Data Structures and Algorithms

By: Kartik Kapur

Contents

Chapter 1

Introduction

1.1 A few words

This purpose of this textbook is to provide students with the foundations that they need to understand Data Structures and Sorting Algorithms. This text will have conceptual and practical problems that will range from beginner difficulty to one of mastery. Because this text is meant for students of Berkeley's CS61B, the first chapter of this book will be an introduction to Java. Following this, most of the concepts in this reading will be independent of language; however, code excerpts will still be in Java. All the code used in examples and in practice tests is compiled and available on `https://github.com/KartikKapur/DataStructuresAndAlgorithms/tree/master/Code`.

This textbook is still in progress, and modifications that I have in mind that are not yet implemented can be seen here: `https://github.com/KartikKapur/DataStructuresAndAlgorithms/issues/1`. If you feel that some particular section needs to be emphasized more or there needs to be a new section all together feel free to email me or just comment on the above link.

1.2 How to Approach this Text

This text will not serve as a replacement for lectures, or its equivalent, rather it is supposed to be a supplemental reading meant to strengthen your foundations and give those who seek an extra challenge the ability to do so. I took the UC Berkeley's Computer Science 61B Lectures, relevant text from Algorithms 4th edition, and my own experience and knowledge to compile this textbook. Essentially, this text serves as a condensed version of lectures and contains the vital information.

For maximum efficiency, I suggest doing the readings after the lecture to make sure you properly understood what was going on. The problems should be done provided at the end of every chapter will serve as a benchmark for each set of major concepts, these are considered easy to midlevel problems. At the end of the textbook, you will find 2 practice midterm 1's (on java), 2 practice midterm 2's (on data structures), and 2 practice finals. These summarize the contents of the book and will be challenging, probably more challenging than the course you are currently taking. These exams are meant to be difficult, especially in the time provided. I suggest that you do them with partners in order to make sure that you understand what is going on. All the problems and practice tests will come with descriptive walkthroughs. I hope to be able to make more materials available in the future.

1.3 Acknowledgments

Josh Hug's Lectures and content are the base for which this study resource was built on. Additionally, I used some relevant aspects of *Algorithms 4th Edition* by Robert Sedgewick and Kevin Wayne as inspiration for some sections.

Chapter 2

Java

2.1 Classes and Objects

Java is an *Object-Orientated Language* in which every file must a *class* or *interface*. Functions within these files is called *methods*. Classes in Java are powerful because we can create a skeleton for many different objects instead of repeating ourselves. Let us take an example of 2 wildcats Jeremy and Josephine.These two wildcats have all the exact same characteristics except their name is different. The naive approach to representing Jeremy and Josephine would be to make a separate class for each one. However, Java lets us do something much better, we can create a new *instance* of the class instead of just rewriting it. Here is an example of how to instantiate 2 objects.

```
1 Wildcat jeremy = new Wildcat("Jeremy");
2 Wildcat josephine = new Wildcat("Josephine");
```

We now know that we can instantiate classes; however, we can have various types of data and methods inside of our classes. Below is a list of what we can store inside of classes

- **Instance Variables:** These are variables that are different for each instance of a class.
- **Static Variables:** Static Variables are variables that are common to all instances of a class. This means that if the static variables changes value in one instance of the class, it will change in all in all. instances.
- **Instance Method:** In order for an instance method to be called, you must call it from an instance of the object.
- **Static Method:** Static methods can be called directly through the class
- **Constructor:** The constructor is a "method" that is used in order to instantiate a class. Whenever you instantiate a class, the arguments that are passed in are what get passed into the constructor.

  •***Important note: Static Methods cannot have any references to instance variables or methods. However, instance methods can reference static variables and methods. More on this in the next lesson.***

Below is an example class, see if you can identify all its components.

```
1 public class Human{
2       public int weight;
3       public static int joy = 100;
4       public Human(int mass){
5           weight = mass;
6       }
```

```
7       public void eat(){
8             weight +=5;
9             System.out.println("I love eating");
10      }
11      public static void play(){
12            System.out.println("I have" + joy + "happiness ");
13      }
14  }
```

Weight- Instance Variable, Joy- Static Variable, public Human- constructor, eat- instance method, play- static method.

2.2 Static Typing vs Dynamic Typing

As we foreshadowed earlier, there is a difference between when static methods can be used as opposed to Instance (dynamic) methods. Let us take the example of the human class from 1.1. This is a possible series of calls:

```
1  Human.play();
2  Human kartik = new Human(5);
3  kartik.eat();
4  kartik.play();
```

Note that before we called kartik.eat(), we had to instantiate kartik to be a human. Another important note to make is that we can still call play on kartik even if play is static. If we attempted to call *Human.eat()* we would end up with a compilation error.

So far, we have not had any difficult problems. Rest assured, in a later lesson, this concept will become much trickier. Make sure that you understand the difference between Static and Dynamic typing for now.

2.3 Primitives and Reference Types

In Java, a variable can be either a *Primitive* or a *Reference type*. There is a concept of bits, which is not discussed too much in 61B, but the main concept is that bits are a unit of measurement to see the amount of space that something takes up. Whenever you declare a variable, of either type, Java sets aside some memory for it.

The Golden Rule of Equals is given two variables x and y, if we set y equal to x, all the bits from x are copied to y.

```
1  y = x;
```

In Java there are 8 primitive types, below I have listed the types, a brief description, and how many bits they use.

- **byte** - a unit of data that is 8 binary digits long. 8 bits.
- **short** - An integer from -32,768 to 32,767. 16 bits
- **int** - An integer from -2,147,483,647 to 2,147,483,648. 32 bits.
- **long** - An integer from -9,223,372,036,854,775,807 to 9,223,372,036,854,775,808. 64 bits.
- **float** - From 3.402,823,5 E+38 to 1.4 E-45. 32 bits.
- **double** - A decimal number. 64 bits.
- **boolean** - True or False value. 1 bit.
- **char** - A sole character. 16 bits.

If something in Java is not a primitive then it must be a reference type, or the instance of an Object. Whenever you instantiate an object, Java stores its **address** in 64 bits. This has an incredibly important consequence. If we take our prior example where we set y equal to x, copying the bits of a reference type would not result in creating a duplicate object like it would for primitives, rather we would just copy over the address to the object. ***This is a very important distinction to make.***

2.4 Inheritance

Inheritance is the idea that a specific object can have all the behaviors and properties of its parent. In order for something to inherit from a parent, you must use the *extends* keyword. For example:

```
1 class Superhero{...}
2 class Batman extends Superhero{...}
```

In the above example, the Batman Class will have all the attributes of a Superhero. Additionally Batman will be able to have its own attributes. This leads us to a key point. The is-a relationship. The Class that extends a parents class "is-a" version of the parent class (in this case Batman is a Superhero). However, a Superhero is not Batman because the Superhero class does not extend Batman. This makes intuitive sense because not all Superheroes have to necessarily be Batman.

We know that when a class inherits from another, it inherits the methods, but a subtle point is that when you instantiate a child class, an implicit call to *super()* is made in the child constructor as the first call. This means that if the parent class has a constructor that takes in no parameters, its contents will get run before the the child's constructor. If you wish to run a parent constructor that has arguments, simply write *super(your arguments here)*. To call any other parent methods from a child's method use *super.parents method name(parent's method arguments);*. There are other cases where you do not have to necessarily use the keyword super. For example, if a class does not have a method but its parent does then you do not need to use the word *super*. In more solid terms, if the class Batman extended the class Superhero and the class Superhero had a method called *suitUp* that Batman did not then calling *suitUp* (without a *super.* in front of it) in a Batman instance would result in the Superhero class's method *suitUp* getting called.

We have gone over class inheritance, now we will discuss interface inheritance. For interfaces, we use the word *implements* instead of extends.

```
1 interface Plumber{...}
2 class Mario implements Plumber{...}
```

Similar to a class, the interface creates an is-a relationship. That a class that implements the interface is-an instance of the interface.

Though on the surface, it may seem that extending classes and implementing Interfaces do the same thing, they serve very different purposes, and it is important to see the distinctions. One of the biggest differences is that when a class extends another class, it does not have to re implement any methods. Assuming you did not have the same method signature in your subclass, calling a method will result in the parent's method being called without a compilation error. Interfaces on the other hand serve more as a blueprint for code that needs to be implemented. The only exception to this rule is if a method in the interface is declared with the *default* keyword and has a body. Another distinction between the two philosophies is that a class can only *extend* from one class; however, it can *implement* multiple interfaces.

Now that we have gone over inheritance, we can bring back static and dynamic types. When we are declaring a variable, the class on the left is the *Static Type* and the class on the right is the *Dynamic type*. When declaring a variable, you must make sure that the item to the right is-a version of the item on the left of the equal sign. ***This is a reference class for the next few examples***

```
1 public class Fellow{
2     String name;
```

```
    public Fellow(){
           this.name = "I'm nameless"; }
     public void breathe(){
            System.out.println("Breathing noise");
     }
}
public class Professor extends Fellow{
    public Professor(){}
    public void teach(){ System.out.println("I taught"); }
}
```

Now let's walk through a few function calls:

```
Fellow josh = new Professor();
System.out.println(josh.name);
josh.breathe();
josh.teach();
```

1st line creates a Professor whose static type is a Human, 2nd line prints "I'm nameless", 3rd line prints "Breathing noise", 4th line is a compilation error.

In the above example, *josh.teach()* gave us a **Compilation Error** (error at compile time) because even though josh's dynamic type is a teacher, its static type is a human. When making a method call, Java checks to see if the method exists in the static type– if it doesn't it raises a compilation error. To get around this, we can use a trick called casting. The idea of casting, is that we believe we know information about some object that the computer does not so we tell the computer to trust us and we make the computer temporarily view some object under a different static type. To make the above code work we would write:

```
(Professor josh).teach();
```

In this case, the code does work; however, if the method does not exist in the casted type, we will get a runtime error. For example in the following sequence of function calls we would get an error:

```
Professor bruce  = (Professor) new Fellow();
(Professor bruce).teach();
```

The reason we would get an **Runtime error** (error during runtime) is because we told the computer we were instantiating bruce to be a professor; however, in reality bruce was just a fellow. During compile time, the computer thought bruce had the teach method because it thought bruce was a professor; however, during runtime, because bruce was only a fellow, it could not find a teach method to execute and it errored out.
The moral of the story is to be careful with casting!

2.5 Generics

Let's say that we want to have some unknown object be a variable inside of a class. Inside of the class header, we can declare unknown types that we know will be variables. Take the following class as an example.

```
public class DictionaryItem<K, V> {
    K key;
    V value;

    public DictionaryItem(K key, V value) {
        this.key = key;
        this.value = value;
    }
```

```
}
```

It is not totally obvious why this would be useful, why use this when we can just have a normal class that looks cleaner? The reason why can be seen below.

```
DictionaryItem<String, String> webster =
new DictionaryItem<String, String>("Kartik", "Kapur");
        DictionaryItem<Character, Integer> numberDict =
        new DictionaryItem<Character, Integer>('a', 1);
```

If we want to have different types of a Dictionary, we may not want to rewrite the whole class. This is why generics are so useful.

Similarly, if we want to make just one method have a generic type, we can do that with a similar declaration.

```
public static<Key, Value> Dictionary <Key, Value> add(Key k, Value v){...}
```

The initial <Key, Value>refer to the Dictionary return value and tell us what types will be in the Dictionary. The second <Key, Value>refers to the types that are the being passed in for the arguments into the method. Generics can be tricky, so it is important to keep track of when and how you declare.

2.6 Comparables and Comparators

It's pretty straightforward how numeric values are compared in Java. Integers can be greater than, less than, or equal to each other. However, what if we wanted to compare different objects such as porcupines? There is no straightforward way that we can compare porcupines; you may want to compare them by how large they are, how many spines they have, or their names. In order to solve this problem, we can utilize comparables and comparators.

