Arizona State University

School of Computing, Informatics, and Decision Systems Engineering

# CSE 205 Lecture Notes

Ryan Dougherty
redoughe@asu.edu

*To my friends and family*

# Table of Contents

# Introduction

Welcome to these lecture notes for CSE205! This guide will follow up on the first lecture notes (for CSE110), and get more into the "algorithms and computation" side of programming. However, all examples will be in Java, to remain consistent between the two sets of notes. As we have said before, programming is becoming more of an important tool in the last few decades. Understanding of these technological shifts, therefore, is imperative.

The topics are split up according to the Table of Contents above, and their corresponding page numbers. At the end of each chapter are written and programming exercises. The written exercises are for testing your knowledge of the material, and the programming ones are to see if you can write programs using the material from that chapter.

If there are any questions/errors/comments that you want to send regarding the material, send an email to: Ryan.Dougherty [at] asu.edu.

# Useful Bits & Pieces

In this section we will give a few “miscellaneous” bits of information that are not big enough topics to warrant a whole section, but are nevertheless important. Knowledge from the first set of lecture notes is assumed.

## 2.1 String.split

There is a useful method in the `String` class called `split(String)` - it returns a `String` array (`String[]`) that consists of the `String` being called on without all instances of the passed in `String`. The passed in `String` is often called the “delimiter.” Here we give some examples:

```java
String original = "I am going to learn Java!";
String[] allWords = original.split(" ");
// allWords = {"I", "am", "going", "to", "learn", Java!"};
String[] words2 = original.split("a");
// allWords = {"I ", "m going to le", "rn J", "v", "!"};
```

## 2.2 ArrayList

Dealing with ordinary arrays in Java (or any programming language) is quite a hassle. Therefore, there is a class in the `java.util` package called `ArrayList`, which is an object wrapper for an array. It is a *templated* type, which means it has angle brackets to denote what type is allowed in the `ArrayList` (this type must be an `Object`, not a primitive type). Here are some examples of its use:

```java
ArrayList<Integer> ints = new ArrayList<Integer>(); // example
    of initialization - must be Integer, not int
ints.add(0);
ints.add(2);
ints.add(0); // can add 0 again
ints.add(1, 2); // add 2 at index 1
ints.remove(3); // remove element at index 3
int value = ints.get(1); // gets element at index 1
ints.set(0, 5); // sets index 0 to 5
if (ints.isEmpty()) {
    System.out.println("The array is empty!");
}
// for loop over the array
for (int i=0; i<ints.size(); i++) {
        System.out.println(ints.get(i));
}
ints.clear(); // ints has no elements
```

There is another class called `Vector`, also in the same package, that provides the same interface as `ArrayList`, but is thread-safe, whereas `ArrayList` is not. This means `Vector` is useful in multithreaded environments, as we will see in a later chapter.

# Interfaces, Abstraction, & Polymorphism

In this section we will start the "real" part of this second set of lecture notes. This particular section deals with some object-oriented design principles, and Java supports them very well.

## 3.1 Interfaces

An interface has the same structure as a class, but instead has the `interface` keyword where `class` would go, and only consists of two things: method declarations (the signature of the method plus a semicolon), and constants (with the `final` keyword). An interface is a "contract" of sorts, where some normal class will "implement" (via the `implements` keyword). By doing this, that particular class must implement every single method listed in the interface - otherwise, it is a compiler error. For example, if we created an interface for a `Ship`:

```java
public interface ShipInterface {
    public void shoot(); // just a declaration, no implementation
}
```

We can then implement this interface in any class we see fit:

```java
public class Ship implements ShipInterface {
    // ...
    // not including shoot() will have a compiler error
    public void shoot() {
        System.out.println("This ship did shoot!");
    }
    // ...
}
```

We can also implement multiple interfaces at once as comma-separated values:

```java
public class A implements Interface1, Interface2, Interface3 {
    // ...
}
```

## 3.2 Inheritance

Inheritance is a well-known technique in object-oriented programming to organize a lot of similar code in a hierarchical way. In this system, there are two kinds of classes: *children* (or *base*), and *parent* (or *super*) classes. The way it works is that the child classes inherit some information from the parent class. For Java, a parent can have one or more children, but a child can have at most one parent. Also, one may only inherit from a class if it is not declared as `final`.

