
        x = v;
    }

    public static void main(String[] args) {
        Die d = new Die();
        System.out.println("Before rolling: " + d.get());
        d.roll();
        System.out.println("After rolling: " + d.get());
    }
}

Modified it to make an N-sided die.
Concept Review

Java, as an object-oriented language, provides structures for programming that let us encapsulate both code and data/information to make it easier to design programs and re-use code for multiple purposes.

Classes

Classes provide the structure on which we build software systems. A class can encapsulate a complete program or provide a small part of a larger program’s functionality. Classes enable us to divide a program into logical parts and write the parts separately. Division of a problem into manageable parts and hierarchical design are two major principles of software engineering and make large programs possible to design, debug, and modify.

A class has several major parts.

- Fields: hold information required by the class
- Methods: code that provides required functionality
- Constructors: methods that are called to initialize newly created instances of a class

Static v. Non-static methods

Classes can have static and non-static methods.

Non-static methods are called using an instance of a particular class. Within a non-static method, the object from which the method was called, its fields and its methods are available via the variable this.

Static methods can be called without an actual instance of the object. Any main method must be declared static so that the program can be executed (no instance of any object exists until the program begins).

Static methods are useful for classes that are intended to encapsulate functionality but not necessarily any data (the Math class, for example). I/O classes are often designed to have static methods, and their purpose is to connect an object to a stream. Thus, System.println() is a static method of the System class.

Primitives v. References

The eight primitive types defined in Java are labels for actual memory locations that hold data: boolean, char, byte, short, int, float, long, double. While Drake claims that byte, float, long, and short are not commonly used, it depends upon the application, precision requirements, and memory availability.

All other types in Java are labels for memory locations that hold pointers to where the data may actually be stored. These are also called references.

Why does this matter?

1. Function arguments are call by value
2. Comparing primitives compares the values of the primitives
3. Comparing references does not compare the value of the references, just the pointers
4. Copying a primitive (assignment statement) copies the value of the primitive
5. Copying a reference copies the pointer to the reference

It also matters because any time you declare a variable to be a non-primitive type you have to remember that the memory for the object itself does not yet exist, only a pointer to NULL. You must use the new statement to actually allocate the space for the object. If you try to access the object prior to allocating memory for it, bad things happen.

**Equality**

Whenever you see a statement like:

```c
if( A == B )
```

It is essential to know what exactly is being compared. If A and B are primitive types, the actual values are being compared, and the result of the statement is likely to be exactly what you want. However, if either A or B are references, then undesirable things can happen because the statement is comparing pointers to objects, not the objects themselves. See figure 1 for an example.

Because of the potential problems, it is wise to create an `equals()` method for each class you create that may be used in an equivalency comparison. For the reference classes belonging to the eight primitive types, the `equals()` is already defined, as shown in figure 1.

**Copying**

The same issues arise when copying from one object to another. A simple assignment statement only sets the value of the memory reference, not the contents of the object. In order to set the value of one object to the value of another object, the typical practice is to make both a constructor and a `set` function that take as an argument the source object and copy the data in the source object to the target object. The code in figure 2 demonstrates the problem.
public class strangeThings {

    // example of how equals does odd things with Objects
    public static void main(String[] args) {
        int a, b;
        Integer A, B;

        a = 6;
        b = 5;

        // with primitives, == works fine
        System.out.println("a: "+a);
        System.out.println("b: "+b);
        System.out.println("a == b: "+(a == b)); // prints false

        b = 6;
        System.out.println("a: "+a);
        System.out.println("b: "+b);
        System.out.println("a == b: "+(a == b)); // prints true

        // with references, things don’t work so well
        A = new Integer(6);
        B = A;
        System.out.println("A: "+A);
        System.out.println("B: "+B);
        // ok here, because A and B point to the same place
        System.out.println("A == B: "+(A == B)); // prints true

        B = new Integer(6);
        System.out.println("A: "+A);
        System.out.println("B: "+B);
        // not here, because A and B point to different places
        System.out.println("A == B: "+(A == B)); // prints false
        System.out.println("A.equals(B): "+A.equals(B)); // prints true
    }

    } // end class

    Figure 1: strangeThings class showing problems with ==.
public class strangeCopy {
    private int q;

    public strangeCopy(int x) {
        q = x;
    }

    public void set(int x) {
        q = x;
    }

    public int get() {
        return q;
    }

    public static void main(String[] args) {
        int a, b;
        strangeCopy A, B;

        a = 6;
        b = 5;

        A = new strangeCopy(6);
        B = new strangeCopy(5);

        System.out.println("a: "+a+" b: "+b); // prints a: 6 b: 6
        System.out.println("a = b");
        a = b;
        System.out.println("a: "+a+" b: "+b); // prints a: 5 b: 5
        System.out.println("a = 4");
        a = 4;
        System.out.println("a: "+a+" b: "+b); // prints a: 4 b: 5

        System.out.println();
        System.out.println("A = B");
        A = B;
        System.out.println("A: "+A.get()+" B: "+B.get()); // prints A: 5 B: 5
        System.out.println("A.set(4)");
        A.set(4);
        System.out.println("A: "+A.get()+" B: "+B.get()); // prints A: 4 B: 4
        return;
    }
}

Figure 2: strangeCopy class showing problems with assignment.
Arrays

Arrays are simply a contiguous chunk of memory that holds one or more pieces of data. The array elements can be primitive types or object references.

Example 1: The following creates an array of ints on the heap. The memory for the data is allocated as a single contiguous block.

```java
int[] myarray;
myarray = new int[100];
```

Example 2: The following creates an array of references to the class myObject on the heap. The memory for the references is allocated as a single contiguous block. Note that the memory space for the individual myObject values is not allocated. To actually store data in the array elements, we must first allocate all of the individual objects for the array.

```java
myObject[] myarray;
int i;
myarray = new myObject[100];
for(i = 0; i < 100; i++) {
    myarray[i] = new myObject();
}
```

Arrays can also be thought of as vectors. A vector is simply a set of numbers that collectively specify a location in an N-dimensional space. Any time a vector is required, one option is to store it in an array.

Arrays as data structures

Arrays, or vectors, are useful data structures for storing data. Consider the strengths and weaknesses of vectors as a data structure.

Strengths:
- Fast access to any element: constant time
- Minimal memory usage: memory is used only by data and a value indicating the size of the vector

Weaknesses:
- Slow element insertion or deletion: have to move aside up to N-1 other elements, linear time
- Slow merging of vectors: have to copy the two vectors to a new vector, linear time

Multi-dimensional arrays

Multidimensional arrays work in very much the same way as 1-D arrays.

Example 1: The following allocates an array that is 100x100 integer elements. As it is a primitive type, the data itself is allocated.
int[][] myarray;

myarray = new int[100][100];

Higher dimensional array are possible, but the curse of dimensionality begins to set in very quickly as memory usage is exponential in the number of dimensions.

Just like with basic arrays, arrays of non-primitive types allocate only the references to the objects. One issue with this approach to arrays is that the actual data for the objects cannot be allocated contiguously in memory. Only the references are guaranteed to be contiguous. Each object in the array must be allocated individually.

myClass[] x[][];
int i, j;

// allocate the references to the objects
x = new myClass[10][10];

// allocate the objects
for(i=0;i<10;i++) {
    for(j=0;j<10;j++) {
        x[i][j] = new myClass();
    }
}

**Partially allocated arrays**

Sometimes we don’t want fully populated multi-dimensional arrays. One common use is sparse matrices in mathematics.

int[][] myarray;

// allocate the row references
myarray = new int[100][];

// allocate the column references
myarray[0] = new int[50];
myarray[1] = new int[15];

The above snippet shows an example of allocating 50 ints for row 1 and 15 ints for row 2. The remaining rows still contain null.

**Explicit array organization**

A computer’s memory is linear, in the sense that each memory location is indexed by a single address, and the addresses form a continuous space from 0 to the memory capacity. Therefore, all multi-dimensional arrays are actually abstractions built upon a linear physical memory arrangement.

Sometimes we want to keep our data as a single linear block of memory, but we still want to access the memory as though it were a multi-dimensional array. We can do this by explicitly organizing the data within the linear array and calculating the appropriate location in the array when we want to access a piece of data.
For example, consider the task of creating a 20 x 30 2-D array out of a linear block of data. There are
20 x 30 = 600 memory locations required in this array, so we allocate a linear array of 600 elements.

```java
int[] arrayData;
int rows = 20;
int cols = 30;

arrayData = new int[rows * cols];
```
The data for the array is now ready to use (ints are primitive types). If we wanted to initialize all of the
elements of the array to a single value (e.g 42), we could treat the array as a 1-D array for the initialization.

```java
int i;
int size = rows * cols;
for(i=0;i<size;i++)
    arrayData[i] = 42;
```
If, however, we wanted to initialize each element to the sum of its row and column id, then we want to use
two for loops and index the array differently.

```java
int i, j;
for(i=0;i<rows;i++) {
    for(j=0;j<cols;j++) {
        arrayData[ (i * cols) + j] = i + j;
    }
}
```
The expression \((i \times cols)\) provides the index that is the first element of row \(i\). Adding \(j\) then indexes into the
proper column. Conceptually, all we are doing is laying out the rows of the matrix in order within the linear
array.
Java Documentation and Resources

There is extensive documentation of Java and all of the Java classes at:
http://java.sun.com/javase/reference/index.jsp

It’s a good place to start when you have questions. In addition, please email me or stop by. My office hours will be Mondays 2-4pm and Thursdays 9-11am, but please don’t hesitate to stop by at other times. There are also student assistants in the Olin lab Sunday-Thursday.

Array Example: Greenfield

Problem description: Simulate the macro-effect of small preferences by individual agents.

- Need a class, Greenfield, that implements a 2-dimensional grid on which agents can exist.
- Need a class, Entity, that represents an agent.
- Need a class, Simulation, that uses the Greenfield and Entity classes to execute a simulation.
- Need a class, Die, that enables rolling random numbers and is used by Simulation.

Die: Completed as part of the homework.

Entity: Needs to know its location (row, col) and identity (id). Needs accessors and mutators to get and set the values. Also has the legalMove function.

Greenfield: 2D array of Entity references, initialized to null. Needs a constructor with the size of the field specified, as well as accessors and mutators for locations on the grid.

Simulation: Needs a Greenfield, an array of class Entity, and three Die classes, one each for the number of rows, number of columns, and number of identity types.

- Constructor: allocates a Greenfield of the requested size, an Entity array of requested size, and called Initialize().
- Initialize: function that puts down N Entities in random locations and sets their ID to randomly selected legal values. Also initializes the Greenfield locations.
- Iterate: executes one iteration of the simulation. Goes through the Entity array and evaluates each Entity. Any Entity that is unhappy with its situation moves to a new random location. An Entity is happy if there are more like than unlike entities around it. If the Entity makes a move, then both the Entity and the Greenfield are updated.
- write: writes out a visualization of the field.
- main: initializes the simulation, writes out the initial map, executes K iterations, and writes out the result.
Polymorphic Types

Sometimes we want to have a collection of objects that may not be exactly the same type. Polymorphism lets us store references to one type of object (A) using a different type of object (B). The restriction is that the object we use to store the collection (A) must be either a parent B or an interface that B implements.

The Object class is automatically inherited by all classes in Java. Therefore, it is always possible to hold a reference to any object in a variable of type Object. The Object method has a useful method, getClass(), that returns the class of the object it holds.

```java
MyClass a = new MyClass();
MyClass c;
Object b;

// assign the MyClass object to the Object reference b.
b = a;

// copy the reference over to c
c = (MyClass)b;
```

You will likely use the Object class when creating an equals method for a class. You can make the argument be an Object reference, in which case the equals function will work properly when comparing objects of the same class, even if you pass in the object using a different type of reference than its base class.

Interfaces

An interface is like a class, but it has no fields or code for its methods. Think of it as a shell class that other classes can emulate. It defines the input and output of each method—and probably makes suggestions in the comments for what each method ought to do—but it does not require a particular implementation of the methods.

```java
public interface Simulation {

    // initialize: resets the simulation to an initial configuration
    public void initialize();

    // iterate: executes one iteration of the simulation
    public void iterate();

    // write: writes out a visualization of the simulation
    public void write();
}
```

A class can implement an interface if it has implementations of each method in the interface. It also needs to specify that it implements that interface in the class declaration.

```java
public class SimulationA implements Simulation {
    Greenfield green[];[];
    Entity individual[];
    Die rowdie;
    Die coldie;
    Die iddie;
```
public void initialize() {
    // code here
}

...
Polymorphism, why?

Why would we want one kind of object to be able to hold a reference to another kind of object?