Comparable is an interface that can be implemented in a class. When implementing comparable, you must write an *int compareTo(T o)* method. The Object that is passed in is what is being compared to the current instance. Frequently, the compareTo method will return a negative number if the current instance is considered to be "less than" the object passed in, a positive number if it's "greater than" the passed in instance, and 0 if the 2 objects are "equal".

```
class FatCat<T> implements Comparable<T>{
    int weight;

    public FatCat(int weight) {
        this.weight = weight;
    }

    public int compareTo(T o) {
        if (!(o instanceof FatCat)) {
            return 1;
        }
        return this.weight - ((FatCat) o).weight;
    }
}
```

In the above example note how we have to check if the object passed in is an instance of Human. If we do not make this check, we may end up with a runtime error (because the passed in object may not be a FatCat). Additionally, note how we need to cast the passed in object to be a human, if we do not do this, we will not be able to access the weight instance variable because the computer is not smart enough to realize that the object is a Human.

Instead of comparables, we can use **comparators**. Comparators are also an interface that need to be implemented. Unlike comparables which have a *compareTo* method, comparators have an *int compare(T o1, T o2)* method. To modify the prior example, we can just modify the header to be

```
class FatCat implements Comparator<FatCat>
```

and the compareTo method would be replaced with

```
public int compare(FatCat o1, FatCat o2) {
        return o1.weight - o2.weight;
    }
```

We did pretty much the same thing that we did for comparables; however, if you notice, the header for our class has *Human* in it. What does this mean? It means that the generic object that Comparator passes in will be of Human type. This means that we no longer have to perform and *instanceof* check, as we know all of the objects passed in will be humans.

So how do we know when to use a comparator as opposed to a comparable? Generally, when there is a natural ordering, or an ordering that will happen the majority of the time, it is recommended to use comparables as you will always be comparing one instance of a certain class to another instance of that same class. A comparator is mainly used for when you want to create an ordering for a one time use case or when you do not have access to edit the class you are trying to compare. These are not hard and fast rules and frequently there is no "right answer" on when to use a comparator vs a comparable so don't sweat about it too much.

2.7 Autoboxing and widening

Autoboxing is a recent feature that was implemented in Java that allows for implicit conversions between wrappers and primitives (i.e int to Integer and vice versa). One of the few times that autoboxing does not work is when we declare an array variable.

```
int[] testArr1 = new int[6];
int[] testArr2= new Integer[6]; //does not compile
```

Another concept that we will briefly touch on is the conversions between a smaller item to a larger. For example, a variable that has a type int may be able to be passed in for a double argument, but a double cannot be able to be passed in for an int. This can be thought of as "widening". This works because doubles have a larger range than ints. Think of it as trying to fit into a shirt- you can fit into a large shirt no problem, it'll just be baggy, but if you try to fit in a tiny shirt you may cause some problems.

2.8 Exceptions

If a program depends on some sort of user input, it is sometimes inevitable that we will have an error in our program. In order to minimize the damage that such an error does, we can use *try*, *catch*, and *finally*. The try statement allows us to run a program and if an exception is thrown that matches the exception a catch statement is looking for, we will move to the catch block. When an exception is caught, your program will no longer error out. After everything is completed, we visit the finally block. The finally block is very special because regardless of what happens, it will be run. If the program ends up erroring out due to an uncaught exception, finally will run right before exception is thrown. See if you understand what's going on in the code snippet below.

```
try {
   int[] arr = new int[3];
   int index = 0;
   while (true) {
      arr[index] = index;
      System.out.print(index);
```

```
7       }
8  } catch(ArrayIndexOutOfBoundsException E){
9          System.out.print("Phew got out of there");
10 } finally {
11 System.out.print("I need a nap");
12  }
```

0 1 2 Phew got out of there I need a nap
Something important to note is that you must have the proper exception caught. If the above catch block was replaced with

```
1  catch(NullPointerException E){
2      System.out.print("Wrong exception");
3  }
```

The exception would not get caught and we would instead have printed "0 1 2 I need a nap java.lang.arrayindexoutofboundsexception".

For one try statement, you can have more than 1 catch block, this makes it so a variety of errors can be caught.

There are two types of exceptions that can be thrown in a program, *checked exceptions* and *unchecked exceptions*. Checked exceptions are exceptions that need to be declared in the method header. The purpose of this is to prevent you from compiling your code so that you cannot have a runtime error. To declare that you may be catching a checked exception, use the *throws* keyword. This is not to be confused with the *throw* keyword which is used to actually throw an exception. An example of throws and throw is below.

```
1  public static void oinkos() throws IOException{
2  ...throw new IOException("protein power")...
3  }
```

Note throws will always be in the header of a method and throw will always be in the body of it.

Unchecked Exceptions, on the other hand, do not need to be declared in the method header. Unchecked exceptions crash at runtime, and are the exceptions that are typically used for our purposes.

2.9 An Introduction to Data Structures

There are a few introductory data structures that we will go over which will serve as a base for many of the concepts that we go over later on in this book.
The first data structure that we'll briefly discuss is an ***array***. The array is a built in data structure in Java that is a fixed-size collections of elements of the same type. To declare an array we do as follows:

```
1  int arr = new arr[8]
```

The above array will be have 8 available spots that can contain the int type. In order to instantiate one item in the array, we do as follows.

```
1  arr[3] = 5;
```

This will make the 4th item, the one at the 3rd index, to be 3- note that the first element in an array starts at the index 0. Attempting to access an item that has not yet been instantiated yet will result in a *NullPointerException* and trying to access an item in the array whose index is not in the array would result in an *ArrayOutOfBoundsException*. Below is an example of what would result in a *NullPointerException* followed by an *ArrayOutOfBoundsException*.

```
1  int i = arr[0];
2  arr[8] = 10;
```

Say that we feel like adding many elements to a sequence, but we don't want to resize our array over and over. To fix this we may wanted to use a linked data structure. In this type of data structure, each item would have a *first* and *rest* variable. The first would be the actual element stored, and the rest would be the rest of the list. The typical API for these is as follows.

```
void addFirst(int x);
int getFirst();
int size();
void addLast(int x);
int getLast();
```

Note that the above example is tailored towards ints and it can be made to hold any type using generics. To have pointers to various nodes, we could use a private helper class within our Linked Data Structure class- once again, this is tailored to ints for simplicity, but we can make it hold generics.

```
private class Node{
     int item;
     Node next;
     public Node(int i, Node n){
          item = i;
          next = n;
     }
}
```

A problem that one can realize while coding is the *addLast* method is that you will need to have a special case to deal with. With all these special cases, we tend to get unwieldy code that just is not practical. To deal with such a problem, we will briefly discuss the concept of the ***sentinel node***. Basically, what we will do is create a "dummy" node that is instantiated whenever the Linked data structure is . This dummy node would be exactly like any other node, with the special case that it will never be null. This allows us to make much fewer checks when creating methods.

Though the *addLast* method has been simplified, we can recognize that anytime we would want to add the the end or retrieve anything using the naive method (traversing the list all the way to the end) could take a lot of time, the length of the list to be precise. Well to fix this problem, we can optimize our Linked Data Structure by adding a "backwards pointer". This will point to the previous element. Thus, in our finalize approach, the first node would have a previous pointer that points to the end and the last node would have a next pointer that points to the front. This makes it so all of our methods (addFirst, addLast, getFirst, getLast, remove()) would all be very fast.

2.10 Iterators and Iterables

When we are given some sort of *collection*, that is some sort of list or set that has various items, we may want to be able go over all of them. An *iterable* is a series of elements that can be iterated over while an *iterator* is a class that manages the iteration over the iterable. To make a class an Iterable, simply implement the Iterable interface.

```
public interface Iterable<T>{
    Iterator<T> iterator();
}
```

The method *iterator* returns a class that implements the iterator interface. Frequently, iterators are implemented as private classes within iterables.

```
public interface Iterator<T>{
    boolean hasNext();
    T next();
    default void remove();
}
```

The method *remove* does not need to be implemented, to do so, you will need to reverse the iterator's previous *Next* call. The *hasNext* method checks if the iterable has another element. In this method, nothing, the state of the iterator should not change as it can have a negative consequence if you call *next* again. The *next* method returns the item of the next element in the iterable and moves the pointer to the next element.

2.11 JUnit Tests

Frequently, in the real world, you will not get autograders. In order to make sure that our programs work, we can use JUnit. Using JUnit requires us to import the package as follows

```
1 import org.junit.Test;
2 import static org.junit.Assert.*;
```

After doing this, we will need to declare certain methods as test methods, we can do that by writing *@test* above every test that we want to have some sort of test. While creating tests, it is very important that you separate your tests. Do not have one large test that checks everything at once, otherwise you will not be able to properly debug. What I suggest to do is make tests for the basic functionality of your program followed by more complicated tests that test two methods together. Oftentimes, test classes are as long or longer than classes that have your actual code.

Common tests in Junit involve the following:

```
1 assertEquals(expected, actual);
2 assertArrayEquals(expectedarray, actualarray);
3 assertTrue(value);
4 assertFalse(value);
```

2.12 Conceptual Questions

1. Identify the types of variables and methods in the following class. Note lines that any errors that may come up on.

```
 1 public class Paper {
 2     public static int thickness;
 3     public final char[] wordswritten;
 4     public static int amountofwords;
 5     private final int paperFriends;
 6
 7     public Paper() {
 8         this.thickness = 1;
 9         wordswritten = new char[10];
10         amountofwords = wordswritten.length;
11         paperFriends = 4;
12     }
13
14     public void ICanWrite() {
15         for (int i = 0; i < wordswritten.length; i++) {
16             wordswritten[i] = 'a';
17         }
18     }
19
20     public static void ICanWriteMore() {
21         char[] temp = new char[amountofwords];
22         for (int i = 0; i < amountofwords; i++) {
```

```
23              temp[i] = wordswritten[i];}
24
25          }
26      }
27
28      public void IWritetheMost(){
29          char[] temp = new char[amountofwords];
30          temp = wordswritten;
31          wordswritten = temp;
32      }
33  }
```

2. If doubles are more versatile than ints, why don't we always use them? Are there any disadvantages to doing this?

3. What will be the value of OG1.x?

```
1  public class OG {
2      int x;
3      static int y;
4      public OG(int x, int y) {
5          this.x = x;
6          this.y = y;
7      }
8      public static void main(String[] args) {
9          OG OG1 = new OG(10, 2);
10         OG OG2 = new OG(0, 1);
11         OG1.y = OG2.x;
12         OG1.x = OG2.y;
13     }
14 }
```

4. What are the differences between inheritance through classes and interfaces? Is there a particular time when you would want to use one over the other?

5. What is the difference between these two headers? Imagine pilo is some generic and T is some class

```
1  Map<pilo extends T>
2  Map<? super T>
```

6. What does the following code print?

```
try{
            char[] arr = new char[10];
            for(int i = 0; i < 2 * arr.length; i++)
                arr[i] = 'a';
        }
        catch(ArrayIndexOutOfBoundsException E){
            System.out.println("seeya");
        }
        catch(IndexOutOfBoundsException E){
            System.out.println("byebye");
        }
        catch(Exception E){
            System.out.println("outta here");
        }
        finally{
            System.out.println("GG");
        }
```

7. What does do the various declarations mean in this header? More specifically, what is the difference between the *T extends Comparable..* and *implements Comparable..*?

```
public class Tierex <T extends Comparable<T>> implements Comparable<Tierex<T>>
```

2.13 Coding Questions

1. You are given a 2d array, and you want to create a new array that has the elements ordered by column.

$$\begin{bmatrix} 1 & 2 & 3 \\ 6 & 7 & 8 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 6 & 2 & 7 & 3 & 8 \end{bmatrix}$$

```
1  public int[] TwotoOne(int[][] arr){
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16 }
```

2. Given an Intlist, you want to see if it is a palindrome. For example, 1 -¿ 2 -¿ 1 is a palindrome because if you reverse the intlist, you get the same values while 1 -¿ 2 -¿ 3 is not a palindrome. Write a program that lets you see if an Intlist is a palindrome.

```
1  public boolean isPalindrome(IntList a){
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16 }
```

3. The following question may require some basic knowledge of linear algebra, don't worry if you don't fully understand it. The determinant of a 2x2 matrix is defined as follows: $\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ab - cd$

   The determinant of a 3x3 matrix is defined as follows: $\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a * \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b * \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c * \begin{vmatrix} d & e \\ g & h \end{vmatrix}$