A child class that inherits from a parent class gets all of the `public` methods, but none of the `private` methods (it does get `private` instance variables, however). So how do we enforce encapsulation for child classes then? This is where the `protected` keyword comes in. It has the same behavior as `private`, but any classes that inherit from it can use it. It is somewhat more “free” than `private`, but also has encapsulation.

In Java, in order to inherit from a class, one must use the `extends` keyword:

```java
public class A {
        // ...
}
public class B extends A {
        // ...
}
```

A way that inheritance is useful is for allowing code reuse for child classes. For example, one may use the `super` keyword to denote the parent method/constructor to call, and it calls the appropriate method/constructor in the parent class:

```java
public class A {
        public A() {
                // ...
        }
}
public class B extends A {
        public B() {
                super(); // calls A();
                // ...
        }
}
```

Also, for all of the instance variables declared in the parent class, one does not need to re-declare them in child classes. However, one must use the `this` modifier to do so:

```java
public class A {
        private int a;
        public A() {
                a = 0;
        }
}
public class B extends A {
        public B() {
            this.a = 1; // must use this modifier
      }
}
```

### 3.3 Abstract

Another keyword in Java is the `abstract` keyword. It is used for a class (that now has the `abstract` modifier) that says that "this method will be implemented in child classes, but I won't personally implement it." The `abstract` method in the parent class is just a method header (like would be in an interface, but now has the `abstract` modifier), and is a normal method in child classes. Here is an example:

```
public abstract class Person {
      public abstract double computePay();
}
public class CEO extends Person { // no abstract
      public double computePay() { // no abstract
            return 100000.0;
      }
}
```

### 3.4 Polymorphism

Now we get to the most important part of this section - polymorphism. It is a way, at runtime, for deciding which method to call when there is a hierarchy of classes, and is relevant to inheritance. For example, if we have the classes:

```
public class A {
      public void m() { ... }
}
public class B extends A {
      public void m() { ... }
}
public class C extends A {
      public void m() { ... }
}
```

We can develop code that has variables that can be one or more of these different classes without having to rewrite code:

```
A a1 = new A(); // works
B b1 = new B(); // works
C c1 = new C(); // works
A a2 = new B(); // ok, B has all attributes of A
// B b2 = new A(); // not ok, A does not have attributes of B
a2.m(); // calls B.m() even though a2 has declared type A
a1.m(); // calls A.m() because a1 is of type A
```

## 3.5 Written Exercises

1. **What is the meaning of declaring a class `abstract` and `final`? What do you think will happen?**

# Big-O Notation

In this short section we will introduce the idea of Big-O notation. It is a measure for analyzing asymptotic (i.e. as the input size "becomes large") behavior and time to run the algorithm.

## 4.1 Big-O

One concept that is often used in Computer Science is asymptotic notation, and Big-O notation is one of those. It is usually written of the form:

$$O(f(n))$$

where $f(n)$ is some function that depends on $n$. We will try to avoid the formal mathematics behind Big-O notation, but it essentially involves looking at an algorithm, and observing the behavior of it if the input is of size $n$. For example, if we were linearly searching through an array, on average, we will look at around $\frac{n}{2}$ elements - therefore, that will be the number of operations, on average, that we will do to run the algorithm. In Big-O notation, however, we discard any constant factors (or lower-order terms). Therefore, we say linear search runs in $O(n)$ time.

## 4.2 Example

Let's analyze the running time of Binary Search for Big-O notation. Each time we run through the loop, we effectively remove half of the search space. So, if the array size was $n$, we will perform $x$ steps, where $2^x = n$. Solving this equation for $x$ yields that $x$ is approximately $log_2 n$. Since $log_2 n = \frac{log(n)}{log(2)}$, binary search runs in $O(log(n))$ time.

## 4.3 Note

A few things to consider:

- Any operation that takes constant time is marked as $O(1)$.
- Any two operations with time $O(f(n))$ and $O(g(n))$ will, put together, have running time $O(f(n) + g(n))$. For example, $O(n) + O(n)$ corresponds to an algorithm that runs in time $O(n)$.