- Sometimes we have many different objects that we need to hold in an array (graphics)
- Sometimes we need to execute the same procedure on different types of objects

Interfaces, why?

Why would we want to define an interface that only specifies the input/output form of methods?

- Sometimes we have generic algorithms that work well on many kinds of objects, but the algorithms need to execute certain methods on the objects in order to function (e.g. \( \land \) or \( \land \) are common). An interface specifies the minimum functionality required for the algorithm to execute (e.g. sorting or searching an array of objects).
- Sometimes we want to have different implementations for the same class. Some might be optimized for one situation, some optimized for a different situation. The interface lets other classes use the different implementations without need to know how they are different.

Inheritance

For example, if we wanted to have many different kinds of Entities in our simulation, we would probably still want to hold them in a single array. All entities are likely to require a location and an id field, so the Entity class from above is useful as a parent class. Let’s say that some entities had limitations on the distance they could move. We could create a new class, EntityLimit, that inherits Entity, but also has a field indicating the maximum distance the entity can move.

```java
public class EntityLimit extends Entity {
    int limit;

    public EntityLimit() {
        limit = 100000;
    }

    public boolean legalMove(int tr, int tc) {
        int distance = Math.abs(tr - this.r) + Math.abs(tc - this.c);

        return (distance > limit);
    }
}
```

Note that we are both extending the Entity class and overriding one of its methods.

In our Simulation class, we can now create a mixture of entities to place in our Entity array. We would want to change the initialization function so that initialize() created a mixture of entities, both Entity and EntityLimit. In the iterate function, which calls the legalMove() function to evaluate whether a move is legal
or not, as Java knows the true class of each individual, the proper legal move function will get called for each Entity or EntityLimit individual.

```java
// put each individual into the green
for(i=0;i<individual.length;i++) {
    // select a random location
    int ranrow = RandomInt.roll();
    int rancol = RandomInt.roll();

    while( green.getLocation(ranrow, rancol) != null ) {
        ranrow = RandomInt.roll();
        rancol = RandomInt.roll();
    }
    // now we have a good location
    // allocate the individual
    if( i % 2 == 0 ) {
        individual[i] = new EntityLimit(ranrow, rancol, identDie.roll(), 5);
    }
    else {
        individual[i] = new Entity(ranrow, rancol, identDie.roll());
    }
    green.setLocation(ranrow, rancol, individual[i]);
}
```

**Method overloading**

In addition to overriding methods from inherited classes, it is also possible to overload methods, which means to have methods with the same name but different arguments. The rule is that to overload a method, the new method must be differentiable from the original method by its arguments. The return type is insufficient to differentiate functions.
Linear Data Types

Stack

A stack is a simple method for storing and controlling access to data. Conceptually, it is the simplest data structure. The typical description is a stack of plates in the dining hall. Plates are put on the stack and plates are taken off the stack. But it’s always the top plate that is put on or taken off. You can usually see the top plate before you take it off, which lets you pick the stack with the cleanest plate on top.

A stack is a first-in last-out [FILO] data structure.

What are the methods that a stack needs to have?

- Push: put an object on the stack
- Pop: take an object off the stack
- Empty: is the stack empty?
- Peek: what is the top item on the stack?
- Size: how many elements are on the stack?

How would we implement a stack?

How about writing an interface for the stack data type?

```java
public interface StackInterface<T>
{
    public boolean empty();
    public void push(T obj);
    public T pop();
    public T peek();
}
```
Generic Classes and Interfaces

A stack interface is a relatively simple thing to write, as we only need to define a small number of methods and their IO specifications. The issue is that we'd like to have stacks for different kinds of variables. It would also be very nice to only have to write the stack code once, if at all possible.

Generic classes and interfaces let us parameterize class definitions. For example, we can define a GenericStack class using the following syntax:

```java
public class GenericStack<T>
```

This says that when we create an object of type `GenericStack`, we are going to specify a type for the objects to be stored in the class. For example:

```java
GenericStack<Float> s = new GenericStack<Float>;
```

Note that primitive types don't appear to be allowed as a generic type. However, because of the automatic casting that Java does for conversions from `float` to `Float` and vice-versa, this is not really an issue.

If we can intelligently write a generic stack class and all of its methods, then we can get away with writing only one implementation of the Stack class for any non-primitive data type. Note that we have to use the Object class to create a container for our data. Because of the way the generic interface was defined, it is not possible to create an array of type T (the data type of the specific implementation). In some other languages (e.g. C++) it is possible to do so.

**Interfaces:** Generic interfaces are also possible, and can be useful for defining data structures. A simple stack interface would begin with:

```java
public interface StackInterface<T> { ...}
```

Then it would define the method prototypes for a stack–push, pop, peek, empty–using the label T as the input or return type. The interface has no code. Any class that implements those functions can be used as a stack by another algorithm.

```java
public class StackInt implements StackInterface<Integer>
{
    int[] x;
    int next;

    public void push(Integer q) {
        x[next] = q;
        next++;
    }

    public Integer pop() {
        next--;
        return x[next];
    }

    public Integer peek() { return x[next-1]; }

    public boolean empty() { return next == 0; }
}
```

Some issues in stack implementation. Stacks are supposed to be very fast for insertion and deletion, because there is very little overhead within the data structure. But that speed comes at a tradeoff for flexibility and...
public class GenericStack<T> {
    Object[] x;
    int next;

    /** Constructs a stack of default size 100 */
    public GenericStack() {
        x = new Object[100];
        next = 0;
    }

    /** Constructs a stack with the given initial size */
    public GenericStack(int max) {
        x = new Object[max];
        next = 0;
    }

    /** Pushes an object on the stack */
    public void push(T q) {
        if(next < x.length) {
            x[next] = (Object)q;
            next++;
        } else {
            System.out.println("The stack is full");
        }
    }

    /** Removes the top object from the stack and returns it */
    public T pop() {
        if(next > 0) {
            next--;
            return (T)(x[next]);
        } else {
            System.out.println("The stack is empty");
            return (T)null;
        }
    }

    /** Returns a reference to the top object on the stack */
    public T peek() {
        return (T)(x[next-1]);
    }

    /** Returns whether the stack is empty */
    public boolean empty() {
        return next == 0;
    }
}
the ability to query information like the current size of the stack.

For example, if we want the stack to be able to handle any number of elements, then if we use an array-based implementation, at some point we may have to copy the array over to a new, larger array. That means that some call to push will take time linear to the number of objects in the stack rather than constant time. This could be very, very bad (think big robot arm moving quickly).

There is a way to implement a stack that does not require an array.
Reference Implementations

There is a way to implement a stack that does not require an array or re-allocating arrays when space has run out.

Consider first the information we really need to store if we want to implement a stack. The stack entity needs to know where the top of the stack is, how to add an item to the stack, and how to access the next element on the stack if the first element is taken off. How the stack elements are organized is irrelevant to the stack functionality.

An array-based implementation organizes the elements in consecutive locations in memory. The process for adding an item to the stack is to put it in the next free location in the array—which may require expanding the size of the array—and the method for finding the next element in the stack after popping off the top element is to look in the next location in the array. If the elements are not stored in consecutive memory locations, then each element needs to remember where the next element is stored. Thus, we trade the time-complexity introduced by having a fixed-size memory block to hold the data for the space-complexity that is required for each element to store a reference to the next element in the stack.

Thinking about the information required also tells us how to implement a stack without arrays.

**Stack Class:** The actual Stack class needs to know where the first element is and implement the methods push, pop, peek, empty, and size. It will need to explicitly keep track of the size of the stack by watching as elements are pushed and popped off the stack, because there will be no next pointer.

**Node Class:** The elements on the stack need to keep track of where the next element is. Therefore, we need to have a wrapper around each object on the stack that keeps a reference to the object and a reference to the next object.

**New concept:** Since no other class needs to know anything about the internal information of the Stack class, the Node class can be defined within the Stack class as a private class.

We can make our Stack generic by parameterizing the type used in the Stack and Node definitions.
public class PStack {
    private class Node {
        int data;
        Node next;

        public Node(int d, Node n) {
            data = d;
            next = n;
        }

        public int getData() { return data; }
        public Node getNext() { return next; }
        public void setNext(Node n) { next = n; }
        public void setData(int d) { data = d; }
    }

    private Node root;
    private int size;

    public PStack() {
        size = 0;
        root = null;
    }

    public void push(int d) {
        Node n = new Node(d, root);
        root = n;
        size++;
    }

    public int pop() {
        Node n = root;
        if(n != null) {
            size--;
            root = n.getNext();
            return n.getData();
        }
        return 0;
    }

    public int peek() {
        if(root != null)
            return root.getData();
        return null;
    }

    public boolean empty() { return root == null; }

    public int size() { return size; }
}
Queue

A queue is a first-in first-out [FIFO] data structure. It represents a line of objects where things go in one end and come out the other. Queues do not generally allow objects in line to be inserted, deleted, or accessed.

What are the methods we need for a queue?

- Push something on the back of the queue (add or offer). It might be nice to know if the push succeeded or failed.
- Take something off the front of the queue (remove, or poll). It might be nice to know if there was something to remove.
- Test if the queue is empty.
- Look at the first item in the queue.
- Find out how many items are in the queue (optional).

What fields do we need for a queue?

- Data
- Front
- Back
- Size

How would we implement a queue?

**Array implementation:** A little trickier than a stack because we have to keep track of the front and back of the array.

Front: keeps track of where in the array the next element to remove is located.

Back: keeps track of where in the array the next empty space in the queue is located.

When incrementing either the front or back index, we need to always account for the fact that the array is a fixed size and the indexes need to wrap around to the beginning. We can either use an if statement to test for that event or just always use modulo arithmetic when calculating the next location of either front or back.

```java
front = (front + 1) \% data.length;
```

If Back is equal to front, the queue is either full or empty. How do we keep track? There are a number of options, two are given below. Note that the two options require the same amount of information storage.

- Explicitly keep track of the size of the array
- Don’t let front wrap around to equal back when the array is full. Always leave one empty space in the array

**Linked implementation:** What information do we need to build a linked implementation (no arrays?).

The Queue needs to know where the front and back elements are, as well as keep track of the size of the queue.
Each Node needs to know where the next element in the queue is located. When one Node comes off the queue, it needs to tell the front reference where to point next.

When inserting the first element into the node, we have to explicitly catch that situation and set the front reference as well as the back reference.

When removing the last node from the queue, we have to explicitly catch that situation and set the back reference to null as well as the front reference.

On both removal and insertion, we always need to test for an empty queue.
Queue

Linked implementation: What information do we need to build a linked implementation (no arrays?).

- Keep references to the front and back elements and keep track of the size of the queue.
- Each Node needs to know where the next element (after it) in the queue is located.
- When inserting the first element into the node, set the front reference as well as the back reference.
- When removing the last node from the queue, set the back reference as well as the front reference.

On both removal and insertion, always need to test for an empty queue.

```java
public void insert(Agent val) {
    Node n = new Node(val, null);

    // point the current node at the back of the line to the new node
    if(back != null)
        back.setNext(n);

    // point the back of the line to the new node
    back = n;

    // if this is the first node in line, point front to it
    if( front == null )
        front = n;

    // increment size
    size++;
}

public Agent remove() {
    // check if the queue is empty
    if( empty() ) {
        System.out.println("Queue is empty");
        return null;
    }

    // get the node on the front of the queue
    Node f = front;

    // set front to the node after the current front
    front = f.getNext();

    // if this was the last node on the queue
    if( front == null ) {
        // set back to null as well
        back = null;
    }
    size--;

    // return the value of f, and f has no more references now
    return(f.value());
}
```
Modeling Life with a Queue

You are at the super huge expanded Q-mart trying to decide which checkout lane to use. You can use many different strategies to select which line to get in.

- Randomly pick one
- Scan all of the queues and pick the shortest one
- Scan some of the queues and pick the shortest one

Which strategy will get you through the lines in the shortest total time (search + standing in line)? If everyone uses the same strategy, is there an optimal strategy? If people use different strategies, is there still an optimal strategy for you?

How does the situation compare to having a single queue, with each checkout counter serving only one person at a time? (Same as the optimal strategy of picking the shortest queue.

Clearly, the situation requires using one or more queue data structures, but we also need to make a model of the overall situation.

- Model each checkout queue
- Model time per person to check out
- Model amount of time to inspect each line
- Rate at which agents enter the checkout lines

Consider a very simple simulation with $K$ checkout lines and a regular interval update.

- Each of the $K$ checkout lines is a single queue.
- Each agent takes 2 cycles to check out.
- Inspection time is negligible.
- Each cycle $K/2$ agents enter the checkout lines.