   For a general n x n matrix, we can define the determinant recursively. The determinant of an n x n matrix will take the following form:

   $$\begin{vmatrix} a_1b_1 & a_2b_1 & ... & a_nb_1 \\ a_1b_2 & a_2b_2 & ... & a_nb_2 \\ a_1b_3 & a_2b_3 & ... & a_nb_3 \\ ... & ... & ... & ... \\ a_1b_n & a_2b_n & ... & a_nb_n \end{vmatrix} = a_1b_1 \begin{vmatrix} a_2b_2 & ... & a_nb_2 \\ a_2b_3 & ... & a_nb_3 \\ a_2b_4 & ... & a_nb_4 \\ ... & ... & \\ a_2b_n & ... & a_nb_n \end{vmatrix} - a_2b_1 \begin{vmatrix} a_1b_2 & ... & a_nb_2 \\ a_1b_3 & ... & a_nb_3 \\ a_1b_4 & ... & a_nb_4 \\ ... & ... & ... \\ a_1b_n & ... & a_nb_n \end{vmatrix} ... \pm a_nb_1 \begin{vmatrix} a_1b_2 & ... & a_{n-1}b_2 \\ a_1b_3 & ... & a_{n-1}b_3 \\ a_1b_4 & ... & a_{n-1}b_4 \\ ... & ... & ... \\ a_1b_n & ... & a_{n-1}b_n \end{vmatrix}$$

   Or in other terms, find the sum of the following terms:
   $\sum_{i=0}^{n} -1^i * det(\text{matrix without the first row and the ith column})$.
   Write a program that computes the determinant for an nxn matrix given only the matrix.

```
1  public double determinant(double[][] arr){
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31 }
```

2.14 Chapter 2 Conceptual Question Solutions

1.

```
1  public class Paper {
2      public static int thickness;
3      public final char[] wordswritten;
4      public static int amountofwords;
5      private final int paperFriends;
6
7      public Paper() {
8          this.thickness = 1;
9          wordswritten = new char[10];
10         amountofwords = wordswritten.length;
11         paperFriends = 4;
12     }
13
14     public void ICanWrite() {
15         for (int i = 0; i < wordswritten.length; i++) {
16             wordswritten[i] = 'a';
17         }
18     }
19
20     public static void ICanWriteMore() {
21         char[] temp = new char[amountofwords];
22         for (int i = 0; i < amountofwords; i++) {
23             /*temp[i] = wordswritten[i];} Errors out because wordswritten is an instance
24             variable and you cannot access from a static context
25                  */
26         }
27     }
28
29     public void IWritetheMost(){
30         char[] temp = new char[amountofwords];
31         temp = wordswritten;
32         /* wordswritten = temp; Even though the address is the same, you cannot
33         *ever attempt to change the bits
34         * (Java is not as smart as you)*/
35     }
36 }
```

2. We don't always use doubles because of memory. Doubles use more memory than ints, and as a result, we want to avoid them when possible.

3. 0
   The key in this problem is to realize that y is a static variable. That means once OG2 is instantiated, the y's value is 1. Once we set OG1.y to OG2.x, the value of y becomes 0. Finally, we set OG1.x to OG2.y (0) and its value becomes 0.

4. When you inherit from classes, you can use methods from parent classes and interfaces you are told to implement the method from the interface (unless it is a default method). Variables in interfaces are always public static final whereas you can have instance variables in inheritance through classes. A final major difference is that you can implement multiple interfaces but you can only extend one class.

5.

```
1 Map<pilo extends T> //this one is for anything that extends T
2 Map<? super T> //this one is anything that is the superclass of T
```

6. seeya
   GG

7. The former declares a generic that will be used in the class that is Comparable, while the later makes it so the class Tierex is comparable.

2.15 Chapter 2 Coding Question Solutions:

1. The basic intuition for this problem is that you would keep track of what column index you are on and then loop over all the rows.

```
1  public static int[] twoToOne(int[][] arr){
2          int[] oned = new int[arr.length * arr[0].length];
3          int index = 0;
4          for(int i = 0; i < arr[0].length; i ++){
5              for(int j = 0; j < arr.length; j++){
6                  oned[index] = arr[j][i];
7                  index ++;
8              }
9          }
10         return oned;
11     }
```

2. In this problem it is important to remember that we are in the IntList class (we are adding a method to the class IntList). This means that we have access to the Node class as well as the next attribute for each item. We create a reversed IntList and compare it to our initial IntList. We loop over both of them and return false if, at any point, the items don't match, otherwise, at the very end we return true.

```
1  public boolean isPalindrome() {
2          IntList reversed = new IntList();
3          Node a = this.sentinel.next;
4          while (a != null) {
5              reversed.addFirst(a.item);
6              a = a.next;
7          }
8          a = this.sentinel.next;
9          Node b = reversed.sentinel.next;
10         while( a != null){
11             if(a.item != b.item){
12                 return false;
13             }
14             b = b.next;
15             a = a.next;
16         }
17         return true;
18     }
```

3. In this problem, we need to figure out how to recursively traverse our 2d array (matrix) so that we can reach our base case. We would want to add everything to the "sub matrix" except for items that are in the 1st row or the same column as the element that is multiplying the determinant of the sub matrix. The determinant works by keeping track of which element of the row we are on (even indices are added and odd elements are subtracted from the total).

```
1   public double determinant(double m[][]) {
2      if (m.length == 1) {
3          return m[0][0];
4       } else if (m.length == 2) {
5          return m[0][0] * m[1][1] - m[0][1] * m[1][0];
6        } else {
7          double det = 0.0;
8          for (int i = 0; i < m.length; i++) {
9              double[][] a = new double[m.length - 1][];
10             for (int k = 0; k < (m.length - 1); k++) {
11                 a[k] = new double[m.length - 1];
12             }
13             for (int j = 1; j < m.length; j++) {
14                 int q = 0;
15                 for (int p = 0; p < m.length; p++) {
16                     if (p == i) {
17                         continue;
18                     }
19                     a[j - 1][q] = m[j][p];
20                     q++;
21                 }
22             }
23             det += Math.pow(-1.0, i) * m[0][i] * determinant(a);
24         }
25         return det;
26     }
27  }
```

Chapter 3

Asymptotics

3.1 An Introduction to Asymptotics

Now that we've discussed programming, we are going to focus on how to make our programs run well enough to work in the real world. In order to do this, we will discuss *asymptotics*, a way of measuring either the **Time Complexity** or **Space Complexity**. The former comes more into play in the next few chapters; however, we will discuss Space Complexity in Chapter 5 and 6 briefly.

3.2 Order of Growth

The speed at which a program runs can vary from computer to computer based off various factors such as processors. In order to measure speed at a uniform scale, we use asymptotics, which give tell us how a program runs as the input scales to an arbitrarily large number. It is extremely important to realize that no matter how powerful your computer is, your asymptotic runtime will be the same (though the actual runtime may be different).

There are a few simple orders of growth. In increasing order they are:

- **log(n)**
- **n**
- **n*log(n)**
- $\boldsymbol{n^2}$
- $\boldsymbol{2^n}$
- **n!**

The higher the order of growth, the slower the program runs- as programmers, we strive for a low order of growth. The order of growth ignores the constant factor in the start of a function. This means that

100N is written as **N**.

We do this because they both scale the same in the long run. Additionally, when calculating the order of growth and two terms are being added or subtracted, we always take the highest term. The result of this is that the order of growth of

NlogN + 1000N + 99 is **NLogN**.

Let's do a basic exercise to see if we understand what is going on. Find the asymptotic running time of the following code where N is the length of the array

```
1 public static void funtimes(int[] a){
2    int j = 0;
3    for(int i = 0; i < 2*a.length ; i++){
4         j ++;
5     }
6 }
```

*The running time of this snippet of code would be **N** because we do a total of **2N steps**. Because in asymptotics we disregard constants, the running time of this snippet.*

3.3 Asymptotic Notation

We've gone over basic runtimes, but there are various notations used to describe certain needs of the algorithm.

- **Ω** - Big Omega is used to describe the lower bound of a function. For example $n^3 \in \Omega(n^2)$ because the runtime would always be less than n^3
- ***O*** - Big O is used to describe the upper bound of a function. For example $n^2 \in O(n^3)$ because the runtime would always be greater than n^2
- **Θ**- Big Theta is used to describe a tight bound of a function. This means that the upper and lower bound are the same in this scenario. For example $n^2 \in \Theta(n^2)$ because the function always runs in n^2 time

Though technically, Ω and O can be used to be anything lower or higher than the true runtime respectively, we tend to want the tightest bound possible, so we would usually want the tightest possible lowest bound or upper bound (meaning what are the lowest/highest running times that could actually occur).

Instead of saying Ω or O, we could generate a stronger statement by saying Θ in the worst case/best case. This statement is equivalent to what we said in the previous paragraph. It allows us to keep a tight bound and provides us more information than we would get from either Ω or O. The distinction is subtle but let's take this short example.

- In worst case, our code runs in $\Theta(n^2)$ time.
- Our code is bounded by $O(n^2)$

This two statements seem nearly identical, but upon a second glance, you may notice that the first statement tells us that the program can actually run in n^2 time in the worst case (it can never go above) whereas the second statement just says the program will never go above n^2. Basically, the second statement does not really tell us if the program can physically run at that speed- it will just never exceed it. It is imperative to realize that big O and Big Omega are bounds and don't necessarily tell you anything about how fast a method runs.

Though on first glance, it may seem that Ω means best case and O means worst case, this is not necessarily the situation. In the worst case, you may have a lower bound and upper bound, and a similar situation may occur in the best case. Let's take this pseudocode example

```
1 if(n is even){
2    Randomly pick a number 1 or 2;
3    if(number is 1){
4            do a function that takes n time
```

```
        }
  else{
           do a function in n^2
        }
else{
   Randomly pick a number 1 or 2;
   if(number is 1){
           do a function that takes n^3 time
        }
  else{
           do a function in 2^n
        }
```

In this case, the best case is that n is even; however, there are 2 scenarios that could occur in this situation. The lower bound of the best case is $\Omega(N)$ while the upper bound of the best case is $O(N^2)$. The worst case is n is odd. The lower bound of the worst case is $\Omega(n^3)$ and the upper bound of the worst case is $O(2^n)$. After this analysis, we can see that Ω and O are not in reference to a specific scenario for a function.

One last thing that we will discuss is *Tilde Notation*. Tilde is a special type of notation denoted with $\sim$. Tilde notation is similar to normal runtime as it looks for the highest running time; however, it takes into account the constant factor of the higher order term. Let's take this example:

$$15n^2 + 100n + 100 \sim 15n^2$$

For function to have the same running time as another function, it just needs to be of the order- $15n^2$ has the same asymptotical running time as $100n^2$. However, to have the same tilde running time, the constant on the highest order terms must be the same so $15n^2$ would only have the same running time as another function with $15n^2$ as it's highest term. Another way of thinking about this is that given two functions $f(x)$ and $g(x)$

$$f(x) \sim g(x) \text{ if } lim_{x\to\infty}\frac{f(x)}{g(x)} = 1$$

3.4 Amortized Runtime

Though we frequently want the worst case runtime, the more realistic case is that we want the average runtime, this is called *amortized runtime*. Amortized runtime tends to choose a worst case input and then calculating how long a series of operations would take on an input of that size For example, say we want to calculate the amortized runtime when adding to an array, we can do it as follows:

$$\frac{(1+1+1...)+N}{N} \to \frac{N+N}{N} \to \frac{2N}{N} \to 2 \text{ amortized runtime is } \Theta(1) \quad \text{N is the size of the array}$$

The above equation comes from the fact that N adding operations done in constant time and there is 1 call to resize the array which takes N time. We divide this by the total amount of operations which is N -1 and as a result we get $\Theta(1)$ as the amortized runtime.

It is imperative that amortized runtime not be confused with average case runtime. Average-case implies that the input is "average" whereas amortized runtime refers to the runtime of a worst case input in the long run. Amortized runtime will become increasingly important when we start to discuss data structures.