## 4.4 Best, Average, Worst Case

You may have noticed in trying to analyze other algorithms that there are multiple paths to take sometimes. How do you analyze this? You can in three ways: best, average, and worst cases. The best case corresponds to the absolute lowest amount of asymptotic time that is possible to complete the algorithm, and worst corresponds to highest. Average case is harder to study (how does one define "average"?), so we will leave this out. Most algorithm analysis does worst case, because it provides a guarantee on the running time - the algorithm is guaranteed to asymptotically run in less time than in the worst case.

### 4.5 Written Exercises

1. **What are the running times of {Selection, Insertion} Sort?**
2. **What is the Big-O notation of the function $f(n) = 3n^3 + 2n^2 + n$?**

# Linked List, Stack, Queue

In this section we will introduce 4 new data structures.

## 5.1 Linked List

A Linked List (`java.util.LinkedList`) is a very simple data structure. It consists of `Node`s that have a *data* member and a *pointer* which points to another `Node` (a "doubly linked list" has 2 pointers - i.e. one points "forward" and the other "backward"). This construction may be very similar to that of an array, but has distinct advantages:

– If one is at a particular `Node`, one can remove/add a `Node` in $O(1)$ time. This is not possible for arrays, which takes $O(n)$ time (shifting elements forward or back depending on the operation).

– The linked list does not need to be reallocated, since the pointer points to the next `Node`, which may or may not be near the current one in memory.

However, there are disadvantages:

– If one loses the beginning (sometimes called *head*) pointer, then there is no other way of accessing other elements in the linked list, since that beginning pointer was the only `Node` pointing to the second `Node`, which points to the third, etc.

– One needs to traverse the linked list to get to a specific index, since one cannot just access any given index - therefore, accessing a specific element takes $O(n)$ time (to traverse); for arrays, since we can index directly, this takes $O(1)$ time.

Despite these disadvantages, we will see two data structures - Stack and Heap - that can be implemented using a linked list in an efficient way.

## 5.2 Stack

A Stack (`java.util.Stack`) is a data structure that resembles a stack of books - you can insert books one at a time on the top of the stack, but you can only remove the top item from the stack one at a time. One cannot directly any element in the stack other than the top element (if it exists). A stack, therefore, is called a LIFO (last-in-first-out) data structure. The operations are as follows: `pop`, remove the top element and return it; `push`, add an element to the top of the stack; `peek`, see the top element without doing anything to it. All stack operations require $O(1)$ time, since they do not depend on the size of the stack.

A Stack is implementable using a Linked List, where the head pointer is the top of the Stack. When one adds an element to the top of the Stack, the head pointer changes to the new element, and the new element's pointer points to the original top element.

## 5.3 Queue

A Queue (`java.util.Queue`) is a data structure that resembles a line queue of people - people can be serviced from the front of the queue one at a time in order, and people can enter the back of the queue one at a time. A queue, therefore, is called a LILO or FIFO (last-in-last-out or first-in-first-out) data structure. The operations are as follows: `enqueue`, insert an element into the back of the queue; `dequeue`, remove the element at the front of the queue. All queue operations require $O(1)$ time, since they do not depend on the size of the queue.

A Queue is implementable using a Linked List, where one has a pointer to the front of the queue, and another to the back of the queue. It is much easier to implement a Queue using a doubly linked list than a singly linked list, but it is still possible.

## 5.4 Written Exercises

1. **What is the big-O notation for the operations on a {linked list, stack, queue}?**
2. **What do you think is the big-O notation for cost of inserting an element into an array/`ArrayList`? Assume, if needed, constructing a new array/`ArrayList` takes $O(1)$ time.**

# Recursion

In this section we will introduce the concept of recursion. Recursion is all about reducing source code complexity by calling the method in which the statements occur with a “smaller” parameter. We will give a motivating example: computing the factorial of an integer.

For recursion to work, we need two key factors: base case(s), and sub-problem(s). The base case(s) is/are what define when the recursion terminates, and handle the “simple” examples. The sub-problem(s) are all about assuming that the problem of the size given by the sub-problem(s) is/are solved. Then, all we need to provide is the code to move from the sub-problem to the initial problem. Choosing an appropriate sub-problem size is problem-specific, but common examples include decreasing the input by 1, or dividing by 2.

## 6.1 Example: Factorial

The factorial of an integer, written $x!$, is precisely $1 \times 2 \times 3 \times ... \times (x-1) \times x$. We could write a simple `for`-loop, as follows:

```java
public int factorial(int x) {
     int xCopy = x;
     int result = 1;
     while (xCopy >= 1) {
          result *= xCopy;
          xCopy--;
     }
}
```

But there are too many cases for the program to give the wrong output.