**Agent:** an Agent needs to have a strategy, keep track of the time it takes to get through the queue, and know how long it takes for it to go through the checkout process.

- timeEnterQueue: step on which the agent entered the queue
- timeExitQueue: step on which the agent completed the checkout process
- strategy: which strategy to use when picking a queue
- checkoutState: checkout steps left to go.

The major Agent function is pickQueue, which returns the index of the queue selected by the agent given the strategy.

**Queue:** The queue needs to be able to hold Agents.

**Simulation:** the Simulation needs to have a queue for each checkout line, a data structure to hold agents when they are finished, and a set of checkout locations to hold the agents in the process of checking out.
• AgentQueue[] line: an array of agent queues to represent the checkout lines.
• AgentQueue finished: a queue to hold the finished agents.
• Agent[] checkout: an array of agent references to hold the agents currently checking out.

The required Simulation functions are initialize() and step().

• initialize():
  – Allocate the array of queue references, then allocate each individual queue.
  – Allocate the checkout array.
  – Allocate the finished queue.
  – If you wish, put a few agents into the queues to initialize the system (optional)
• step():
  1. Scan over the checkout array
     (a) if the checkout line is not empty
        i. if the agent is done
           A. set the agent’s checkout time
           B. remove the agent from the queue
           C. insert the agent into the finished queue
           D. set the checkout location to null
        ii. else, decrement the number of checkout steps for the agent
  2. Insert K/2 agents into the queue using the pickQueue function.
  3. Scan over the checkout array. Any null checkout location gets a reference to the head of the queue (if not null).

List

Queues and Stacks are fine data structures, but they don’t give you access to any data except the two ends. The data is also not ordered or organized in any way except based on the order of insertion.
Priority Queue

What if we still had the concept of a queue, but we needed to differentially insert items into it. At an airport, for example, passengers for a plane that is close to leaving may be priority over passengers on later flights. In particular, the higher priority passengers get to go to the front of the line. However, within the set of passengers with priority, the queue is still first-in first-out.

**Algorithm:** Each agent has a priority rating. Starting at the back of the queue, find the first agent $p$ with the same or higher rating and insert the agent into the queue after $p$.

Array-based implementation:
1. Scan over each location $i$ from back to front.
2. Find the location in the queue for the new agent
3. Shift everything after the agent in the array and update the back and size fields
4. Insert the new agent

Linked implementation:
1. Test if the list is empty, in which case it gets pointed to by both front and back
2. Test if the new node should go first in the queue
3. Scan through the list looking for the first node with a lower priority
4. Insert the new node just before the first node with a lower priority
5. If there are no nodes with a lower priority, put the new node on the back of the queue
public void insert(int val, int p) {
    Node n = new Node(val, p, null);

    // find the place for the node going from front to back

    // check the empty case
    if( empty() ) {
        front = n;
        back = n;
        size = 1;
        return;
    }

    // check if the node goes in front
    if( p < front.getPriority() ) {
        n.setNext(front);
        front = n;
        size++;
        return;
    }

    // go through the rest of the list
    Node q = front;
    while(q.getNext() != null) {
        if( p < q.getNext().getPriority() ) {
            n.setNext(q.getNext());
            q.setNext(n);
            break;
        }
        q = q.getNext();
    }

    // check if the value gets inserted into the back of the list
    if( q.getNext() == null ) {
        q.setNext(n);
        back = n;
    }
    size++;
    return;
}
Unsorted List

An unsorted list is simply a collection of objects. However, a list ought to let us access nodes in the collection that are not at the front or the back of the queue. This requires some additional methods.

- public int get(int index): returns a reference to the element in the queue at location index.
- public int remove(int index): returns the element in the queue at location index and removes it from the list

```
public int get(int index) {
    if (index < 0 || index >= size || empty()) {
        return 0;
    }
    int i;
    Node p = front;
    for (i = 0; i != index; i++) {
        p = p.getNext();
    }
    return (p.getValue());
}

public int remove(int index) {
    if (index < 0 || index >= size || empty()) {
        return 0;
    }
    // check if we’re removing the first element
    if (index == 0) {
        return this.remove();
    }
    int i;
    Node p = front;
    for (i = 0; i != (index - 1); i++) {
        p = p.getNext();
    }
    // p points to the element before the one we want to remove
    Node n = p.getNext();
    p.setNext(n.getNext());
    // check if this is the last node
    if (n == back) {
        back = p;
    }
    size--;
    return n.getValue();
}
```
A list also lets us put a new element into the list wherever we want. Note the structure: empty check, front check, middle traverse, insertion, and back check.

Note that it is quite possible to eliminate the back pointer from a generally accessible list. We can always get to any node by traversing from front to back. The back pointer does permit insertions into the list and the back in constant time, however, which can be beneficial.

```java
public void insert(int index, int val) {
    Node n = new Node(val, null);
    if( empty() ) {
        front = n;
        back = n;
        size = 1;
        return;
    }

    // check front case
    if( index <= 0 ) {
        System.out.println("inserting in front");
        n.next = front;
        front = n;
        size++;
        return;
    }

    // search for the location to insert the node
    Node q = front;
    int i;
    for( i = 0; i < index-1 && i < size; i++ ) {
        System.out.println("going past value: "+q.value);
        q = q.next;
    }
    System.out.println("q is: "+q.value);

    // q points to the node before the insert
    n.next = q.next;
    q.next = n;
    size++;

    if(q == back) {
        System.out.println("q is back");
        back = n;
    }

    return;
}
```
Useful List Functions

What other useful list functions might be worth adding to a List class?

Things we already have:

- public void insert(E val) - inserts on the end of the list
- public E remove() - removes from the front of the list
- public boolean empty() - whether the list is empty
- public int size() - number of elements on the list
- public void insert(int index, E val) - inserts val into the list at location index
- public E remove(int index) - removes the element at location index

Things it might be nice to have:

- public void clear() - removes all elements from the list
- public boolean contains(Object o) - returns true if the list contains the element
- public boolean equals(Object o) - compares the list to the object for equality (list to list comparison)
- public int IndexOf(Object o) - returns the first instance of the object in the list or -1 if it doesn’t exist
- public int lastIndexOf(Object o) - returns the last instance of the object in the list or -1 if it doesn’t exist
- public boolean remove(Object o) - removes the object from the list, if it exists, returns whether any-thing was removed
- public Object[] toArray() - returns an array of all of the elements in the list in order
- public String toString() - returns a string representing all of the elements in the queue

Which of the above operations are easier/faster if we maintain a back pointer?

For which of the above operations would it be useful to have an internal (non-public) method that returns a reference to the Node at the ith location in the list?

private Node nthElement(int n) {
    if( empty() ) {
        return null;
    }

    Node p = front;
    int i;
    for(i=0;i != n && i < size;i++) {
        p = p.next;
    }

    return p;
}
Merge, Intersection, and Difference

How would we take two lists and merge them together? If we’re not concerned about making a copy, the operation just involves moving around some pointers

```java
// adds the second list onto the first list
// the operation does not make a copy of the elements on the list
public void merge(ListInt b) {
    // check if this list is empty
    if( empty() ) {
        this.front = b.front;
        this.back = b.back;
        this.size = b.size;
        return;
    }

    // check if the other list is empty
    if( b.empty() ) {
        return;
    }

    // connect the two lists
    this.back.setNext(b.front);
    this.back = b.back;

    // increment size
    this.size += b.size;
}
```

Intersection is more complex, as we need to know which elements are in both lists. The basic idea is to go through each element of one list and test if it is in the other list. If both lists contain the object, keep it. Difference uses the opposite rule: if only one list contains the element, keep it.

Question: does it matter which list we search for the matching element in? (It could if we had logN search)

Sorted list

The insert function for a sorted list looks a lot like the insert function for a priority queue. The only difference is that the elements being stored are compared rather than the priority ratings.
public void insert(int val) {
    Node n = new Node(val, null);

    // find the place for the node going from front to back
    // check the empty case
    if( empty() ) {
        front = n;
        back = n;
        size = 1;
        return;
    }

    // check if the node goes in front
    if( val < front.getValue() ) {
        n.setNext(front);
        front = n;
        size++;
        return;
    }

    // go through the rest of the list
    Node q = front;
    while(q.getNext() != null) {
        if( val < q.getNext().getValue() ) {
            n.setNext(q.getNext());
            q.setNext(n);
            break;
        }
        q = q.getNext();
    }

    // check if the value gets inserted into the back of the list
    if( q.getNext() == null) {
        q.setNext(n);
        back = n;
    }
    size++;
    return;
}
The Java ADT Interfaces

Java has a number of generic interfaces that provide a template for making abstract data types.

- Stack< E >
- Queue< E >
- List< E >

Some ADT interfaces have very few methods. For example, Queue has the following methods defined.

- E element() - equivalent to peek, returns, but does not remove the element at the head of the queue. Throws an exception if the queue is empty.
- boolean offer(E o) - inserts the element into the queue, if possible, and returns true if the operation succeeded.
- E peek() - returns, but does not remove, the element at the head of the queue. Returns null if the queue is empty.
- E poll() - removes and returns the item at the head of the stack, returns null if the queue is empty.
- E remove() - removes and returns the item at the head of the stack, throws an exception if the queue is empty.

Note, for example, that the functionality of the interface is somewhat simpler than what we defined for our generic queues. Peek, remove, and insert are the only operations defined. Your implementation can provide other operations, such as empty, clear, or size, but it does not have to do so to implement the interface.

The interface also makes a distinction between functions that throw exceptions if the queue is empty versus functions that return null.

The List interface, on the other hand, is much more complex. It permits searches, list merging, and set operations on lists (union and difference).
Linear ADT Concepts

What can we state about linear ADT implementations?

- Array implementations enable fast access to elements within the list
- Array implementations use less memory (implicit organization)
- Array implementations are easier to sort
- Linked implementations enable constant time push and pop operations
- Linked implementations are easier to make indefinite length
- Linked implementations are easier to merge

What can we generalize about linked implementations?

- Operations that modify the structure need to keep track of the element prior to the modification
- Operations generally have a structure: empty, front, middle, back
- Traversal is a common operation, not only within the data structure implementation
- Random access to interior elements is slow for linked implementations

The concept of having elements point to the next element in the data structured–distributed information organization–is very important. It permits many, many different kinds of data structures, including ones that model many situations in the real world.

- People and social networks
- The internet
Iteration and Iterators

The idea of iteration is common in data structures. Within a linear data structure the process is simple to do using a while loop and going from object to object in the list. In an array implementation, we would simply move from element to element in the array.

From outside the data structure, however, a programmer does not have access to the internal representation of the data structure (nor should they). We’d like to have a method that works regardless of the internal representation.

What information do we need to do a traverse?

- Know what value to return next
- Know how to get to the next element in the structure
- Know when the traversal is over

Because we often want to traverse the data structure and go through all of its elements. The task is so common, there is a Java interface defined for an object that permits traversal through a data structure: Iterator<

An iterator has only three methods: next, hasNext, and remove.

- T next(): returns the next element in the iteration.
- boolean hasNext(): returns true if there are more elements left to go.
- void remove() removes the last element returned by the iterator (optional operation).

A data structure is Iterable if there is an Iterator class defined for it. The Iterable<T> interface has only one method defined.

- Iterator<T> iterator(): returns an iterator over a set of elements of type T.

How would we define an iterator for a linked List of type Integer? The Iterator can be a private class within the ADT. The world only knows about the object through the Iterator<T> interface.
private class ListIntIterator implements Iterator<Integer> {
    Node curNode;

    private ListIntIterator(ListInt l) {
        curNode = l.front;
    }

    public Integer next() {
        if (curNode == null)
            return null;

        Integer returnValue = curNode.value;
        curNode = curNode.next;

        return returnValue;
    }

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

    public void remove() {
        throw new UnsupportedOperationException("remove() is not supported by this iterator");
    }

    // creates the iterator and returns it
    public Iterator<Integer> iterator() {
        ListIntIterator lit = new ListIntIterator(this);
        return lit;
    }
}
How would we use an Iterator for an Iterable class?

- Declare a variable of type Iterator<T>, or Iterator<Integer>, in this case.
- Assign the variable the result of calling the iterator() function on the ADT.
- While the iterator’s hasNext() function returns true, get the next value using iterator’s next() method:

```java
Iterator<Integer> lit = q.iterator();
while(lit.hasNext()) {
    System.out.println("Element: "+lit.next());
}
```

There are some handy shortcuts in Java that are available for ADTs that implement the Iterable<T> interface. The following shows how to execute a for loop over a data structure using a shorthand notation. Within the loop, the loop variable defined within the for statement holds the value of the current element in the ADT:

```java
for(Integer val: q) {
    System.out.println("Element: "+val);
}
```
Iteration and Iterators

Remove is a more difficult operation to execute, and it requires the iterator to keep around more information than just how to get to the next unit in the list. In particular, because it could potentially modify the first and last elements of the list, the iterator must be able to modify the list itself.