3.5 Calculating Basic Runtimes

We have gone over what runtime is and the basic notations for it; now we are going to learn how we can calculate the runtimes of some basic functions. Before we get started with actually calculating the runtimes, we must define some formulas that will be helpful in the future.

$$1+2+3+4+5+....+n=\frac{n(n+1)}{2}\rightarrow N^2$$

$$1+2+4+8+16+...+N=N$$

When attempting to figure out runtimes, it is important that we realize what we are trying to find the runtime in terms of. When there are multiple variables in play, we want to be explicit on what the running time is scaling to. Usually, we scale in reference to an input array (or other data structure) size, some number, or the length of some string. Now that we know some basic formulas, let's attempt to figure out some runtimes. Let's look at the following function:

```
1 public void hello(int n){
2     for(int i = n; i > 0 ; i = i/2){
3          System.out.println("Hello?"}
4 }
```

The running time for the above method would be $log(n)$. This is because the problem is divided by 2 each time. The i in the for loop will start off as $n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4}... \rightarrow 1$ leading to $log_2(n)$ amount of iterations which simplifies to $\Theta(log(n))$

Let's look at another problem, this time with while loops:

```
1 public void ptTwo(int[] arr){
2      int i = 0;
3      while(i < arr.length){
4          i++;
5         }
6  }
```

The running time for the above method is n. This is because you are doing 1 iteration for each element of the array. Since there are N elements, you do N iterations.

3.6 Complicated Runtimes

Not all runtimes are straightforward. Recursive steps and nested for loops are the prevalent manners in which runtimes can become more tedious.
For most problems, where the runtime relies on a different factor, we can use the following summation.

$$\sum_{i=0}^{\#oflayers} \frac{work}{node} * \frac{\#ofnodes}{layer}$$

Or in English, the sum of the work per node times the number of nodes per layer. Let's step through this recursive function as practice.

```
1 public void stepper(int n){
2       int q = 0;
3       for(int i = 0; i < n; i++){
4            q++;
5       }
6       stepper(n/2);
7 }
```

Let's use the sum formula that we established earlier. It seems that the recursive call makes n $\frac{1}{2}$ of what it was; however, the recursive call does not increase the amount of nodes. Because of this, the number of

nodes per layer is 1. The for loop that occurs at each iteration does work proportional to $\frac{n}{2^i}$ where is the current layer. Thus the formula for the sum is:

$$\sum_{i=0}^{log(n)} \frac{n}{2^i} * 1$$

. We can then rewrite this as $n + \frac{n}{2} + \frac{n}{4} + + ... \frac{n}{2^{log(n)}} = \Theta(2n) = \Theta(n)$

```
1 public void(int n){
2          int q = 0;
3      for(int i = 0; i < n; i++){
4           for(int j =0; j < i; j++){
5               q++;
6            }
7      }
8 }
```

The running time for this method would be n^2 this is because the outer loop works a total of n times and the inner loop works in proportion to what iteration the outer loop is on. This results in the inner loop working 1 time on the first iteration, 2 times on the second etc. This make the sum 1+2+3+4+... $n = \Theta(n^2)$

Right now you're probably wondering "how can I solve these, is there a universal method?". Well, I'm sorry to be the bearer of bad news, but really, the only way to get good at asymptotics is to practice over and over again. You will begin to recognize patterns in the way asymptotics are formed and eventually you'll be able to master them. In the next chapter, we will discuss runtimes and their relation to data structures.

3.7 Revision of Strategies for Solving Asymptotics

Now that we have gone over strategies for solving asymptotics informally, let's list when to use which method. Using the wrong method at the wrong time may result in the wrong answer or a misunderstanding.

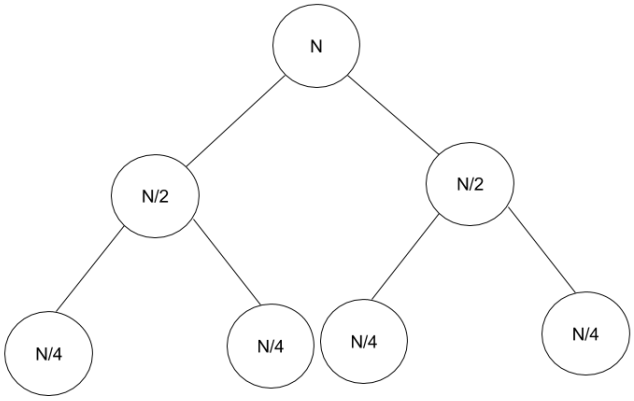
If the code is relatively straightforward, such as a basic while loop, with no recursion or loops that rely on it, you can simply just "logic" it out and count. We have an example here:

```
1 int i = 0;
2 while(int i < N){
3         i++;
4 }
```

The asymptotic runtime of this function is trivially $\Theta(N)$, no complex math is needed, just logic. These asymptotics are nice and easy; however, these are not too common.

One strategy that helps those who are more visual is the "recursion tree". For each iteration, create one more layer. Once you get a good image of it in your mind, you can stop drawing it. Let's draw a recursion tree for the following code.

```
1 public void  grewt(int N){
2      for(int i = 0; i < N ; i++){
3           System.out.println("I am grewt");
4      }
5      grewt(N/2); grewt(N/2);
6 }
```



We could have gone done more layers, but for the sake of space, let's just do 3 layers. Now we can see from the following tree that we made, that each layer we divide by 2, and thus our work divides by two. This leads to the work per node being $\frac{N}{2^i}$ where i is the current layer. Now, let's look at the nodes per layer- we see that each node has 2 children because of the 2 recursive calls. So the first layer has 1 node, the second layer has 2 nodes, the third has 4 and so forth. The pattern that we can see is that there are 2^i nodes per layer where is the current layer. Now let's try to plug all the information that we gathered into our equation:

$$\sum_{i=0}^{log(n)} \frac{n}{2^i} * 2^i \rightarrow \sum_{i=0}^{log(n)} n \rightarrow \Theta(nlog(n))$$

If you go step by step, these problems become much easier to solve

3.8 Some Closing Words on Asymptotics

When thinking about "N" and best case/worst case, make sure you do not think of a specific number. For example, if the code resembles the following:

```
1 if(n ==1){
2    return;
3  }
```

you CANNOT assume that the input will just be 1 and call it a day. What you can do, when thinking about best case is a quality of the number such as:

```
1 if(n %2 == 0){
2     return;
```

This is because you are now looking at a quality of the input and assuming something about it, not the actual numeric quantity.

Additionally, when writing out your summations make sure you simplify as much as possible, (cancel out all terms that repeat) otherwise you will end up with weird numbers. Don't get demoralized if you find asymptotics hard, it is difficult to internalize the rules and situations; however, with enough practice (trust me I'll put enough practice on this document), you will be able to solve them.

3.9 Conceptual Asymptotic Problems

1. : List the following runtimes in increasing order:
   $\Theta(N!)$, $\Theta(NlogN)$, $\Theta(N^2)$, $\Theta(\frac{n^3}{log(N)})$, $\Theta(2)$

2. Each line will have 2 runtimes provided, in between them, write which runtime is ALWAYS asymptotically greater, less than, equal, or not enough information to tell. Provide justification for your choice.

   (a) $\Theta(N^2)$ $\Theta(N)$

   (b) $\Theta(N^2 + 5n)$ $\Theta(N^2 + 3N)$

   (c) $O(N!)$ $\Omega(1)$

   (d) $O(1)$ $\Omega(N)$

   (e) $\Theta(\frac{n^2}{n^3})$ $\Theta(1)$

   (f) $\Omega(N)$ $\Omega(N)$

   (g) $\Theta(1)$ $O(1)$

   (h) $O(N)$ $O(N)$

3.10 Find The Runtime Questions

Find the runtime for the following methods. Use Theta when possible, if not use Omega and Big O. Make your bounds as tight as possible.

```
1   public int recursFib(int N) {
2          if (N <= 1) {
3              return n;
4          }
5          return recursFib(N - 1) + recursFib(N - 1);
6      }
```

1.

```
1  static int yumGrub(int[] grubbart) {
2          int N = grubbart.length;
3          int chug = 0;
4          for (int i = 0; i < N; i++) {
5              for (int j = i + 1; j < N; j += 1) {
6                  if (grubbart[j] == grubbart[i]) {
7                      chug++;
8                      break;
9                  }
10             }
11         }
12         return chug;
13     }
```

2.

```
1  public void lion(int timon, int pumba) {
2          for (int zazu = 0; zazu < timon; zazu++) {
3              int mufasa = zazu * zazu;
4              while (mufasa <= pumba) {
5                  System.out.print("Hakuna Matata");
6                  mufasa += 1;
7              }
8          }
9      }
```

3.11 Chapter 3 Conceptual Question Solutions

Conceptual Problems

1. The ordering would be $\Theta(2), \Theta(NlogN), \Theta(N^2), \Theta\frac{N^3}{log(N)}, \Theta(N!)$

   (a) $\mathbf{\Theta(N^2) > \Theta(N)}$ The function on the right is tightly bounded by something of higher order than the function on the left.

   (b) $\mathbf{\Theta(N^2 + 5n) = \Theta(N^2 + 3N)}$ We ignore lower order terms in asymptotic running time so these two are equal.

   (c) $\mathbf{O(N!)}$ **?** $\mathbf{\Omega(1)}$ We are provided an upper bound for one function and a lower bound for the other; however, we are not sure how either of the functions truly run- what their average case is/worst and best case running times. So no conclusion can be made

   (d) . $\mathbf{O(1)} <\mathbf{\Omega(N)}$ We know that the function on the left is upper bounded by 1. This means that in worst case, it runs no worse than constant time. The function on the right is lower bounded by n- meaning in best case, it is no better than linear time. It is trivial to see the relationship from here

   (e) $\mathbf{\Theta(\frac{N^2}{N^3})}$ **either =** $\mathbf{or < \Theta(1)}$ Theoretically, the answer should be that the running time on the right should be greater. After all, in the long run, $\frac{1}{N}$ (what the left simplifies to) is less than 1. However, a $\frac{1}{N}$ running time is functionally impossible so both answers were accepted.

   (f) f. $\mathbf{\Omega(N)}$ **?** $\mathbf{\Omega(N)}$ It is impossible to tell the running time for this function, as we are just given the lower bound. We need more information.

   (g) $\mathbf{\Theta(1)}$**either =** $\mathbf{or ?\ O(1)}$ This one is a bit tricky. We know that the function of the left ALWAYS runs in constant time and the function on the right has an upper bound of constant time (meaning it could be less than). However, there is no real function than runs in less time than constant. The only possible way to get such a method is to have a method with 0 lines of code as its body. So theoretically, it is possible for the function corresponding to the runtime on the right to run in time less than what was provided. Practically, there is no runtime under constant.

   (h) $\mathbf{O(N)}$ **?** $\mathbf{O(N)}$ It is impossible to tell what the relation between these two runtimes is. They both have an upper bound of n, however one could run in $\sqrt{(N)}$ time and the other could run in $log(N)$ time. We need more information

3.12 Find the Runtime Question Solutions

1. Answer: $\Theta(2^N)$
   We start off with an int n. At each layer, the work done is constant (no work is done). However, each function call returns 2 more function calls. This makes the nodes per layer equal to $\mathbf{2^i}$. We know that that there are a total of N steps because N decreases by 1 each step. Plugging this into our summation, we get
   $$\sum_{i=0}^{N} 1 * 2^i \rightarrow \Theta(2^N))$$

2. Answer: $\Omega(N), O(N^2)$
   We see that N is equal to the length of grubbart. The first for loop runs a total of N times. The second for loop runs in relation to the first for loop. It is important to realize that there is a best and worst case in this code- item j and i in grubbart could be equal as the first element (best case) or they could never be equal (worst case). In the best case, the 2nd for loop is constant time each time- only one iteration, so the function would run in $\Theta(N)$. In the worst case, the the loop, in total, would run $N-1+N-2+N-3+...+1$ or it could be rewritten as $1+2+3+...+N-1$ which simplifies to $\Theta(N^2)$

3. Answer($O(timon + pumba\sqrt{pumba})$
   The first loop works a total of *timon* times. This occurs regardless of what else is happening, and it can never terminate early. The inner loop runs only when the i value would be less than $\sqrt{pumba}$. For each time this occurs, the work done by this loop would be upper bounded by n.

Chapter 4

Data Structures

4.1 An Introduction to Data Structures

Data Structures are in the most simple to understand definition, a method of organizing data. Basic Data Structures that we have seen already include arrays and the various "Deque"'s[1] and IntLists. These data structures are perfectly fine; however, we may want to store our data in a different manner that would be more optimized. In this chapter, we will go through the data structures, their runtimes, and pros and cons of using certain data structures.