For example, the base cases for factorial are: $0! = 1$ and $1! = 1$. An appropriate sub-problem for factorial is $(x-1)!$. Moving from $(x-1)!$ to $x!$ is easy: all we need to do is multiply by $x$, since $(x-1)! \times x = x!$. Since we have both the base cases and the sub-problem, we can now define a recursive version of factorial:

```java
public int factorialRecursive(int x) {
     // handle base cases first
     if (x == 0 || x == 1) { // can also use x <= 1
          return 1;
     } else { // handle subproblem step
          return factorialRecursive(x-1) * x;
     }
}
```

As one can see, if we did not have a base case, then the method would keep calling itself over and over without terminating. Therefore, when doing recursion, **always have a base case**.

## 6.2 Example: Fibonacci Numbers

The Fibonacci numbers are defined as follows: $F_0 = 1, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. The first few Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, and so forth. Informally, each number is the sum of the previous two. Again, we could create a `for`-loop to find the $n$th Fibonacci number, but we can simplify with recursion.

So what are the base cases? They were given to us: $F_0 = 1, F_1 = 1$. So what is the sub-problem? We actually have two here: $F_{n-1}$, and $F_{n-2}$. To get to $F_n$, all we need to do is to add $F_{n-1}$ and $F_{n-2}$. Therefore, we can easily create a recursive version for the Fibonacci numbers:

```
public int fibonacciRecursive(int n) {
     if (n == 0 || n == 1) {
          return 1;
     } else {
          return fibonacciRecursive(n-1) +
                 fibonacciRecursive(n-2);
     }
}
```

## 6.3 Written Exercises

1. **Is it possible to take any recursive function/program and convert it into some kind of iterative loop (i.e. a `for`, or `while` loop)?**
2. **What is the running time of the fibonacci algorithm?**
3. **Find a way of optimizing the recursive fibonacci algorithm (hint: "keep track" of pre-existing solutions by using an array).**

# Trees

In this section we will introduce the trees data structure. It allows efficient lookup but not much larger storage space for $n$ data elements. Then we will introduce several subclasses of trees.

## Trees

We will think all elements of a tree as “nodes.” In a tree, there is a “root” node that is the top of the tree. Any node can connect to any other node in the tree, such that there are no cycles in the tree (i.e., one can traverse some set of nodes and get back to where one started). All nodes that a node is connected to are called the “children” of that node. Any node that does not have any children is called a “leaf node.” Any non-leaf and non-root nodes are called the “internal nodes” of the tree. For a general tree, there can be arbitrarily many children per node.

## Binary Trees

A binary tree is defined as a tree that has a maximum of 2 children for each node. One can think of each node as a “sub-tree” of the tree that has a root as that node. Therefore, each node in a binary tree is a root of a sub-tree of the whole tree. Since each node has at most 2 children, then we can think of one as the “right” node and the other as the “left.” For an implementation, a particular node is very easy:

```java
public class Node {
     Object data;
     Node rightNode;
     Node leftNode;
}
```

## Traversals

Now that we have a binary tree, how do we traverse the tree? There are three ways of doing so:

– Pre-order: root-left-right

– In-order: left-root-right

– Post-order: left-right-root

We will explain what each of these items mean; take pre-order for example. Starting at the root node, we process the root’s data, then we recursively call the pre-order function on the node’s left child; when that is complete, we recursively call the function on the right child. For example, the following would be an implementation of a pre-order traversal:

```java
public void preOrderTraversal(Node node) {
     // first call to preOrderTraversal has node == root
     processData(node.data); // process the node's data
     if (node.leftNode != null) {
          preOrderTraversal(node.leftNode);
     }
     if (node.rightNode != null) {
          preOrderTraversal(node.rightNode);
     }
}
```

The other two traversals are defined and implemented very similarly.

**Binary Search Trees**

## 7.1 Written Exercises

1. **Is it possible to take any recursive function/program and convert it into some kind of iterative loop (i.e. a `for`, or `while` loop)?**
2. **What is the running time of the fibonacci algorithm?**
3. **Find a way of optimizing the recursive fibonacci algorithm (hint: "keep track" of pre-existing solutions by using an array).**

# Heaps

In this section we will introduce the heaps data structure. It is a special type of binary search tree that is used for many data structures.