How much information does remove need?

- Has to be able to change the front and back pointers of the list.
- Has to have a reference to the element before the element returned by the last call to next.
- Has to know if next has been called at all.

Think about the traversal as going through a set of states. Assume the list is not empty (we can test for that since we have a reference to the List in the Iterator).

- Initial state: next has not been called
- First call: next has been called, remove would delete the first element in the list
- Traversal complete: hasNext() returns false

Note that First call and Traversal are independent of one another: both can be true, either can be true, or both can be false. Therefore, there are five states in which the iterator could be for a non-empty list (initial, first and complete, first and incomplete, not first and incomplete, not first and complete).

We have to be able to differentiate those states based on the values of the fields the iterator maintains. Consider, for example, if we maintain the state of the iterator using two references. Let curNode reference the node last returned by next() or be null. Let prevNode be the node just before curNode or null. Do we have enough information then?

- If both prevNode and curNode are null, then next has not yet been called (initial state)
- If curNode is non-null and prevNode is null, then next has been called once (first call)
- If curNode.next is non-null, then there are more nodes to traverse (middle)
- If curNode.next is null, then the traversal is complete (end)

Using two references we end up with plenty of possible states because we have the next field of curNode to work with in addition to the references themselves. Can we do it with only a single reference? Consider if we just keep prevNode: the node prior to the node whose value was returned by the last call to next.

- If prevNode is null, next has not yet been called (initial state)
- If prevNode is the front reference, then next has been called only once (first call)
- If prevNode.next is non-null, then there are more nodes to traverse (middle)
- If prevNode.next is null, then the traversal is complete (end)

prevNode gives us two states (null and non-null), prev.next can be equal to the front pointer or not if prevNode is not null (two additional states), and prev.next.next can be null or not if prev.next is not null (two more states). That gives us five states.
private class ListIntIteratorWithRemoveC implements Iterator<Integer> {
    ListInt L;
    Node prevNode;

    private ListIntIteratorWithRemoveC(ListInt l) {
        // have to have access to the list to handle remove the first and last elements
        L = l;
        prevNode = null;
    }

    public Integer next() {
        // check the empty and end of traversal cases
        if( L.empty() || !hasNext() )
            return null;

        // front case
        if( prevNode == null ) {
            prevNode = new Node(null, L.front);
        } else {
            prevNode = prevNode.next;
        }

        // return the value
        return( prevNode.next.value );
    }

    // remove the element that was last returned (prevNode.next)
    public void remove() {
        // empty check
        if( L.empty() ) {
            return;
        }

        // see if next has been called at all
        if( prevNode == null ) {
            return;
        }

        // check for the front of the list...
        if( prevNode.next == L.front ) {
            // move the front pointer forward
            L.front = L.front.next;

            // see if the front was also the back of the list
            if(prevNode.next == L.back) {
                L.back = null;
            }

            // set prevNode.next to the new front of the list
            prevNode.next = L.front;
        } else {
            // delete the node referenced by prevNode.next
            if( prevNode.next == L.back ) {
                L.back = prevNode;
            }

            // update prevNode
            prevNode.next = prevNode.next.next;
        }

        return;
    }

    public boolean hasNext() {
        return !L.empty() && ( (prevNode == null) || (prevNode.next != null) );
    }
}
Algorithm Efficiency

Why do we care about algorithm efficiency?

- Limited resources
- Some problems cannot be solved if they’re not solved efficiently

What is the difference?

<table>
<thead>
<tr>
<th></th>
<th>(n)</th>
<th>(\log n)</th>
<th>(\log^2 n)</th>
<th>(n\log n)</th>
<th>(n^2)</th>
<th>(2^n)</th>
<th>(n!)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>11</td>
<td>10</td>
<td>33</td>
<td>(10^4)</td>
<td>(10^4)</td>
<td>(10^9)</td>
</tr>
<tr>
<td>(10^2)</td>
<td>7</td>
<td>44</td>
<td>100</td>
<td>664</td>
<td>(10^4)</td>
<td>(10^{30})</td>
<td>(10^{94})</td>
</tr>
<tr>
<td>(10^3)</td>
<td>10</td>
<td>99</td>
<td>1000</td>
<td>9966</td>
<td>(10^6)</td>
<td>(10^{301})</td>
<td>(10^{1,435})</td>
</tr>
<tr>
<td>(10^4)</td>
<td>13</td>
<td>177</td>
<td>10000</td>
<td>132,877</td>
<td>(10^8)</td>
<td>(10^{3010})</td>
<td>(10^{19,335})</td>
</tr>
<tr>
<td>(10^5)</td>
<td>17</td>
<td>397</td>
<td>100000</td>
<td>1,660,964</td>
<td>(10^{10})</td>
<td>(10^{30,103})</td>
<td>(10^{243,338})</td>
</tr>
</tbody>
</table>

What is really important about algorithm efficiency?

- Time
- Space (memory)

Example: One way to make a data structure that is very fast at determining if something is in a set is to define a data structure that has a pre-allocated place for every possible thing that could be in the set. That gives constant time access to any possible element of the set, if it is in the set. The space efficiency of that algorithm, however, is poor.

How do we describe the complexity of an algorithm or memory usage? What is really important?

- Worst case
- Best case
- Average case

It turns out that all of the above can be important; worst case is usually the easiest to analyze. Many commonly used algorithms, however, have bad worst case performance but very good average performance, typically better than similar algorithms that have better worst case performance.

Counting

What are we really counting when we say steps in an algorithm? We are counting atomic operations on the computer—operations that generally take a fixed length of time.

What can a computer do?

- Store data: hold data in a memory location or long-term storage
- Manipulate data: execute operators on data
- Move data: transfer data from one location to another
- Control flow: control the flow of execution based on data
Computers have an instruction set that specifically outlines all the things a particular processor can do. A compiler converts a program written in a high level language into a program written in the computer’s instruction set. So there is a relationship between operations in a high-level language and the number of operations the computer actually executes.

At a very high level, each of the following generally qualify as a computing step.

- Assignment: moving or storing data
- Unary or binary operator: manipulating data
- An if-statement or other conditional test: control flow

From the point of view of finding asymptotic performance, any step in a program that does not depend upon the amount of data being processed takes constant time.

- Example 1: a++;
- Example 2: c = a + b;
- Example 3: c = (a - b) / (a + b);

However, we must be careful, because the following might not be constant time:

- Example 4: c = L.get(index) + L.get(index + 1);

In particular, the time it takes to get the elements at location index and index+1 may be linear in the number of elements within the list L.

**Asymptotic Performance**

Consider the task of summing up a series of numbers from 1 to N.

- For the number of elements in the series, for the value of the number, add one to a sum variable
- For the number of elements in the series, add the value of the number to a sum variable
- Sum = \( N(N + 1)/2 \)

Case 1: the number of operations ends up being \( o = n^2 + n + 1 \), as there are \( 1 + n(n + 1)/2 \) assignments and \( n(n + 1)/2 \) additions (not including the for loop operations).

Case 2: the number of operations is \( o = 2n + 1 \) as there are \( n + 1 \) assignments and \( n \) additions (again, not including the for loop operations).

Case 3: the number of operations if \( o = 4 \): one addition, one multiplication, one division, and one assignment.

When do we really care about performance issues, when N is small, or when N is large?

Big O notation: A function \( f(n) \) is of order at most \( g(n) \) if a positive real number \( c \) and a positive integer \( N \) exist such that \( f(n) \leq cg(n) \) for all \( n \geq N \). In other words, \( cg(n) \) is an upper bound on \( f(n) \) when \( n \) is sufficiently large.

In shorthand, we say that \( f(n) = O(g(n)) \) if the above condition is holds.
Worst-case Cost v. Amortized Cost

Worst-case cost is the longest time a task could possibly take. We could also think of it as the worst possible single event that could occur. Amortized cost, on the other hand, is the average cost given the worst possible sequence of events. Many times, the two are indistinguishable, as the amortized cost may simply be the result of the worst case event repeated over and over again.

Consider, however, the array-based list implementation where we double the size of the array any time it runs out of space.

Worst case: with \( N \) elements in the list, to add one more we have to allocate twice as much space and copy \( N \) elements to the new space. \( O(n) \) running time.

Amortized case: Consider how often each element is copied. If there are \( 2^k \) elements, then \( 2^{k-1} \) elements were copied the last time the array was increased, \( 2^{k-2} \) were copied the time before that, and so on down to \( 2^0 \) the first time the array was increased from length 1 to 2. So the total number of copies made during the entire sequence is a sum of powers of two.

\[
 n = 2^0 + 2^1 + \cdots + 2^{k-2} + 2^{k-1} = 2^k - 1 \tag{1}
\]

To find the average time per insert for \( n \) elements, we divide the total cost of inserting all of the elements by the number of elements inserted. In the worst case, the number of elements will be \( 2^{k-1} + 1 \), in which case the average cost will be just less than 2, or \( O(1) \).

Notation

When talking about bounds, sometimes we want to describe lower bound performance, upper bound performance, or have a tight description of performance.

- \( f(n) = \Omega(g(n)) \) - \( f(n) \) is of the same order as \( g(n) \) or higher.
- \( f(n) = \Theta(g(n)) \) - \( f(n) \) is of the same order as \( g(n) \).
- \( f(n) = O(g(n)) \) - \( f(n) \) is of the same order as \( g(n) \) or lower.

Showing that an algorithm is \( O(g(n)) \) means that the algorithm’s performance will never be worse than \( g(n) \). On the other hand, showing that an algorithm is \( \Omega(g(n)) \) means that an algorithm’s performance will never be better than \( g(n) \). Sometimes, both concepts are important.
Algorithms I: Searching

Searching is a common thing in many applications.

**Linear search:** go through the elements in a list, one-by-one, until you discover if the item is in the list.

- No alternative if the list is unsorted
- No alternative if random access in constant time is not possible (linked implementation)

**Binary search:** on a sorted array, start in the middle and see if the item is above or below that point. Repeat the process, throwing out at least half the data each step.

```java
public static boolean binarySearch(int[] data, int target) {
    int bottom = 0;
    int top = data.length - 1;
    while(bottom <= top) {
        int midpoint = (top + bottom) / 2; // integer math
        if( target < data[midpoint] ) {
            top = midpoint + 1;
        } else if ( target == data[midpoint] ) {
            return true;
        } else {
            bottom = midpoint + 1;
        }
    }
    return false;
}
```

Why is binary searching $O(\log n)$? If we always divide the data in half at each step—and we’re working with discrete, integer things—then at some point we hit a single element and we’re done. One way to think about it is: starting at one, how many times would we have to double the number of elements in the array to be equal to or greater than $N$? The answer is $p$ doublings for $N \leq 2^p$. The $\log_2()$ function is what gives us $p$ if we know $N$.

**Everything in the world array search:** $logn$ is a fast time for an algorithm, almost equal to constant time. The only way we can get faster is to find a constant time search algorithm.

Consider, for example, a situation where we have a finite number of things that can be in our list, and that only a single one of each type exists. For example, there are about 6.6 billion people in the world (July 2007) and each person is unique. We could, fairly easily, create an array with an entry for each person.

Now imagine that CS holds a colloquium talk and we want to keep track of who is at the talk. One way to do that is to create an array of type boolean with 6.6 billion entries, all initialized to false. When a person shows up at the talk, the entry for that person is set to true.

If we then want to search the list and see if someone attended the talk, all we have to do is use their index to access the array of boolean values. The process is constant time, since it is a simple lookup.

Question: what if we don’t want to assign people index numbers, but instead use their names? How could we organize the data to continue to achieve constant time lookup?
What if we make a function that just takes in a person’s name and produces a number from that?

- We end up with collisions, because there may be multiple people with the same name
- Solution 1: store all the people with the same name in an array, organized by a second, unique criterion.
- Solution 2: use more information to differentiate people: birthdate, zip code, height, etc.

The function that takes information in one form and transforms it to an index is called a hash function. The combination of an array and a hash function is called a hash table. A perfect hash function—one that assigns a unique identifier to each element—requires a single entry at each table. A imperfect hash function—one that may assign the same identifier to more than one element—requires some kind of data structure at each table entry to hold the duplicates.