4.2 Introductory Data Structures

Before we discuss more complicated data structures, we will discuss more basic ones, specifically ***LinkedLists***, ***Arrays***, ***Stacks***,and ***Queues***.

Linkedlists were one of the first data structures we went over, in the form of IntLists. A LinkedList is basically a set of nodes, each with a next and previous pointer. With all the full optimizations, a Linkedlist takes $O(N)$ time to find an item, $O(N)$ to insert, $O(1)$ time to add and $O(N)$ to delete an item. Linkedlists, along with arrays, are the data structures that most other data structures are derived from.

Arrays are "containers" that can hold a fixed number of objects- you establish how many items it can hold when instantiating it. Arrays have a useful counterpart, ArrayLists, which are arrays that resize themselves automatically. An arraylist takes $O(1)$ time to find an item (given an index), $O(N)$ to find a specific item (not given the index), $O(N)$ to insert into an array (because of resizing), and $O(N)$ to delete because you would have to shift over elements (given the index) and find the element (not given the index).

Stacks are First in Last Out Data Structures (FILO). This means that if you insert item A into a stack followed by item B and remove all the items, B will come out before A. The *push* operation of a stack puts an item on the top, the *peek* operations returns the most recently put item, and the *pop* operation removes the first item. All of these operations take $O(1)$ time.

The final basic data structure is a Queue. A queue is a First in First Out Data Structure (FIFO). This means that if you insert item A in a queue followed by item B and remove all the items, item A will come out before item A.

[1]The Deque's are from UC Berkeley's Computer Science 61B course taught under Josh Hug, it was not discussed explicitly in this material, the basic concept is similar to that of a LinkedList (which we will reference soon).

4.3 The Disjoint Sets:Quick Union and Quick Find

One problem that comes up frequently in Computer Science, and the world in general, is "are two things connected?". This problem is found in social networks (think mutual friends), power grids, and more. We will refer to this problem as the "Disjoint Set Problem". The API for this problem would be

```
1 void union(int p, int q); \\connects two ints p and q.
2 int find(int p); \\ the parent/value associated with a certain index(p).
3 boolean connected(int p, int q); \\are two int's,p and q,connected.
4 int count(); \\amount of values.
```

The easiest way to approach this problem would to use some sort of array, we'll use *arr* as our array variable. Where the index of the array is the item we are looking at and the value is the "set" that it is in. In the naive implementation, when checking *connected(a,b)* we just check if arr[a] == arr[b]. To make sure that this is always the case, whenever we call *union(a,b)*, we will check if the two values are the same. If the values are the same, nothing would need to be done; however, if they are different, that means we will need to change all the items in *arr* that have the value of a to the value of b. Let's take the following example, the first image is of the initial array, the second is one where the call *union(2, 4)* has occurred

parent	1	2	2	4	4
index	1	2	3	4	5

Union(2,4)

parent	1	4	4	4	4
index	1	2	3	4	5

Analyzing this code, we can recognize that *union* would take N time in worst case because you would have to go through $O(N)$ elements in the array to connect the two. Find would take $O(1)$ because you would simply return the element associated with it. This data structure is essentially a linkedlist in terms of connectivity. This is decent, but we can do a lot better. Because *find()* takes such little time, we will call this data structure ***Quick Find***

So let's say we wanted to speed up the *union* method- we would use a similar data structure called a ***Quick Union***. The underlying data structure would also be an array, and the value of each index would be its parent; however, our implementations of *find*, *union*, and *connected* would be quite different. Before, *find* would immediately return the value of the parent; however in the Quick Union Implementation, find would start at one item and go to its parent- this process would keep repeating up until the the parent of the node is itself, in other words, it is the root. To implement *union*, we would use *find* to find the root of the items, then all we would do is change the value of the root to be the value of the other root. Below is the basic java code for *find* and *union*.

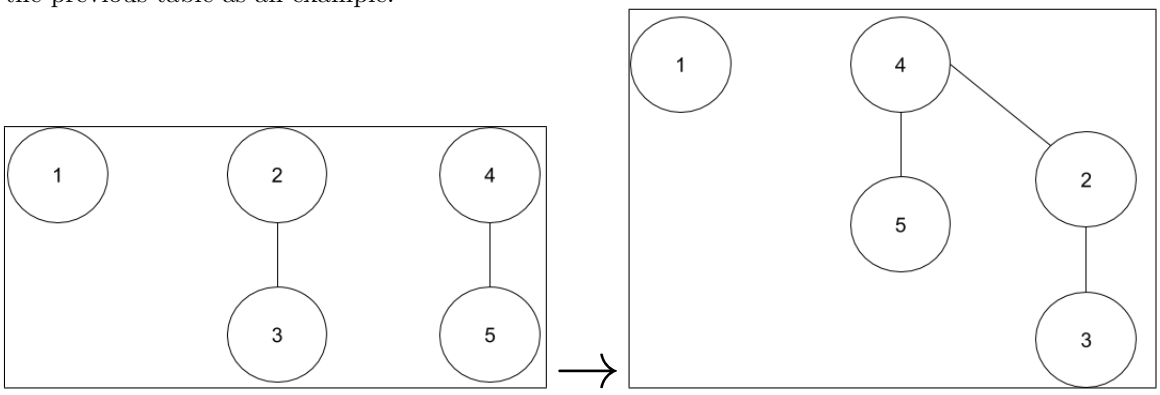
parent	1	2	2	4	4
index	1	2	3	4	5

Union(3,5)

parent	1	4	2	4	4
index	1	2	3	4	5

```
1  private int find(int p)
2    {
3           while (p != id[p]){
4               p = id[p];
5           }
6       return p;
7  }
8  public void union(int p, int q){
9       int pRoot = find(p);
10      int qRoot = find(q);
11      if (pRoot == qRoot){
12           return;}
13      id[pRoot] = qRoot;
14      count--;
15
16 public boolean connected(int p, int q) {
17      return find(p) == find(q);
18     }
19 }
```

Note how only one of the values changed. Intuitively, this makes us feel that *Union* would be faster, but exactly how much faster? Well let's look at a diagram of how exactly QuickUnions are formed. Let's use the previous table as an example.



We can see that this forms a tree-like structure. This means that in worst case, the runtime for *union* and *find* is the $\Theta(N)$; however, the average runtime for both these functions is $\Theta(log(n)$, because that tends to be the height of a tree. Because the connections are arbitrary, the height can, at times, make the structure essentially a linked list. We'll tackle how to solve this problem in the next section.

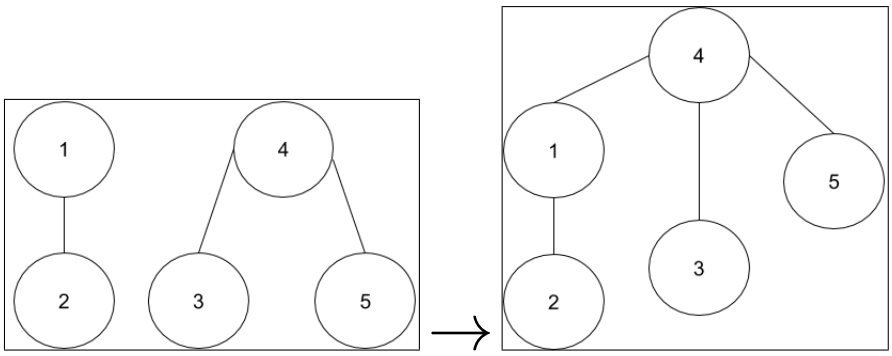
4.4 Disjoint Set Improvement: Weighted Quick Union

To solve the prior problem of having ridiculously large tree heights, we will implement a data structure called the ***Weighted Quick Union***. A Weighted Quick Union follows the same pattern as the Quick Union; however, we keep track of the size of the two trees being connected- we can do this with a separate "size" array or store it inside the node. The root of the smaller tree is then connected to the root of the larger tree- becoming its child- this ensures that the height of the tree will be no bigger than $log_2(n)$ or $lg(n)$. This makes it so *union* takes $lg(n)$ time in the worst case, a substantial improvement over N.

parent	1	1	4	4	4
index	1	2	3	4	5

Union(2,5)

parent	1	4	4	4	4
index	1	2	3	4	5

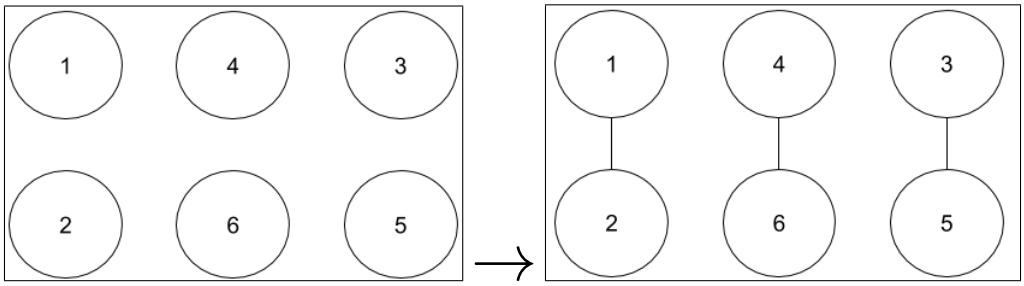


Though the Weighted Quick Union is good, we can make it even better by using a strategy called path compression. This strategy will provide us with a very fast *find* and *union* take nearly constant time. To implement this strategy, inside our *find* method, every node that you find on the way will connected directly to its current root. Unlike Quick Find, where we connect everything in *union*, we connect everything inside the *find* method- this means that we will not go out of our way in order to connect things to the root. We will go through a full example of constructing a Weighted Quick Union with Path Compression.

parent	1	2	3	4	5	6
index	1	2	3	4	5	6

Union(1,2), Union(4,6), Union(3,5)

parent	1	1	3	4	3	4
index	1	2	3	4	5	6



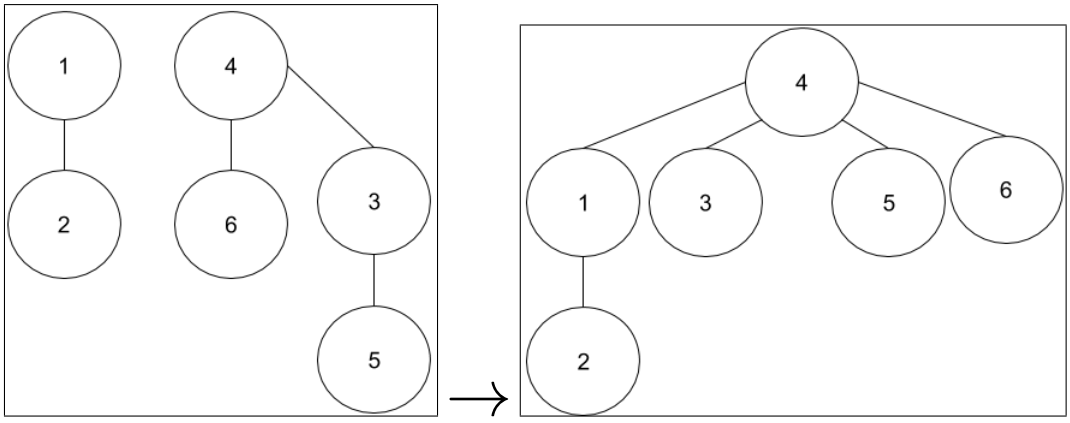
We connected all the nodes. 2's parent became 1, 6's parent became 4, and 5's parent became 3. The node that we chose to be the root was arbitrary in this case.

Union(6,5)

parent	1	1	4	4	3	4
index	1	2	3	4	5	6

Union(2,5)

parent	4	1	4	4	4	4
index	1	2	3	4	5	6

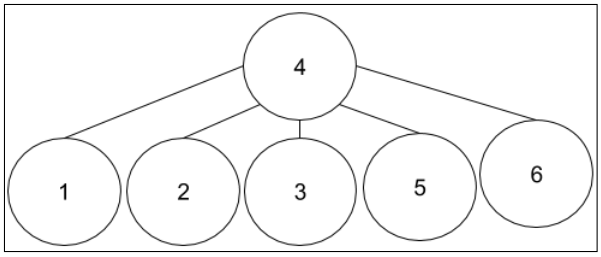


In this case, we first unioned 6 and 5. To do this, we would find the root of 6 and the root of 5 and then connect them. This would be 4 and 3 respectively. The decision for which one is the root is once again arbitrary since both trees are of the same size. Up until now, this is just a normal Weighted Quick Union.