## Heaps

A heap is defined as a binary search tree such that it has the following properties:

- The tree is complete: this means that each row of the tree is completely full except for possibly the last row.
- There is some constraint on the elements.

For example, the constraint for adding an element to the heap is that the two child nodes are less than or equal to the parent. This example is called a “max” heap (i.e., the root is the maximum element in the heap).

## Heap Operations

Heaps support the following operations:

- Adding an element to the heap.
- Finding the maximum value in the heap.
- Removing the maximum value in the heap.

We now detail each of these operations and how they work.

## Adding an Element

To add an element $x$ to a heap, we do the following:

- Add $x$ as a leaf such that the heap is still complete.
- Move $x$ toward the root, exchanging positions with its parent, until the heap property says that $x$ cannot move up the heap any more.

## Removing Largest Element

To remove the largest element from a heap, we do the following (assuming we have a max heap):

- Remove the root and reconstruct heap from the 2 disjoint sub-heaps.
- Move the last leaf of the heap to be the new heap of the tree.
- Move this new root down the heap as needed until the heap property says that it cannot move down the heap any more.

## HeapSort

A while ago we covered various sorting algorithms. There actually is a very fast algorithm for sorting using the heap data structure. The way that it works (informally) is that it adds each element from a list into a heap, and then removes them one at a time. The largest element comes off the heap first, and then after re-"heapifying" (fixing the heap), the entire sequence of removing elements will be in descending order.

The following is how the algorithm works:

– Build a heap from the array (as a max-heap) of size $n$.

– Repeat the following $n$ times: extract the root, put it aside.

– The order of extraction is a sorted array, in ascending order. If we instead wanted an array in descending order, we use a min-heap.

The key to how this algorithm works is the re-"heapifying" step. The following is how that algorithm works:

– Repeat the following until both children are larger or are at the bottom of the heap:

– Compare the node and its children. If at least one of them is larger than the node, swap it with the larger child; otherwise, we terminate.

– If we don't terminate, we go back to the first step with the node's new location.

## HeapSort in Java

The following is an implementation of HeapSort in Java, which uses many of the concepts described above.

```java
public class HeapSorter {
     private int[] a;
     public HeapSorter(int[] anArray) {
          a = anArray;
     }
     // Sorts the array managed by this heap sorter.
     public void sort() {
          int n = a.length - 1;
          // for each node that has at least one child node
          for (int i = (n - 1) / 2; i >= 0; i--)
               fixHeap(i, n);
          while (n > 0) {
               swap(0, n);
               n--;
               fixHeap(0, n);
           }
     }
     private void swap(int i, int j) {
          int temp = a[i];
```

```
            a[i] = a[j];
            a[j] = temp;
    }
    private void fixHeap(int rootIndex, int lastIndex) {
        int rootValue = a[rootIndex];   //Remove root
        // Promote children while they are larger than the root
        int index = rootIndex;
        boolean more = true;
        while (more) {
            int childIndex = getLeftChildIndex(index);
            if (childIndex <= lastIndex) {
                // Use right child instead if it is larger
                int rightChildIndex = getRightChildIndex(index);
                if (rightChildIndex <= lastIndex &&
                    a[rightChildIndex] > a[childIndex]) {
                    childIndex = rightChildIndex;
                }
                if (a[childIndex] > rootValue) {
                    // Promote child
                    a[index] = a[childIndex];
                    index = childIndex;
                } else {
                    // Root value is larger than both children
                    more = false;
                }
            } else {
                // No children
                more = false;
            }
        }
        // Store root value in vacant slot
        a[index] = rootValue;
    }
    private static int getLeftChildIndex(int index) {
        return 2 * index + 1;
    }
    private static int getRightChildIndex(int index) {
        return 2 * index + 2;
    }
}
```

A straightforward analysis shows that the worst-case running time of HeapSort is $O(n \times log(n))$ for an array of size $n$.

## 8.1 Written Exercises

1. **Is it possible to take any recursive function/program and convert it into some kind of iterative loop (i.e. a `for`, or `while` loop)?**
2. **What is the running time of the fibonacci algorithm?**
3. **Find a way of optimizing the recursive fibonacci algorithm (hint: "keep track" of pre-existing solutions by using an array).**