Theoretically, an imperfect hash function creates a data structure that is $O(\log n)$ to search, assuming the data is sorted within the hash buckets and we can use a binary search. We could, for example, have a situation where all of the elements get put into one bucket and the whole data structure degenerates to a single sorted array. In practice, however, hash tables have access times close to constant. We have to be careful, however, to make the hash function fast. $\log n$ time is so fast (< 33 steps to search a 6.6 billion element array), that any hash function that takes more than 33 steps will actually be slower than a simple binary search.

The important lesson is that: $O(\log n)$ time is almost $O(1)$ time.
Algorithms II: Sorting

There are a number of different sorting algorithms.

Bubble Sort

Bubble sort is a simple algorithm that works on either arrays or linked lists.

```
SomethingChanged = true
while SomethingChanged
    SomethingChanged = false
    Traverse the list
        At each node, see if it needs to be switched with the next item in the list
        If a switch takes place
            SomethingChanges = true
```

How many times does bubble sort have to iterate?

- Consider the best case, when the items are already in order. One pass through the list confirms the ordering and the algorithm is complete (n steps).
- Consider the worst case, in which the items are in reverse order. The first pass moves the last element into place (n steps). The second pass moves the next to last element into place (n steps), and so on (n times n steps).

Bubble sort, therefore, appears to be fast for partially sorted lists, but it’s running time is $O(n^2)$. It’s never worse than $O(n^2)$, however, so it provides an upper bound on the performance of any sorting algorithm: any sorting algorithm that is worse than $O(n^2)$ is not worth considering.

Insertion Sort

We have already seen this sorting algorithm as part of an ordered list: put the items into a list such that, if the items on the list are sorted before the insert, then the items on the list are sorted after the insert.

How many steps does insertion sort take?

- Best case: items are already in order, so each insertion is one step (n steps).
- Worst case: items are in reverse order, so each insertion has to go to the end of the list $(n(n + 1)/2$ steps).

Insertion sort, therefore, is also $O(n^2)$, but has a lower constant than bubble sort, on average. Insertion also works with linked or array implementations equally well.

Merge Sort

Merge sort is a divide and conquer algorithm. The concept is simple:

- If the list has one element, it is sorted, so return the list.
- If the list has more than one element, divide the list in half and sort each half.
• Once the halves are sorted, merge the halves back together so they remain sorted.

How do we write an algorithm that implements this concept?

For an array of size four, consider what we would need to do:

• create 4 arrays of length 1
• create 2 arrays of length 2 that are pairs of merged and sorted 1-element arrays
• create 1 array of length 4 that is a pair of merged and sorted 2-element arrays

The algorithm for merging is straightforward:

```java
public int[] merge(int a[], int b[]) {
    int[] sortedArray = new int[a.length + b.length];
    int anext = 0;
    int bnext = 0;
    for(int k=0; k<sortedArray.length; k++) {
        if (bnext >= b.length || (anext < a.length) && (a[anext] < b[bnext])) {
            sortedArray[k] = a[anext];
            anext++;
        } else {
            sortedArray[k] = b[bnext];
            bnext++;
        }
    }
    return sortedArray;
}
```

It’s difficult, but not impossible, to write down the complete algorithm for an arbitrary size array. However a recursive solution is simple, as shown below.

The concept of recursion is important to merge sort. Recursion is when a function calls itself. In the case of merge sort, the algorithm calls itself but passes only half the data to the new instance of the function. That guarantees that the recursion will stop; at some point the data passed to the new instance will consist of only a single element.
/**
 * Recursive version of merge sort
 */
public void sort() {
    int[] result;

    result = recursive(x, 0, x.length-1);
    x = result;
}

public int[] recursive(int[] x, int start, int end) {

    // base case
    if( end == start ) {
        return new int[] {x[start]};
    }

    // otherwise
    int mid = (start + end) / 2;
    return merge ( recursive(x, start, mid), recursive(x, mid+1, end) );
}
Recursion

What is recursion? Recursion occurs when a function, method, or program calls itself. In other words, a series of instructions includes a call to execute the same series of instructions.

Why use recursion? Often, recursion is the simplest method of solving a problem. It naturally implements divide and conquer style strategies. Recursion can also produce very space-efficient code, which is why it was popular in the early days of computer science when program length was extremely important.

What are the important rules?

- The method must have an input, generally an input argument.
- The method must contain logic that analyzes the input and executes at least two cases.
- At least one of the cases must generate a result that does not require recursion: a stopping case.
- At least one of the cases must involve a recursive call to the function with an argument that makes progress towards the stopping case.

Infinite recursion occurs when there is no stopping case, or the recursive call does not make progress towards the stopping case.

A simple example: counting down

```java
public static void Countdown(int N) {
    System.out.println(N);
    if(N > 0)
        Countdown(N-1);
}
```

There are two cases in the control flow: call Countdown if N is greater than zero or return.

How does the system implement recursion?

A recursive call to a method or function is implemented just like any call to any method or function. When a method is called, the computer has to keep track of a number of pieces of information.

- The computer must know where to return in the calling method
- The computer must store the state of the calling method (all of its local variables)
- The computer must set up any arguments for the called method
- The computer must create local variables for the called method
- The computer must store any returned value from the called method and make it accessible to the calling method.
The System Stack

All computer code is stored in memory. The sequence of instructions in memory are stored sequentially, and the computer always keeps track of where the next instruction will come from. The variable that holds this information is often called the program counter.

Almost all computers maintain a system stack in order to manage function or method calls. When a method is called, the system puts the following information on the stack.

- The return value for the called method.
- The arguments for the called method.
- Where in the calling method to return when the called method is finished. Generally, this is represented by the current value of the program counter.
- Information about where the local variables for the calling method are stored in memory.
- Space for any local variables for the called method

The set of information placed on the stack is often called a stack frame. The stack frame is anchored at the location of the beginning of the local variables for the called function.

Consider the series of calls to Countdown, starting with $N = 3$.

<table>
<thead>
<tr>
<th>Memory location</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Return Address</td>
<td>address after call to Countdown</td>
</tr>
<tr>
<td>2</td>
<td>Old Stack Frame Pointer</td>
<td>address of old stack frame pointer</td>
</tr>
<tr>
<td>3</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Return Address</td>
<td>address of return call in Countdown</td>
</tr>
<tr>
<td>5</td>
<td>Old Stack Frame Pointer</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Return Address</td>
<td>address of return call in Countdown</td>
</tr>
<tr>
<td>8</td>
<td>Old Stack Frame Pointer</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>N</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>Return Address</td>
<td>address of return call in Countdown</td>
</tr>
<tr>
<td>11</td>
<td>Old Stack Frame Pointer</td>
<td>8</td>
</tr>
</tbody>
</table>

Consider a slightly more complex recursive function:

```java
public static int sumToNv2(int N) {
    int Q = 1;
    if(N > 1) {
        Q = sumToNv2(N-1) + N;
    }
    System.out.println("Returning "+Q);
    return Q;
}
```

The function `sumToNv2()` has a local variable `Q` that needs stack space to store it.
<table>
<thead>
<tr>
<th>Memory location</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Return value</td>
<td>ends up with 6</td>
</tr>
<tr>
<td>1</td>
<td>N</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Return Address</td>
<td>address after call to sumToNv2</td>
</tr>
<tr>
<td>3</td>
<td>Old Stack Frame Pointer</td>
<td>address of old stack frame pointer</td>
</tr>
<tr>
<td>4</td>
<td>Q</td>
<td>various</td>
</tr>
<tr>
<td>5</td>
<td>Return value</td>
<td>ends up with 3</td>
</tr>
<tr>
<td>6</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Return Address</td>
<td>address of println call in sumToNv2</td>
</tr>
<tr>
<td>8</td>
<td>Old Stack Frame Pointer</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>Q</td>
<td>various</td>
</tr>
<tr>
<td>10</td>
<td>Return value</td>
<td>ends up with 1</td>
</tr>
<tr>
<td>11</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>Return Address</td>
<td>address of println call in sumToNv2</td>
</tr>
<tr>
<td>13</td>
<td>Old Stack Frame Pointer</td>
<td>8</td>
</tr>
<tr>
<td>14</td>
<td>Q</td>
<td>1</td>
</tr>
</tbody>
</table>

When a method returns, the local variables are discarded, the old stack frame pointer is restored into the computer’s state, and the program counter is restored to the value from the stack. The calling method is then able to discard the arguments to the called function and access the return value from the stack.
Exam Review (just a little)

**Simulation Design: Hawks v. Squirrels**

We want to design a more sophisticated simulation that is going to let us create a more interesting scenario.

- Entities: squirrels, hawks, and trees, possibly other types of creatures
- Locations can hold many entities
- Many parameters for the system: birthrates, death rates, hunt success rates, etc.

**Major points**

- What are the things that all of the entities have in common? Location fields, type fields, possibly birthrate and death rate fields.
- Do we want an interface for entities?
- Where do we put the constants?
- How do we store the type of each critter?
- What data structure should the greenfield hold at each location?
- What are the methods the data structure needs to support?
- What kind of information do we want out of the system?

We might also like to simulate a little bit of evolution

- Instead of constant or random success rates for hunting and dodging, what if the success rates came from an entity’s parents with some additional randomness?
- We could track the average dodge rate and average success rate over time.
Recursion and Sorting

Merge Sort Review

Merging step: take two sorted lists and combine them into a single sorted list by taking the lesser value from the head of the two lists.

```java
public int[] merge(int a[], int b[]) {
    int[] sortedArray = new int[a.length + b.length];
    int anext = 0;
    int bnext = 0;

    for(int k=0;k<sortedArray.length;k++) {
        if( bnext >= b.length || ( (anext < a.length) && (a[anext] < b[bnext]) ) ) {
            sortedArray[k] = a[anext];
            anext++;
        } else {
            sortedArray[k] = b[bnext];
            bnext++;
        }
    }
    return sortedArray;
}
```

Division step: split the array in half and merge the two sorted halves.

```java
/**
 * Recursive version of merge sort
 */
public void sort() {
    int[] result;

    result = recursive(x, 0, x.length-1);
    x = result;
}

public int[] recursive(int[] x, int start, int end) {

    // base case
    if( end == start ) {
        return new int[] {x[start]};
    }

    // otherwise
    int mid = (start + end) / 2;
    return merge ( recursive(x, start, mid), recursive(x, mid+1, end) );
}
```
Quicksort

Quicksort is a method of sorting that can be executed in place, which means no additional memory is required. The idea is to pick a pivot value from the array and then sort the array into values less than the pivot and values greater than the pivot. By then recursing on the two halves, the algorithm is able to sort the entire array.

In merge sort, the merging aspect of the process is what takes time as the division step is trivial. In quicksort, the opposite is true: the division step takes time and the merging is trivial. It turns out this has a significant impact on the theoretical worst case run-time, but the algorithm is often used because its average performance is good and it works with no extra memory.

The recursive structure of the program is shown below in the doTheSort() function. The data is partitioned into two parts and the midpoint location is provided. Then the two halves of the array get sorted recursively. Merging the two halves back together does not require any more work, because the sorting is done in place.

```java
public static void quicksort(int[] data) {
    doTheSort(data, 0, data.length - 1);
}

public static doTheSort(int[] data, int bottom, int top) {
    if(bottom < top) {
        int midpoint = partition(data, bottom, top);
        doTheSort(data, bottom, midpoint - 1);
        doTheSort(data, midpoint+1, top);
    }
}
```

The partition function is where all of the work goes on. It divides the array into two parts: all of the values less than or equal to the partition value, and all of the values greater than the partition value. The partition value is arbitrarily selected to be the last element in the part of the array to be sorted (index top).

```java
protected static int partition(int[] data, int bottom, int top) {
    int pivot = data[top];
    int firstAfterSmall = bottom;
    for(int i=bottom;i<top;i++) {
        if(data[i] <= pivot) {
            swap(data, firstAfterSmall, i);
            firstAfterSmall++;
        }
    }
    swap(data, firstAfterSmall, top);
    return( firstAfterSmall );
}
```

```java
protected static void swap(int[] data, int i, int j) {
    int tmp = data[i];
    data[i] = data[j];
    data[j] = tmp;
}
```
Analysis of Quicksort

Quicksort seems like a very nice sorting algorithm. The structure of doTheSort() looks a lot like merge sort, which has an $O(n \log n)$ complexity. The important part of the algorithm, however, is the partition function. The partition function takes time $O(q)$, where $q$ is the number of elements passed into the partition function. If the partition function is able to divide the array in half each time, then each call to doTheSort() takes half as long as its parent, just like merge sort.

Another way to think about it, is as a tree. The first level of the tree partitions the whole array, which takes time $N$. The second level of the tree partitions the two halves individually, but still takes time $N$. The third level of the tree partitions the four quarters individually, but takes time $N$. Dividing the array in half each time, however, means that there are only $\log N$ levels to the tree, so an algorithm that divides its work in half at every step is $O(n \log n)$.