Next we perform a union operation for 2 and 5. We once again find the root of 2 and 5 and connect them. The root of 2 is just 1, its direct parent, so there nothing that changes. However, for 5, the root is not its direct parent, so we make its parent the parent of its parent. In this case, that node is the root; however, in larger trees, all the nodes on the way to the root would have their parent change to be the parent of the parent over and over until the root is reached. Since the tree rooted at 4 is larger than the tree rooted at 1, we make 4 the root of the overall tree.

Find(6,2)

parent	4	4	4	4	4	4
index	1	2	3	4	5	4



For the final operation, we are performing a find operation. We want to see if 6 and 2 are connected. To do this, we will find the roots of each of them. Since we are going up the tree, it is only natural to change the parent of the nodes along the way, if their parent is not the root. So as a result

of this connected operation, we change the parent of 2 to be 4 from 1 because its direct parent is not the root.

Algorithm	Union Runtime	Find Runtime
Quick Find	N	N
Quick Union	Tree Height	Tree Height
Weighted Quick Union	lgN	lgN
Weighted Quick Union with Path Compression	Almost constant time Amortized	Almost constant time Amortized

We now can see the drastic difference between our data structures and the improvement that we got by writing a few lines of code.

4.5 Trees and Binary Search Trees

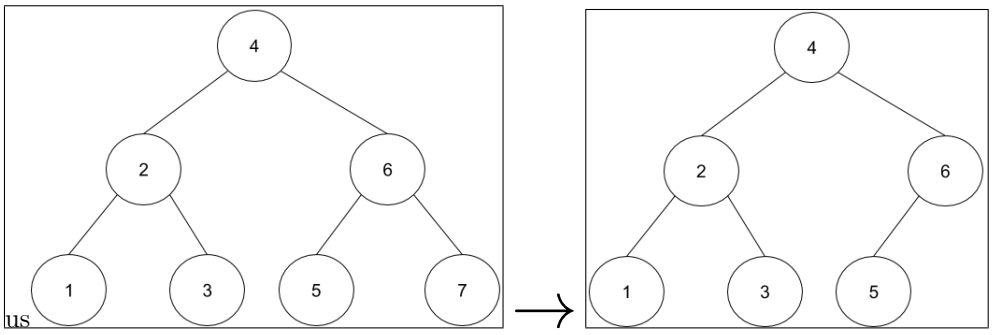
Trees, in the most basic sense, are a set of nodes connected to each other by a set of edges. Nodes can have 0 children, in which case they are called a *leaf*, or more children. A property of a tree is that every child of a tree is also a tree. The only constraint is that, from any one node, there is only one path to any other node. A node that is the child of no other node is called the *root*

A specific type of tree that is useful for ordering data is the ***Binary Search Tree***. The Binary Search Tree, or BST for short, is a way of organizing *comparable* nodes. Nodes in BST's have between $0-2$ children. Children to the left of a node will are always "less than" the root and items to the right of a node are always "greater than" the root, we will not really worry about handling duplicates at this point of time. The API for a basic Binary Search Tree is as follows.

```
1 public BST get(BST T, Key k); //gets the node that has the given key k
2 public BST insert (BST T, Key k;//inserts a key k, if it does not exist already.
```

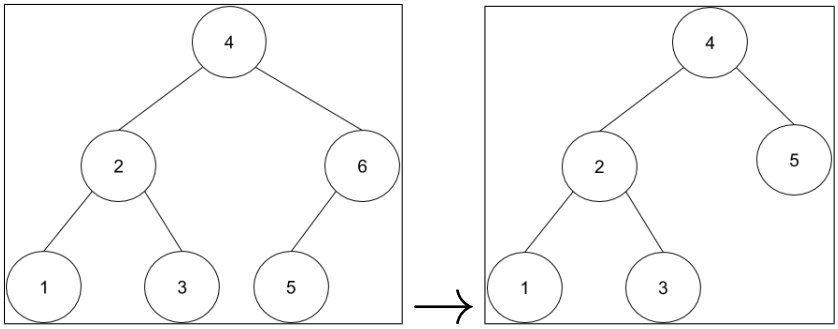
Inserting and getting from a BST are very similar processes. In both cases, you start at the root of the tree, if your key is less than the current node, you recursively go to the left node, if your key is greater than the current you go to the right and if your key is equal, you return your current node. Inserting and getting from a BST would both operations, $\Theta(log(n))$, as that tends to be the height of the tree.

The process for adding to a Binary Search Tree is relatively trivial; however, the process for removing a node can be more complicated. In the case that a node has no children, all you do is remove the parent's link to it, then Java garbage collects it. In the case that a node has 1 child, you just make the parent of that node point have its pointer (whichever was pointing at the initial node) and have it point to the child. This works because the child of the initial node will always have the same relation to the parent that the initial node did (either less than or greater than). Below are examples of removing nodes. The first example is removing a node with no children (7).

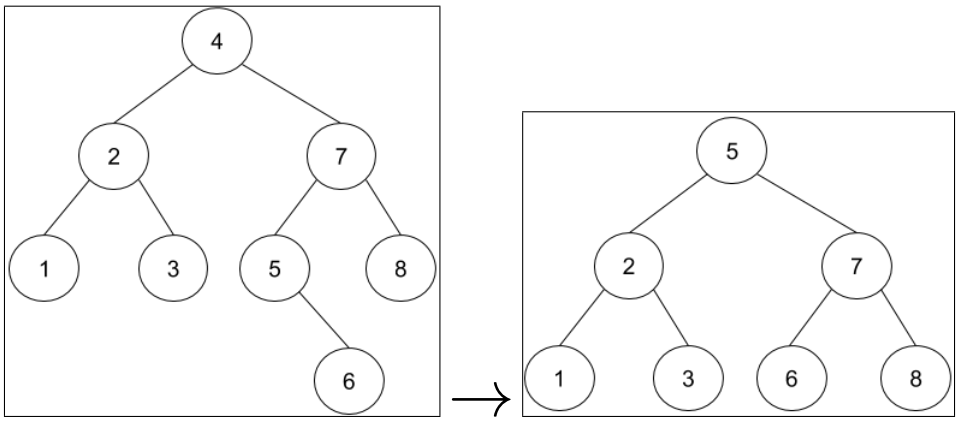


us

Below is an example of how to remove a node with only one child, in this case 6.



When a node has 2 children, things get a bit trickier. You cannot always pick the immediate child otherwise the BST property may not always be maintained. Look at the following example.
To solve this problem, we can replace the node that is being deleted with either the node that is greatest on its left side or least on its right side. The node chosen will always have 1 child maximum, as if it had two children, it is the case that, on the right side, something would be less than it, and on the left side, something would be greater than it. Once we replace the node, we do the process for the deletion of a node with 1 child. That was a lot of words, so let's show an example, in it, we will take the smallest item on the right side of the node 4.

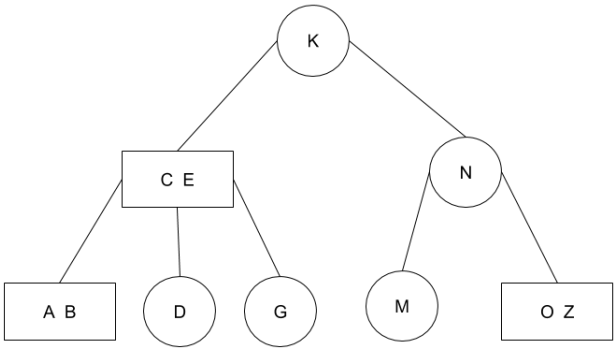


As we stated, runtimes of Binary Search Trees can vary based off the height of the tree. The height of the tree is affected by the order in which keys are inserted. For example, if the first item in the tree is "2" and the second item is "1", it will look different than a tree than a tree with "1" inserted first and then "2". This means that the order in which keys are inserted can actually affect the runtime. If the keys are inserted in a good way, we will get a "bushy" tree, one that looks like a shrub and if the items are inserted in a poor way, we may get a "spindly" tree, or one that looks like a linked list.

We state the runtime of a Binary Search Tree operation like insert to be $\Omega(lg(n))$ in the worst case. This is because each item in a bushy tree will have 2 children. This means that at each layer, we divide the problem in $\frac{1}{2}$. As we go through the layers, the problem's size becomes $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}$.... If we were to, for some reason, increase the number of children each node could have to 3 for example, the runtime for insert would be, in best case, $\Theta(log_3(N)$. We don't really care much about the base change for logarithms, but this is important so you understand how we come across these runtimes.

4.6 Balanced Search Trees: 2-3 Trees

In order to ensure that we don't get the worst case runtime for a Binary Search Tree, we can attempt to use a category of data structures called ***Balanced Search Trees***- these data structures will always have a logarithmic height regardless of how keys are inserted. The first such data structure we will discuss is called **2-3 Trees**. The difference between this and a standard Binary Search Tree is that we can have "2 nodes" which have 2 elements inside of them and have either 0 or 3 children- "1 nodes" will have either 0 or 2 children. To provide a visual representation, this is how a 2-3 tree looks.

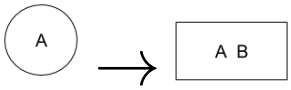


Now that we know what a 2-3 Tree is, how do we go about searching through one? Well, just like the Binary Search Tree, we start at the root; however, instead of immediately traversing the tree, we check to see if the node is a "2 node". If it is a "2 node", we check to see if the element we are searching for is less than the smallest element in the node, greater than the largest, or in between. If the element is less than the smaller element in the node, we traverse the left branch, and if the item is greater than the larger element we traverse the right branch- this process is similar to a Binary Search Tree search. However, if it is the case that the element is between the two elements, we go down the middle path. We recursively do this process until we find the element, or until there is no element to go to (the element is not in the tree).

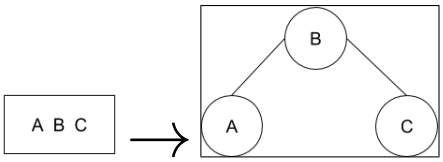
Now that we know how to do search on a 2-3 Tree, let's see if we can do insert. In order to insert an element into a 2-3 tree, we have to recognize a few things: A node can have no more than 2 elements within it and the number of children a node has depends on how many elements are within it. There are a few cases that we need to account for.

- Inserting into a node with 1 element.
- Inserting into a node with 2 elements.
- Inserting into the root.

Let's begin to create a 2-3 tree. Following each image will be a brief description of what happened.

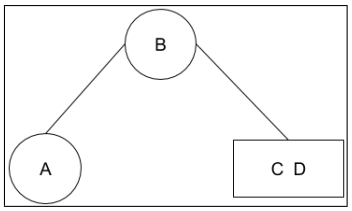


Initially, we created a 2-3 Tree composed just of A. In the next step, we added B. We then made it into a 2-node.

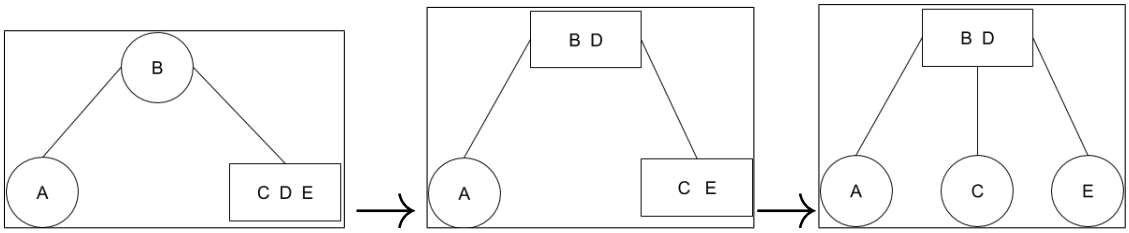


In the first step of this process, we inserted C. However, unlike the previous step, when we insert into this node, we have 3 elements in 1 node. By our definition of 2-3 Trees, we can only have 2 elements max per node. Since there is no parent to move an element to, we will make the middle element the root (in this

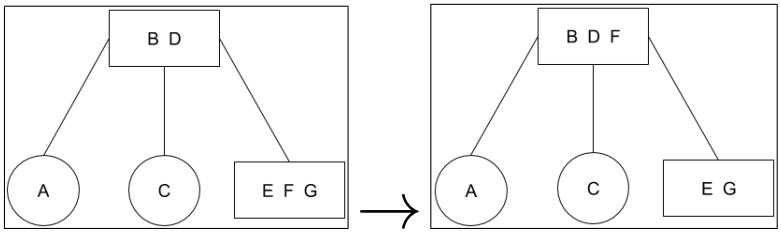
case B) and make A its left child and C its right child.