The problem with quicksort occurs when none of the values in the array are larger than the selected pivot point. For example, consider a sorted array. In every call to partition, none of the values will change places. Therefore, the midpoint returned by partition will be equal to the top value passed in to the function. Going back to the tree visualization, this situation produces an unbalanced tree. At each recursive step, all of the data except the rightmost value in the array gets passed to the first recursive call to doTheSort, while a single value gets passed to the second call. This means there are $N$ levels to the tree, as it takes $N$ recursive calls to doTheSort in order to read the termination condition.

Example:

Input array: [ 1 2 3 4 5 ]

__________
Bottom = 0, Top = 4
midpoint = 4
     |-----------------------------
     |     |
      Bottom = 0, Top = 3      Bottom = 5, top = 4
Midpoint = 3
     |-----------------------------
     |     |
      Bottom = 0, Top = 2      Bottom = 4, top = 3
Midpoint = 2
     |-----------------------------
     |     |
      Bottom = 0, Top = 1      Bottom = 3, Top = 2
Midpoint = 1
     |-----------------------------
     |     |
      Bottom = 0, Top = 0      Bottom = 2, Top = 1

What the tree shows is that quicksort, in the worst case, is $O(n^2)$. That occurs any time the array is sorted (in either direction). However, given randomly ordered arrays, it can be shown that the average run time for quicksort is $O(n \log n)$, which is what we would expect if the partition works well.
Recursion versus Iteration

All algorithms that can be written recursively can be be written iteratively. This must be so, because the programs run on a virtual machine that can only do one thing at a time. If the computer is executing the recursive algorithm, then the steps that it takes to execute the algorithm can be written down explicitly as a program.

All things being equal, an iterative algorithm is preferable over a recursive algorithm. The reason is the overhead incurred by a function call, which requires putting lots of extra information on the stack. The system stack may also not be large enough to handle deep recursion, in which case the program can fail or have unintended consequences.

In some cases, converting recursive algorithms to iterative algorithms is trivial. This is the case then an algorithm uses what is called tail recursion. Tail recursion is when the recursive call is the last step in the function.

If you consider what the recursive call is doing, it is storing the existing values of the local variables and parameters on the stack and making a call to itself using different parameters. However, since the recursive call is the last step in the algorithm, none of the current parameters or local variables will ever be used again. So why store them? Instead, we can substitute a while loop or for loop for the recursion and just overwrite the existing local variable and parameter values.

Example: Counting down

```java
// recursive version
public void countDownRe(int n) {
    System.out.println("countdown: "+n);
    if(n > 0)
        countDown(n-1);
}

// iterative version
public void countDownIt(int n) {
    while(n >= 0) {
        System.out.println("countdown: "+n);
    }
}
```

The while loop condition lets the contents of the loop run the same number of times as the recursive version. The output of the two functions is identical, but the recursive version takes more time and memory because of the function calls.

Example: Binary search

A more interesting example is binary search, which is also a tail recursive procedure.

```java
// recursive search
public boolean binSearchRe(int[] data, int target, int bottom, int top) {
    if(bottom > top)
        return false;
```
int midpoint = (bottom + top) / 2;

if( data[midpoint] < target )
    return binSearchRe(data, target, bottom, midpoint-1);
else if( data[midpoint] > target)
    return binSearchRe(data, target, midpoint+1, top);
else
    return true;
}

// iterative version
public boolean binSearchIt(int[] data, int target) {
    int bottom = 0;
    int top = data.length-1;

    while( bottom <= top ) {
        int midpoint = (bottom + top) / 2;

        if( data[midpoint] < target )
            top = midpoint-1;
        if( data[midpoint] > target )
            bottom = midpoint+1;
        else
            return true;
    }

    return(false);
}

Don’t be fooled by the fact that there are multiple calls to binSearchRef or that they aren’t the last line of code in the function. Because of the if statement, which allows only one of the options to be executed, the calls to binSearchRef are the last thing executed by the function, whichever path is taken.

### Non-linear data structures

Linear data structures are useful for many tasks, and are simple to implement and maintain. For some tasks, however, nonlinear data structures offer a better alternative. In general, nonlinear data structure are more complex to maintain but can offer better performance.

### Binary trees

A tree data structure is one that has the following structure:

- A **root** node that is the base of the tree
- Each node has the potential to have one or more **children**, of which it is the **parent**
- Each child is the root of a **subtree**

Note the recursive nature of the tree definition. Some more vocabulary:

- A node with no children is a **leaf** node.
• The **depth** of a tree is the maximum number of links that can be traversed from the root to some leaf node.

• The **level** of a node is the number of links from the root to that node.

• The **siblings** of a node are all of the other nodes that share the same parent

A binary tree is a subset of general trees where each node can have at most two children. Note that linked lists form a subset of binary trees where one child of each node is empty.

Formally, a binary tree is defined as

• an empty tree, or

• a node with a left subtree and a right subtree, each of which are a binary tree.

Compare this definition to that of a linked list:

• an empty list, or

• a node with a sublist, which is a linked list.

The definition for a node that holds type int is as follows:

```java
private class Node {
    int value;
    Node left;
    Node right;
}
```
Non-linear data structures

Linear data structures are useful for many tasks, and are simple to implement and maintain. For some tasks, however, nonlinear data structures offer a better alternative. In general, nonlinear data structure are more complex to maintain but can offer better performance.

Binary search trees

The definition for a node that holds type int is as follows:

```java
private class Node {
    int value;
    Node left;
    Node right;
}
```

Review insertion and searching

- In a binary search tree, things are in order
- Sometimes BSTs don’t allow duplicates (formal sets)
- Duplicates can be included, the decision about where to put them is arbitrary, but needs to be consistent

Tree traversal

- In-order
- reverse order
- by level with a queue
- in-order using an iterative algorithm (no recursion)
- using an iterator
Deletion

Removing a leaf

- As a leaf has no children, removing it has no impact on the rest of the tree
- The reference to the node in the parent is set to null

Removing a node with one child

- The children of the node to be removed should remain in the same subtree
- The parent of the node to be removed should point to the single child

Removing a node with two children

- This impacts the tree, as the parent of the node to be removed cannot point to both children
- We need a new node to replace the node to be removed
- The rightmost node in the left subtree is a good candidate (will have at most one left child)
- The leftmost node in the right subtree is a good candidate (will have at most one right child)
- In either case, remove the candidate node and connect its child (if any) to its parent
- Replace the node to be removed with the candidate

We treat the root the same as any other node, except we may have to update the root reference in the overall data structure.

A recursive solution: the idea is to recurse down to the node that needs to be removed, then call a function removeFromRoot that takes care of the details of removal and returns the replacement root, which may be null. The return value of each recursive call is the new root value, which allows the proper information to be propagated back up the tree.

- An interface function that calls the recursive remove function with the root of the tree
- The main recursive function
  - Check if the root is the one we want to delete. If so, call removeFromRoot
  - Check if the value might be in the left subtree. If so, recurse on the left subtree.
  - Check if the value might be in the right subtree. If so, recurse on the right subtree.
- The function removeFromRoot
  - If the node has two children, find the largest element of the left subtree (or smallest of right) and replace the data in the root node. Then remove the largest element of the left subtree. Both finding and removing are recursive calls. Return the new root.
  - If the node only has a right child, set the root to the right child and return the new root.
  - else the node only has a left child, or no children, so set the root to the left child and return the new root.
- The findLargest function recurses down the right children until it finds a node with a null right child, which is the return value for the base case.
• The removeLargest function recurses down the left children until it finds a node with a null right child. It then replaces the node with its left child (which may be null) and returns the new node value, which is the return value for the base case.

General Trees (Directed graphs)

The binary trees we’ve been studying are really called binary search trees, because they enforce an ordering on the data.

Trees don’t have to be sorted, and they don’t have to have just two children per node. General trees have many uses:

• Representing searches for computer games. Each node has as many children as there are possible moves.
  – Tic-tac-toe
  – Chess
  – Backgammon (branching factor is huge because of the dice rolls)

• The game of 20 questions
  – Each node is a yes/no question
  – The computer builds the tree by playing the game
  – If you stump the computer, it asks you to give it a question that differentiates it from its guess.
Heaps

Trees can also be sorted differently than a binary search tree. What if we used the rule that the root node was always equal to or larger than anything in either of its child subtrees? This is called a heap.

Let’s focus on just a few functions for this type of tree:

- void add(int val)
- int removeMax()

Rather than leave the shape of the tree to chance, let’s also make the requirement that the tree must always be complete. In other words, all of the leaves on one level must be filled before new leaves can be inserted, and new leaves always get inserted from left to right.

How can we implement an add function?

- We know some node has to go into the next leaf location
- What if we put the new node in the leaf location?
  - Most of the time it won’t be correct
  - But we can test if it should swap with its parent
  - If it should be swapped, make the swap and repeat the process

What is the complexity of the add function? $O(\log n)$

How can we implement a removeMax function?

- The root is always the one removed
- We need to get rid of the last leaf in the sequence
- What if we replace the root by the last leaf?
  - Most of the time the new root won’t be correct
  - Find the larger of the two children
  - Swap the root with the larger child
  - Repeat the process until the node sinks to its proper location

What is the complexity of the removeMax function? $O(\log n)$

How should we implement a heap? What about an array?

- A complete tree fits nicely within an array
- If we start the array at location 1, then the children of a node at index $i$ are at $2i$ and $2i + 1$.
- The root node is location 1
- The size of the tree is saved by the index of the most recent leaf added
- If the tree is empty, then lastIndex is zero
- To find the parent of a node, divide its index by two
Note that in the add function we don’t really need to swap. The new leaf starts out empty, so just shuffle the values down the tree until the empty spot is in the right location.

```java
public void add( int newEntry ) {
  // add a leaf
  lastIndex++;

  // start at the newly created leaf
  int newIndex = lastIndex;
  int parentIndex = newIndex / 2;

  // while the parent is smaller than the new entry, move up the tree
  while( parentIndex > 0 && newEntry > heap[parentIndex] ) {

    // put the parent value in the child location
    heap[newIndex] = heap[parentIndex];

    // move up the tree
    newIndex = parentIndex;
    parentIndex = newIndex/2;
  }

  // put the new entry in the open location
  heap[newIndex] = newEntry;
}
```

The removeMax function works in a similar way. We know the root will be removed, so it starts out as an empty slot with the just-removed leaf in a holding spot. Then we shuffle the largest child into the slot and work down the tree until we hit the right location for the old leaf.

A useful way to break down the function is to create a reheap() function that takes any heap with an out of place root (called a semi-heap) and converts it back to a heap.

```java
private void reheap(int rootIndex) {
  boolean done = false;

  // orphan is the out of place value we need to stick in the proper location
  int orphan = heap[rootIndex];
  int leftChildIndex = 2 * rootIndex;

  // while we aren’t done and aren’t at a leaf
  while( !done && (leftChildIndex <= lastIndex )) {
    // figure out which child is larger
    int largerChildIndex = leftChildIndex;
    int rightChildIndex = leftChildIndex + 1;

    if( (rightChildIndex <= lastIndex) && (heap[rightChildIndex] > heap[leftChildIndex]) ) {
      largerChildIndex = rightChildIndex;
    }

    // see if orphan can go into the current root index location
    if( orphan < heap[largerChildIndex] ) {
      // move the empty spot down the heap
      heap[rootIndex] = heap[largerChildIndex];
    }
  }
}
```
The overall removal method is simple, given the reheap() function.

```java
public int removeMax() {
    int root = 0;
    if( !empty() ) {
        root = heap[1];
        heap[1] = heap[lastIndex]; // put the last leaf in the root location
        lastIndex--; // shrink the heap
        reheap(1); // reheap
    }
    return(root);
}
```

What can you do with a heap?

- Implement a priority queue, with the heap ordered by priority. The removeMax function always removes the most important item from the queue.

- Implement a sorting algorithm
  - Put the elements of an array into the heap: \( O(n \log n) \)
  - Remove the elements of the heap in order and put them back into the array: \( O(n \log n) \)

It is possible build a heap out of an array of data without allocating more memory (assuming we can index the entries starting at 1, for now).

1. Begin with the parent of the last node (rootIndex = lastIndex / 2)
2. While rootIndex is greater than zero
   - execute the reheap() function using rootIndex as the root of the heap
   - decrement rootIndex
The algorithm is:

```java
// converts an array indexed from 1 to N into a heap
public static makeMaxHeap(int[] values) {
    // get the index of the last value in the array, which is also the last leaf
    int lastIndex = values.length - 1;

    for (int rootIndex = lastIndex / 2; rootIndex > 0; rootIndex--) {
        reheap(rootIndex);
    }
}
```

In actuality, the sorting can be done in place using the reheap algorithm.