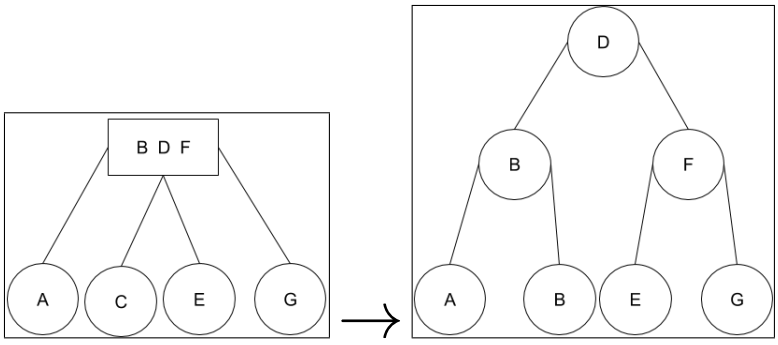
This step is trivially adding D, since it is greater than D, it goes to the right side. Since C is in a normal node, we transform it into a 2-node and add D.



In this step, we added E, once again we end up at the node with C and D. We add E to this node; however, it becomes a node of 3. We solve this by "promoting" the middle child- D. We move D to the root and make the root a 2-node. The new problem that arises is that the node BD only has 2 children. To solve this, we split the 2-node that we promoted D from. We know that both elements in the 2-node that D was promoted from are greater than B as they were to the right of B. We also know that there is one element less than D and one greater than it because D was the middle element of the node. As a result, from this node, we can make one node that is between B and D and one node that is greater than D.



In the initial step, we added F and G (adding F was trivial so I skipped that step). Once again, we end up with a 3-node. To deal with this, we promote F. However, once we promote F we have a 3-node in the root. We'll solve this in the next step.



To solve the problem of getting a 3 node in the root, we first split the 2-node that F was promoted to into 2 separate nodes. Now we have 1 node less than B, 1 between B and D, 1 between D and F, and one greater than F. We finish off by making D the root and the item less than it in the root (B) its left child,

and the item greater than it in the root (F) its right child and make the corresponding old children of the 3-node their children.

Just about now, you are probably wondering "is there some sort of universal way to insert into a 2-3 tree? Here are a set of "steps" that can be followed in order to insert into a 2-3 Tree.

- No matter what element you are inserting, the item you insert will first go to a leaf.
- If the node that you insert into is a single node, you make it into a 2-node and you are done.
- If the node becomes a 3-node, you promote the middle element.
- You split the children so that the parent has $1 + amountofitemsinthenode$.
- If the node becomes a 2-node after the promotion, you are done, otherwise you promote the middle element again.
- This process keeps repeating unless the node is the root.
- If the root becomes a 3-node, then the middle child becomes the new root, and the item less than it in the node becomes the left child and the item greater becomes the right child.

All of a 2-3 trees operations will run in $\Theta(log(N))$ time in worst case. The reason behind this is that the height of the tree will never exceed $log(N)$.

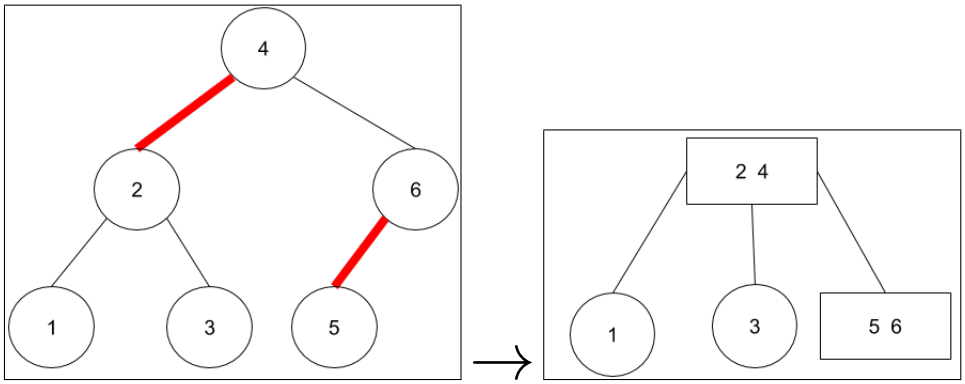
4.7 Balanced Search Trees: Red Black Trees

There is also another type of Balanced Search Tree called the ***Red Black Tree***. Red Black Trees are isometric with 2-3 trees, that is there is a red black tree which can be represented as a 2-3 tree. Similar to a 2-3 tree, Red Black trees have a guarantee of $lg(n)$ height, so what is the benefit of it over a 2-3 tree? Well in practice, Red-Black Trees are easier to implement, especially the that we will go over, Left-Leaning Red Black Trees (LLRB). With 2-3 trees, you would need various classes for different types of nodes and would constantly need to modify a node based off insertions and deletions. With LLRB's, you have a standard class that can be used regardless of any conditions. Let's now discuss the structure of red black trees.

Instead of having "nodes" that have more than 1 element, Left Leaning Red Black Trees have 2 distinct kinds of edges. A black edge between two nodes indicates a parent child relationship while a red link shows a "same node" relationship. For a Left Leaning Red Black Tree to be valid, it must follow certain criteria:

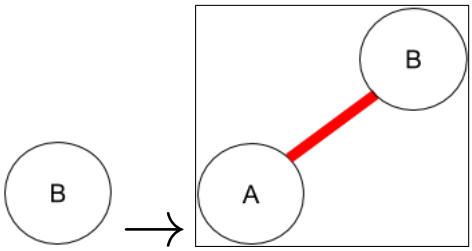
- Red Links can only lean left
- No Node has 2 red links connected to it
- Every leaf in the tree has the same amount of blank links above it.

Here is an example of an LLRB, and it's equivalent 2-3 Tree:

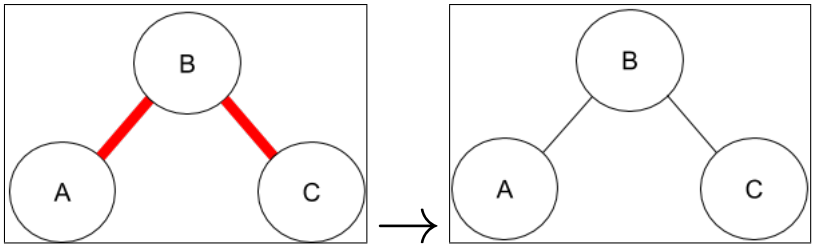


Search is relatively self explanatory in the LLRB, you just look to see if a node is less than the current, if so go left otherwise go right- you continue this until you find the element. Inserting; however, can be a bit tricky.

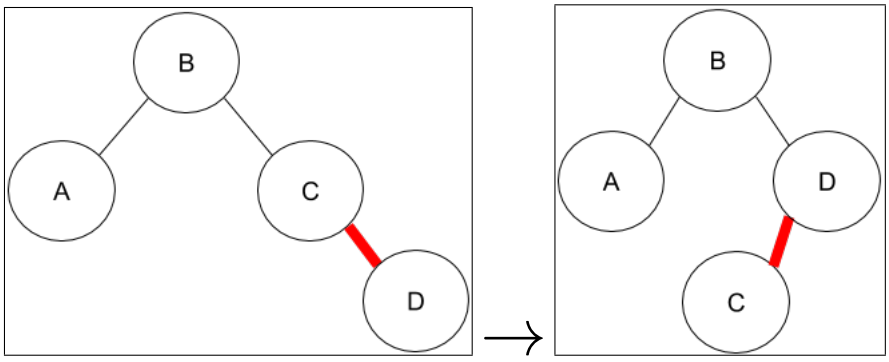
There are 2 broad ways that we can insert into a LLRB- either we can convert the tree into a 2-3 tree, the usual preferred approach as it is simple, or we can do a series of color flips and rotations. The 2-3 tree method is straightforward, so we won't discuss how to do it; however, we will go over the way to insert into an LLRB. Whenever inserting into a LLRB, the node that you insert is inserted with a red link. After the element is inserted, we must make sure that it follows our above rules. We will follow a similar process of creating a red-black tree from scratch in order to show all the possible cases.



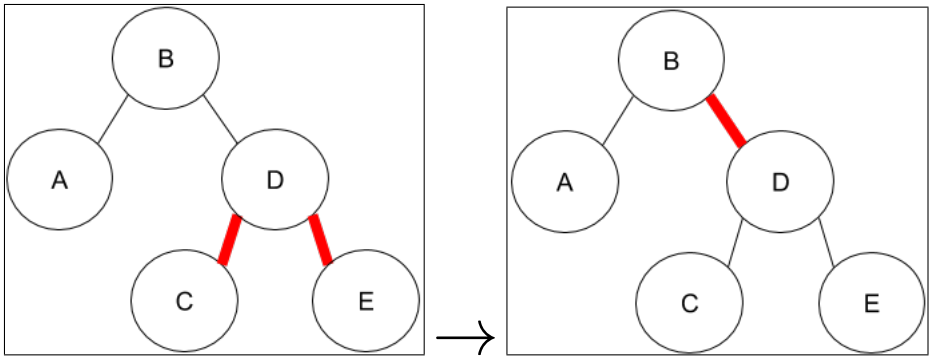
In the above image, we inserted the node B, following this, we inserted A. Whenever we insert into a red black tree, we insert with a red link. Since A is less than B, we make it the left child of B.



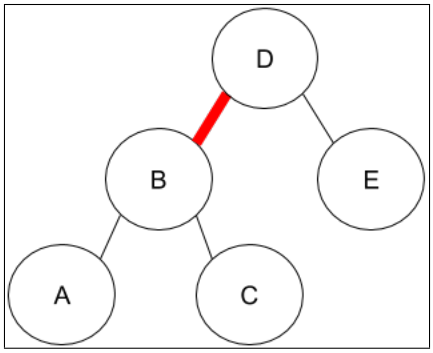
In this example we inserted C, and since it is greater than B, it is inserted as a right child of B. Now the problem is that both of B's children have red links, to circumvent this problem, we "flip the colors" so that only the parent B's parent link is red, and its children links are black. Since B has no parent links, the only color flips are from the links to A and C from B.



In this image, we inserted D; however, now we have a red link "leaning right". Since this is never allowed, we sim-
ply rotate the node D to the left. The link between C and D stays as red, and it is now a valid Red Black Tree.



In this step, we inserted E into our LLRB. We now have a similar situation to the step when we inserted C. We do a similar process of color flipping and D's parent link becomes red. Keep in mind that this is a valid trick because the amount of black links will remain the same (If the two children have red links making both links black and turning the parent's red keeps each leaf's amount of black links the same since 2 links are changing per leaf). However, after this we realize that we created a new problem, there is a right leaning link from B to D, we'll fix it now!



In order to fix this problem, we simply rotated the D node to the left. Keep in mind when we do this, that the children less than D, in this case just the node C, must become the right child of the old root. If we want to logic this out it is because C was less than D but greater than B.

Here is a compiled "cheat sheet" of rules for the rotation of Left Leaning Red Black Trees and they are as follows:

- A node is ALWAYS inserted with a red link.
- When you insert an item, and it causes a problem, take care of the small subproblem, which will only contain the inserted node's parent and potential other child.
- If the item was inserted to the left of the parent, there is no problem.
- If the item was inserted to the right of the parent and the parent does not have a second child, perform a rotation so the link becomes left leaning. Remember to move any children properly to make sure the rotation is valid.
- If the item was inserted to the right of the parent and the parent has a second child, perform a color flip.

The steps for inserting into a red-black tree are much fewer than those for inserting into a 2-3 tree which is partially why the implementation is easier.