1. Put the original array into heap ordering in place
2. Swap the first (root node) and last (last leaf) elements of the array
3. The last element of the sorted array is now in place
4. Reheap the rest of the array
5. Repeat from step 2 until all elements of the sorted array are in place.
Heaps

Review: Reheap Pseudocode

1. Input: rootIndex is the index of the root node that is out of place
2. orphan ← value of the root node
3. leftChild ← index of left child of rootIndex
4. while there is a left child
   (a) largerIndex ← index of larger child of rootIndex
   (b) if orphan is less than the value of the larger child
       • Put the value of the larger child into the rootIndex location
       • rootIndex ← index of larger child
       • leftChild ← left child of the new rootIndex
   (c) else break;
5. Put orphan into location rootIndex

The removal method, removeMax(), is simple, given the reheap() function.

```java
public int removeMax() {
    int root = 0;
    if( !empty() ) {
        // get the value of the root node
        root = heap[1];

        // put the value of the last leaf in the root location
        heap[1] = heap[lastIndex];

        // shrink the heap
        lastIndex--;

        // call reheap to fix the tree
        reheap(1);
    }
    return(root);
}
```

What can you do with a heap?

- Implement a priority queue, with the heap ordered by priority. The removeMax function always removes the most important item from the queue.
- Implement a sorting algorithm
  - Put the elements of an array into the heap: $O(n \log n)$
  - Remove the elements of the heap in order and put them back into the array: $O(n \log n)$
It is possible build a heap out of an array of data without allocating more memory (assuming we can index the entries starting at 1, for now).

1. Begin with the parent of the last node (rootIndex = lastIndex / 2)

2. While rootIndex is greater than zero
   - execute the reheap() function using rootIndex as the root of the heap
   - decrement rootIndex

The algorithm is:

```java
// converts an array indexed from 1 to N into a heap
public static makeMaxHeap(int[] values) {
    // get the index of the last value in the array, which is also the last leaf
    int lastIndex = values.length-1;

    for(int rootIndex = lastIndex/2; rootIndex > 0; rootIndex--) {
        reheap(rootIndex);
    }  
}
```

With clever use of memory, heapsort can be done in place using the reheap algorithm.

1. Put the original array into heap ordering in place (see above) $O(n \log n)$
2. Swap the first (root node) and last (last leaf) elements of the array $O(1)$
3. The last element of the sorted array is now in place
4. Reheap the rest of the array $O(\log n)$
5. Repeat from step 2 until all elements of the sorted array are in place. ($n$ times)

Given the guarantee of $O(n \log n)$ time for heapsort and the ability to execute the algorithm with no additional memory (unlike merge sort), it seems like heapsort is the sorting algorithm of choice. However, on average quicksort tends to be faster because there are simple ways to pick good pivot points and the overhead on quicksort is less.
Balanced Binary Trees

Heaps are nice for priority queues and offer strong guarantees on performance. However, searching a heap for a specific value is not efficient. Binary search trees offer the potential for good performance on searching, but only if they remain reasonably balanced after insertions and deletions.

A basic part of maintaining balance on a binary tree is the concept of a rotation. A rotation takes place when a child of a root node becomes the root and the root becomes a child.

Given: a root node A and its two children B and C

Left rotation

- The right child C of the root A will become the new root
- The current root A will become the new left child of C
- The left child of C becomes the new right child of A
- The left child B of the original root A remains the left child of A

Right rotation:

- The left child B of the root A will become the new root
- The current root A will become the new right child of B
- The right child of B will become the new left child of A
- The right child C of the original root A remains the right child of A

In both cases the tree remains a binary tree because the subtree that switches locations (left child of C in a left rotation or the right child of B in a right rotation) contains values that are between the values of A and C for a left rotation and between the values of B and A for a right rotation.

Rotations play a significant role in maintaining a balanced tree.
Balanced Binary Trees

AVL Trees

An AVL Tree maintains a balanced search tree by testing the tree after each insertion or deletion to see if it is still balanced. If the tree is unbalanced, then the balance is restored using a series of rotations. Any tree (or subtree) is balanced if its two subtrees differ in height by no more than 1.

Example 1: Add 60, 50, 20 to a tree. A rotation to put 50 in the root restores balance

Example 2: Add 50, 20, 60, 80, 90 to a tree. A rotation to put 80 as a new subtree root restores the balance

When working with AVL trees, we can assume that the tree is balanced before either an insert or delete operation. We just need to restore the balance after the insertion or deletion has taken place.

As noted above, rotations are guaranteed to keep the binary search tree properly ordered. The node about which we want to execute rotations is the node closest to the added (or deleted) node that exhibits an imbalance. In example 2 above, both nodes 50 and 60 are unbalanced after the addition of 90. Node 60 is closest to the newly added node, so the rotation occurs around that node.

Addition: There are four cases that need to be considered. Let N be the lowest node in the tree containing the new addition that is unbalanced.

1. The addition occurred in the left subtree of N’s left child
2. The addition occurred in the right subtree of N’s right child
3. The addition occurred in the right subtree of N’s left child
4. The addition occurred in the left subtree of N’s right child

Case 1: The left child of N is C. The addition occurred in the left subtree of C. Prior to the addition, the left and right subtrees of C had a depth equal to the right subtree of N.

- A right rotation about node N solves the balance issue
- C becomes the new root
- N becomes the new right child of C
- The old right child of C becomes the new left child of N

Case 2: The right child of N is C. The addition occurred in the right subtree of C. Prior to the addition, the left and right subtrees of C had a depth equal to the left subtree of N.

- A left rotation about N solves the balance issue
- C becomes the new root
- N becomes the new left child of C
- The old left child of C becomes the new right child of N

Case 3: The left child of N is C. The addition occurred in the right subtree of C, which has root G.

- A single right rotation about node N will not solve the problem
A left rotation about C that makes G the left child of N balances the subtree under C
A right rotation about N that makes G the new root balances the tree

Case 4: The right child of N is C. The addition occurred in the left subtree of C, which has root G.
A single left rotation about N will not solve the problem
A right rotation about C that makes G the right child of N balances the subtree under C
A left rotation about N that makes G the new root balances the tree

Deletion: The same four cases occur when deleting a node, although the location in the tree that is actually removed may not be the node that was deleted.

Note that it makes life much, much easier if you maintain a parent pointer for each node so that travel up and down the tree is possible without having to remember a node’s parent.

Red-Black Trees

AVL trees work to keep a tree balanced, but there is actually a simpler approach to maintaining almost balanced trees.

The rules for defining a red-black tree are:

- The elements of a red-black tree follow the same rules for ordering as a binary search tree
- Each node is either red or black
- The root is black
- A red node does not have a red child
- Every path from the root to a leaf of the tree must have the same number of black nodes

Red-black trees do not guarantee that the biggest difference between two subtrees is 1. Instead, they guarantee that the largest difference in path length is $2d$, where $d$ is the number of black nodes in the path. It is possible to show that this rule still makes insertion, deletion, and search $O(\log n)$.

Insertion: Similar to a heap or AVL tree, the idea is to insert the node and then repair the tree.

- If the item exists in the tree, do nothing (red-black trees don’t handle duplicates)
- If the item does not exist in the tree, add it as a red leaf

A red leaf may violate the rules for a red-black tree by giving a red node a red child. There are three possible cases here that depend upon the color of the node’s aunt and whether the node is a left or right child.

- If the new node has a red aunt, swap the colors of the parent, grandparent, and aunt. This solves the problem locally, but may push the problem farther up the tree.
- If the node has a black aunt, then if the parent and node are both on the same side (left children or right children), we can solve the problem completely by swapping the colors of the parent and grandparent and rotating the parent into the grandparent location.
- If the node has a black aunt and the node is an inner child, rotate the node into the parent location, making the node and old parent outer children, and fix the problem as above.
In the latter two cases, the root of the subtree is black and stays black, so no further changes are necessary. In the first case, the root of the tree changes colors, so the repair process needs to treat the grandparent as a new insertion and fix the problem.

Special cases:

- If the aunt of a node is null, treat it as a black node with respect to fixing the tree.
- If the node is the first node inserted into the tree, color it black.
- If the last node to be fixed is the root, color it black.
Balanced Binary Trees

Red-Black Trees

Review: the rules for defining a red-black tree are:

- The elements of a red-black tree follow the same rules for ordering as a binary search tree
- Each node is either red or black
- The root is black
- A red node does not have a red child
- Every path from the root to a leaf of the tree must have the same number of black nodes

Deletion: Delete the node and then repair the tree. The node of concern is the node that was spliced out in the deletion process. If, for example, the node to delete has two children, then the leftmost node of the right subtree gets spliced out and replaces the deleted node (and is given the deleted node’s color).

- Splicing out a red node will never cause problems.
- Splicing out a black node causes an imbalance

The cases to consider when deleting a black node depend upon the color of the node that was spliced out in the deletion process. N is the node that replaces the spliced out node. Consider N to be a black node if the spliced out node had no children.

- If N is red, color it black (done).
- If N is black and its sibling is black and has two black children, color the sibling red, which may cause problems for the parent. Consider the parent as N and address it at a level above.
- If N is black and its sibling is black and has an outer red child, give the sibling the parent’s color, the parent and outer child become black, and the sibling rotates into the parent location (done).
- If N is black and its sibling is black and has an outer black child and inner red child, a color change and a rotation convert it to the previous case.
- If N is black and its sibling is red, swap the parent and sibling colors, rotate the sibling into the parent location, and repair the original node N as a black sibling condition above.

Red-black trees are not trivial to implement, but there is a complete implementation in the book. Java’s TreeSet and TreeMap use red-black trees as the underlying implementation to guarantee \( O(\log n) \) run time for insert, delete, and search.
Graphs

Graphs are important for many practical applications in computer science

- Computer vision: representing relationships between parts of an image
- Robotics: representing relationships between locations on a map
- Mapping: identifying the shortest path between two locations
- Games: picking the best move by searching a graph of possible future moves
- Computer graphics: representing the relationships between objects in a scene
- Logistics: Picking optimal transportation paths between two locations
- Artificial intelligence: planning actions to achieve a goal
- Networking: routing algorithms and network analysis

Graph definitions

- Graphs consist of nodes, or vertices connected by edges
- Edges can be directed or undirected
- An edge can have a weight or cost associated with it
- A vertex can also have a weight or cost associated with it
  - Directed edges can only be traversed in the specified direction
  - In a directed graph, a vertex may have a directed edge back to itself
  - Any undirected graph can be converted to a directed by creating two directed edges for every undirected edge
- The neighbors of a vertex are those that can be reached traversing a single edge
- In a directed graph that is being searched, the vertices that can be reached from a given vertex are sometimes called its children
- A path is a sequence of edge traversals from a start vertex to a goal vertex
- The length of the path is the number of edges traversed
- The distance between two nodes the length of the shortest path between them
- The cost of a path is the sum of the costs associated with traversing the path
- A cycle is a path from a vertex back to itself
- In an undirected graph, a cycle may not use the same edge twice in the cycle
- A graph with no cycles is acyclic
- A directed graph with no cycles is a Directed Acyclic Graph or sometimes DAG
- A graph where it is possible to traverse from any node to any other node is connected
- A fully connected graph has an edge from every node to every other node
A dense graph is close to fully connected
A sparse graph has a small number of edges per vertex relative to the number of vertices

Graph Representation

How can we represent a graph, which can have an arbitrary number of edges per vertex?

Sparse Graphs: In sparse graphs the number of vertices per edge is small, or perhaps even a fixed number.

- City block navigation: each vertex has four neighbors
- Generic navigation: most intersections have between 1 and 5 choices of direction

The vertices are usually kept in an appropriate data structures

- If the number of vertices is changing, use an appropriate data structure to hold them
- If the number of vertices is fixed, use an array

For each vertex, use an appropriate data structure for the edges

- If the number of edges per vertex is fixed, or limited to a maximum number, use an array
- If the number of edges per vertex is flexible, use a variable sized array or a list

Dense Graphs: In dense graphs, each vertex may have as many edges as there are vertices in the graph.

- Think of the graph as an $N \times N$ matrix where $N$ is the number of vertices
- Each entry in the matrix can be a boolean value (edge exists or not)
- Each entry in the matrix can be a cost value (cost of traversing the edge)

The traveling salesman problem, for example, has a dense graph representation
Graphs

Graph Search

When traversing a graph, there are two generic methods: depth-first and breadth-first traversal.

**Depth-first traversal:** Follow the children of a vertex, one at a time, without backtracking or cycling, until the goal state or a dead end is reached. Then back up and try the next option. The order in which nodes are searched may be arbitrary, or may be determined by a function of the node or the edges.

**Breadth-first traversal:** Evaluate all of the children of a vertex before evaluating their children, without backtracking or cycling, until the goal state is reached or the graph has been exhausted. The order in which nodes are searched looks like traversing a binary tree in level order.

Both depth-first and breadth-first traversal can be executed using the same overall algorithm.

- Allocate an array of boolean to indicate if a vertex has been visited
- Create a list to hold the sequence of nodes visited
- Create a stack (DF) or queue (BF) to hold nodes yet to be visited
- Push the start node onto the stack/queue
- Mark the start node as visited
- While the stack/queue is empty
  - Pop the first node off the stack/queue -> curNode
  - Add the node to the traversal list
  - For each neighbor Q of curNode
    - If Q is not visited
      - Add Q to the stack/queue
      - Mark Q as visited
- Return the traversal list

Traversal of a graph can be converted into a search algorithm by testing to see if each node is the goal node. Each node also needs to keep track of its parent in order to trace the path back from the goal node to the root node.

Each type of search has different properties:

- **Breadth-first search** is expensive in terms of memory, but is guaranteed to find the shortest path to the goal, if one exists.
- **Depth-first search** is not guaranteed to find the shortest path from a start to a goal node, but it is memory efficient as only the current path needs to be stored.

Note that depth-first search is often implemented with a cutoff-depth, beyond which the search stops.

**Example: Hawk’s searching for food**

A Hawk looking for a squirrel doesn’t really teleport from one space to another space many squares away. We could model the Hawk’s behavior in a number of different ways:

- Have the Hawk randomly traverse a series of N edges
- Pick a goal location and have the Hawk search for a path to it
- Have the Hawk execute a breadth-first search around its current location
- Others?
Shortest Path Algorithms

There are many methods of discovering the shortest path from a root node to a goal node, if one exists.

- Breadth-first search (costly)
- Iterative deepening depth-first search: iterate DFS, increasing the search depth by one (reasonable)
- A* search: a hybrid of depth and breadth-first search that picks nodes to search based on their path length so far and the estimated distance to the goal
- Minimum spanning trees: convert an arbitrary graph into a directed acyclic graph that represents the shortest path from the root node to every other connected node in the graph
Graphs

Shortest Path Algorithms

There are many methods of discovering the shortest path from a root node to a goal node, if one exists.

- Breadth-first search (costly)
- Iterative deepening depth-first search: iterate DFS, increasing the search depth by one (reasonable)
- A* search: a hybrid of depth and breadth-first search that picks nodes to search based on their path length so far and the estimated distance to the goal
- Minimum spanning trees: convert an arbitrary graph into a directed acyclic graph that represents the shortest path from the root node to every other connected node in the graph

Dijkstra's Algorithm

This is a greedy algorithm, and basically a simpler, less general version of A*. Nevertheless, it is still proven to provide the shortest path between a source node and any other node in the state space.

Dijkstra’s algorithm builds a spanning tree, which is a graph with no loops that gives the shortest path from each achievable node in the state space to the source node. Given a spanning tree, the shortest path to any node is possible to determine in time proportional to the length of the path.
Algorithm:

- $G = (V, E)$: a graph with a set of vertices $V$ and edges $E$
- $S$: the set of vertices whose shortest paths from the source have already been determined
- $Q$: the remaining vertices
- $d$: the array of best estimates of shortest paths to each vertex (one element per vertex)
- $\pi$: an array of predecessors for each vertex (one element per vertex)

```java
shortest_paths(Graph g, Node s)
    initialize_single_source(g, s)
    S := { 0 } /* Make S empty */
    Q := Vertices(g) /* Put the vertices in a priority queue by distance from source */
    while not Empty(Q)
        u := ExtractCheapest(Q);
        AddNode(S, u); /* Add u to S */
        for each vertex v in Adjacent(u)
            relax(u, v, w)

    /* u is the newest node added to the spanning tree */
    relax(Graph g, Node u, Node v, double w[][])
        if g.d[v] > g.d[u] + w[u,v] then
            g.d[v] := g.d[u] + w[u,v]
            g.pi[v] := u

    /* Initialize the source node to a distance of 0, and rest to inf */
    initialize_single_source(Graph g, Node s)
        for each vertex v in Vertices(g)
            g.d[v] := infinity
            g.pi[v] := nil
            g.d[s] := 0;
```

At the end of the algorithm, $\pi$ contains all the information required to extract the shortest path from the root to each node.

**Example:** Robot navigation through Mudd 4th

Example: terminal-based operation with java

Go over the VisGraph class: HW assignment is Dijkstra’s plus a heap
Unix and XTerm Basics

Terminal based control of a computer gives you access to the information and programs on a computer with little or no filtering. There is no graphical user interface filtering what you see. With a little practice, you may find that you are much faster for any particular task using a command line than you are with a graphical user interface.

Unix-style operating systems exist across a wide spectrum of computers. Macs are Unix at their core, all linux machines support a traditional terminal, and the program cygwin provides a unix/terminal interface for Windows. If you know how to use a terminal, you can figure out what you need to know to use almost any computer.

A terminal has some kind of prompt at which commands are typed. Output on a terminal is generally scrolling text. In X-windows or on the mac you can also start GUI programs from the terminal that create new windows.

Terminal commands are typed on the command line. Most commands are some shorthand mnemonic. If you don’t know how to use a command, there are lots and lots of help pages, called man pages, (short for manual).

- `man <command name>` brings up the manual pages, if any exist, for a command
- `man -k <search string>` brings up a list of commands whose descriptions match the search string
- When reading the man pages, hitting Enter moves a line down, b moves a page up, space moves a page down, and q quits.

Most terminal commands have additional command line options that change their behavior.

- For most unix commands, an option is enabled by using a dash followed by a letter or a word indicating the option (e.g. `-v`)
- For some unix commands, the word version of an option requires two dashes (e.g. `--version`)
- For some unix commands (e.g. `tar`) the option does not require a dash
- Many commands have a `-h` or `-help`

Terminals support a number of nifty features that make your life easier

- Tab-completion: if you have typed part of a file or directory name, hitting tab will complete the name if the part you have typed specifies a unique filename.
  - If there are two files that share most of a name, the tab completion will fill in the string until the two filenames diverge (e.g. if `myfile0001.txt` and `myfile0002.txt` exist in a directory, typing `my` and then hitting tab will complete the name to `myfile000`).
  - Hitting tab a second or third time will list all the possible file completions for the string you have types.
- History: terminals remember what you executed before, and the up arrow and down arrow let you move through them.
- Terminals allow you to use a wild card character when describing files or directories in commands
- The files file1.txt, filewhat.txt, file.0.1.2007.txt would all be included by the 
  expression file*.txt.

- Terminals let you use emacs (or vi) text-traversal commands
  - cntl-f moves the cursor forward (right arrow)
  - cntl-b moves the cursor back (left arrow)
  - cntl-p moves backwards through the prior commands (like up arrow)
  - cntl-d moves forward through the prior commands (like down arrow)
  - cntl-a takes you to the start of a line
  - cntl-e takes you to the end of a line
  - cntl-d deletes the character under the cursor
  - cntl-k erases (kills) the rest of the line after the cursor and puts it in the kill buffer
  - cntl-y inserts a copy of the kill buffer (yanks it back) at the cursor

- Terminals let you send (pipe) the output of one command to either a file or another command
  - cat file1 file2 > file3 dumps the contents of file1 and file2 into file3
  - cat file1 | sort dumps the contents of file1 to the sort program which sorts lines of text in ascending order by ASCII value
Unix and XTerm Basics

Navigation and File Manipulation

- **ls** - list the files in a directory (-a see hidden files, -l see file information)
  - `ls -a` shows hidden files, those that begin with a period
  - `ls -l` shows file information such as permissions and file size
  - `ls -lh` shows the file information (e.g. file size) in human readable form
  - `ls myfile*` shows all the files that start with `myfile`

- **cd** `<directory path>` - change to a new directory
  - `cd ..` changes to the parent directory
  - `cd` with no arguments goes to the home directory

- **cp** `<source>` `<destination>` - copies the source file to a destination
  - `cp` with no option flags only copies files, not directories
  - If the destination is a directory, `<source>` can be multiple files
  - `cp -r` will copy a directory (recursively)
  - `cp -i` will ask first if the copy command is going to overwrite something

- **rm** `<file>` - removes the file
  - `rm` with no flags will not remove directories
  - `rm -r` recursively removes files and directories
  - `rm -i` asks before removing anything
  - `rm -f` doesn’t ask anything and returns no warnings (force)
  - `rm -rf` recursively removes things with no warnings

- **mv** `<source>` `<destination>` renames/moves the file from source to destination
  - If destination is a directory, source can be one or more files or directories
  - `mv -i` asks before overwriting anything

- **cat** `<file>` write the contents of one or more files to the terminal. Useful for looking at the contents of small files

- **less** `<file>` interactively writes the contents of the file to the terminal, but lets the user scroll through it interactively

- **tail** `<file>` shows the last few lines of a file, which is useful for looking at log files

Editors

- The two most common unix editors are `vi` (a sharp knife) and `emacs` (the lithium ion battery powered extendable swiss army knife with available AC adapter)
• Pick one and learn it (not that hard)
• Getting out of either editor:
  – In emacs, cntl-x cntl-c exits the program after writing the file to disk
  – In vi, :wq exits the program after writing the file to disk
Java Swing and Event-based Programming

Most modern applications are event-driven programs. You start up the program and it waits for you to do stuff. The main methods for most event-drive programs look very similar.

1. Initialize the program
2. Do until the user quits the program
   (a) Get the next event
   (b) Process the event
3. Clean up and Terminate

You can write an event-based program using just a terminal and scrolling menus, but most applications are based on a windowing system such as MacOS, Windows, or X. Windowing systems are complex, because there are lots of details involved in creating, drawing, and managing windows.

- Drawing windows
- Drawing simple objects: circles, lines, ovals, points
- Drawing fonts
- Clipping and redrawing of windows as they get moved forward and back
- Mouse actions
- Keyboard actions
- Selecting menu items
- Handling images and video
- Hiding and showing windows
- All the cool effects associated with the desktop

Windowing systems are intended to be general purpose, and because of that they leave a lot of the details up to the programmer, which can be tedious. As many applications look similar—create a new document, work with it, save it, print it, etc.—GUI frameworks like Swing, Carbon, or Cocoa handle many of the common details.

There are two things you can’t get away from, however, when creating an application:

- What is the layout of the window?
- What does each window element do?

The first item provides the interface design, while the second provides the program functionality. In Swing, the interface design is generated by creating objects like buttons, text fields, labels (static text), and drawing areas and specifying their relative locations in a window.

To provide functionality, each interface design elements (widget) has two hooks. One hook is a method that defines how to draw the widget, the other hook is a method that defines what the widget does in response to an event.
If your program defines the layout and functionality, Swing takes care of most of the rest of the details required to manage the application, including the main event loop. Therefore, a Swing program will generally look like:

1. Create a window
   (a) Define the widgets in the window and their layout
   (b) Define how the widgets draw themselves
   (c) Define what the widgets do in response to events
2. Tell Swing to execute the current layout
3. Tell Swing to make the window visible

When the window is created, Swing starts up what is called a thread that runs the event loop. This thread is a program running parallel to your main method, and it keeps on running after your code is finished. The event loop thread handles things like mouse clicks and calls the code you defined to implement the functionality of the widgets. The event-loop thread terminates when the user quits the program.
Java Swing and Event-based Programming

Process:

- **Layout**
  - Layout concepts: talk about Border and Flow
  - Panels, Fields, Labels, and Buttons
  - Hierarchical design to achieve the proper layout
  - setPreferredSize() method
  - Dimension(x, y) object

- **Panel drawing**
  - paintComponent method
  - Graphics object
  - Color object
  - setColor()
  - drawLine(x1, y1, x2, y2)
  - Coordinate system (upper left is (0, 0))
  - drawRect(x1, y1, dx, dy)
  - drawOval(x1, y1, dx, dy)
  - fill versions of Rect and Oval

- **Button actions**
  - ActionListener interface
  - Creating a private method with necessary information
  - actionPerformed(java.awt.event.ActionEvent evt) method
  - Connecting the listener object to the component

- **Drawing the field**
  - Grid size v. grid box size (need both)
  - Figuring out where to draw stuff: fillOval(i * dgrid, j*dgrid, dgrid, dgrid)
  - Deciding what to draw