We can see from the insertion steps that we did for both 2-3 trees and Red-Black Trees that the height of

the tree never gets beyond $log(N)$. The runtime for a red-black tree is the same as the runtime for 2-3 trees at $log(N)$ time, which can be derived from its maximal height.

4.8 Complex Trees and Tree Traversals

In this section we will go over a couple of tree traversals as well as a specific type of tree. None of these topics warrants their own individual section because we will not be going too in-depth on them.

4.8.1 Range Finding

Say we wanted to find set of elements between a given range in a Binary Search Tree. If we were to search the full tree we could end up with a bad runtime of $\Theta(N)$. However, if we use some properties of Binary Search Trees we can do better.

Earlier we learned that we could search for an element in a Binary Search Tree, on average, in $\Theta(log(N))$ time. Well in the worst case, say we wanted to get all the items between the 2 elements (insert here). Well then in that case we would traverse down to the bottom of the tree for a and to b. After this we have to move and explore all nodes to the right of a and the left of b- since this is has the possibility of being a non-trivial number, it should be included in our runtime. Given we have an average Binary Search Tree, we can say that our runtime would include the factor $\Theta(log(N))$ and we earlier said that we would have to account for the number of elements that are in our range. This leads us to have the runtime of $\Theta(log(N) + R)$ where R is the amount of elements in the range.

4.9 Hashtables

Say we wanted to store items in an array like structure, where our key has some correlation with it's index; however, what if our key is not an integer, say a String, or Animal? Or what if we have integers with huge differences like 1 and 100000000. The first case would be impossible to achieve in an organized and efficient fashion in an array because we would have no idea where keys would be- what index would an Armadillo be at? The second case is possible; however, the memory that would be taken up would be enormous, and it would be unfeasible to use so much space for only 2 items. These above 2 problems are the motivation behind hashing.

Hashing is the process of creating a number that represents some key. Usually the hashed representation of a key has to do something with an attribute or set of attributes of the object that is being hashed. It is not always trivial to make a good hashing function; however, given enough thought, we can hash names, numbers, food- the possibilities are endless!

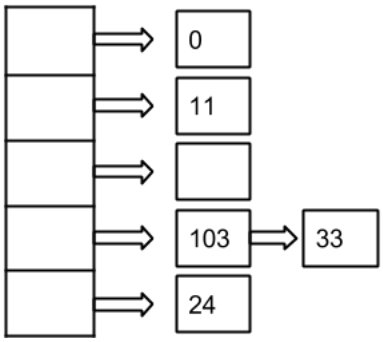
Usually, ***Hashtables*** are represented by arrays, which means that when we hash some key, we should get some index in the array- each index in our array will be referred to as a bucket and we will store keys in the bucket that they hash to. Let's start with a basic example where our key is some integer and M is the amount of buckets we currently have in our Hashtable. [2]

```
1 key%M
```

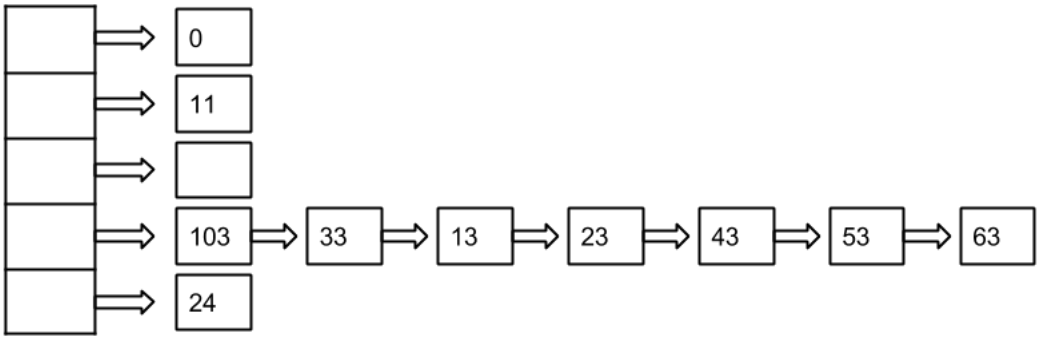
This hash function, like all hash functions, will return some integer between 0 and M - 1. We can use the number returned by this function to map a particular key to some index in the array. Let's try it out on an array of size 5, so M =5, with the calls in the following order: 0,11, 24, 103, 33.

[2]Note usually in hashing functions, we perform some arithmetic on the key before moding it, so that our encryption is more secure.

At this point, we have inserted 0, 11, 23, and 104. But wait there's a problem now, 33 and 103 hash to the same index in the array. How can we fix this? Well instead of storing 1 element within the array, we can instead store some data structure, such as linkedlist at each index, this will allow us to store more than 1 element per index.



Now that we can take care of more than 1 element, we should be good to go right? Let's try inserting the following keys: 13, 23, 43,53,63.



Do you see the problem? If we keep inserting items that have a remainder of 3 when divided by 5, our runtime to get or put an item would be no better than a linkedlist, we would just be using more memory because we have an array too. In order to fix this, while inserting items, we should have some clause that if $\frac{N}{M} \geq$ some number, where N is the total amount of items we have and M is the amount of buckets, we resize our array. This basically means if the average amount of items per bucket is greater than or equal to some number we resize the array. After resizing our array, we rehash all of our items since the amount of buckets has changed. Then , we should get a more even distribution. The question now is how should we increase our buckets? Let's consider the following options:

```
1 M = M * 2
2 M = M + 1000
```

When considering which resizing factor we should choose, we want to make sure that we do not need to resize too frequently because resizing is a relatively expensive operation. At first glance $M = M + 1000$ may seem tempting; however, it is important to realize that the number of buckets being added is not increasing with N as it grows larger. This means that, in the long run, the 1000 will not be substantial enough to

make the resizing factor negligible. The best resizing option we have is $M = M * 2$ since the amount of buckets that we add at each resize operation grows with the amount of buckets we currently have.

Let's analyze the runtime of Hashtables now. First let's consider the average case for all these functions. On average, we have a Hashtable with $O(N/M) = O(L)$ items per bucket. This means that, on average, we a Put or Get operation will take O(L) time where L is the load factor. This is because the amount of items we would need to look through would be upper bounded by O(L) as we would only look through 1 bucket.

Let's now consider the the amortized case. To do this, we will revist the average case. If we can ensure that the average amount of items per bucket, or L, is small, essentially constant, then we would only have to do a constant amount of work to find any given item. This means that Get and Put each take $\Theta(1)$ amortized time.

Now let's discuss the worst case runtime for Hashtable. In a worst case for Put, we would need to resize the array and rehash all the items, which would take $\Theta(N)$ time. There is also another case for which worst case runtime can occur. If we have a bad hashing function, it could be exploited so that, with a series of inserts, everything hashes to the same bucket, essentially making one bucket act like a huge LinkedList while the other buckets have nearly nothing in then. This would make both Get and Put take $O(N)$ time as we would need to check the entire bucket to see if an element is already in the Hashtable. This is why it is imperative that a good hashing function is used in a Hashtable.

To use Hashtables, we need to use the following functions.

```
1 public int hashcode(){
2     ......
3     return some hashcode;
4     }
5 public boolean equals(Object obj){
6     ....
7     return if obj equals your current object based off a factor of your choosing
8 }
```

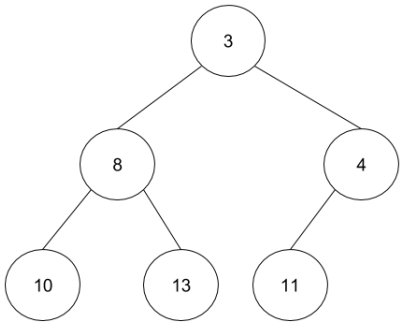
One key idea to remember when analyzing hashing functions is that any 2 items that are equal should hash to same bucket. This means that if *a.equals(b)* then *a.hashcode()* must equal *b.hashcode()*. The implication of this is that if you override the equals method, you must also overwrite the hashcode method. Another important characteristic of hash functions is that they should provide a relatively even distribution meaning that one bucket should not be hashed to a disproportionate amount compared to other buckets.

4.10 Priority Queues: Heaps

Sometimes instead of just storing data, we want to be able to get them in a certain order. To do this, we can use a ***priority queue***. The API for a minimum priority queue is as follows:

```
1 public interface MinPQ<Item> {
2         public void add(Item x);
3         public Item getSmallest();
4         public Item removeSmallest();
5         public int size();
6 }
```

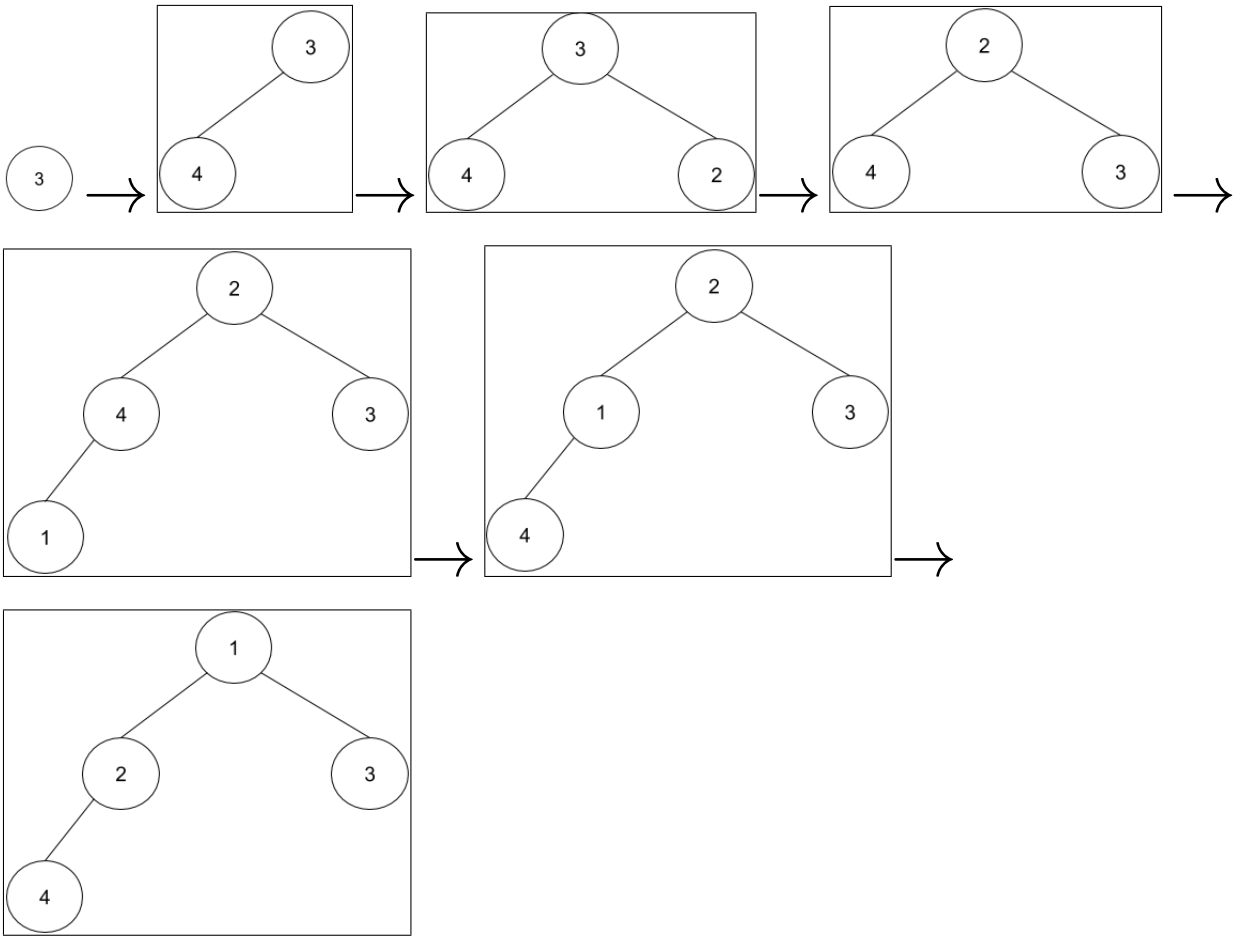
To represent priority queues, we can use a tree like structure called a heap. A heap looks like this:



This above heap is a min-heap, that is the minimum element is the item on the top. There are a few rules for min-heaps.

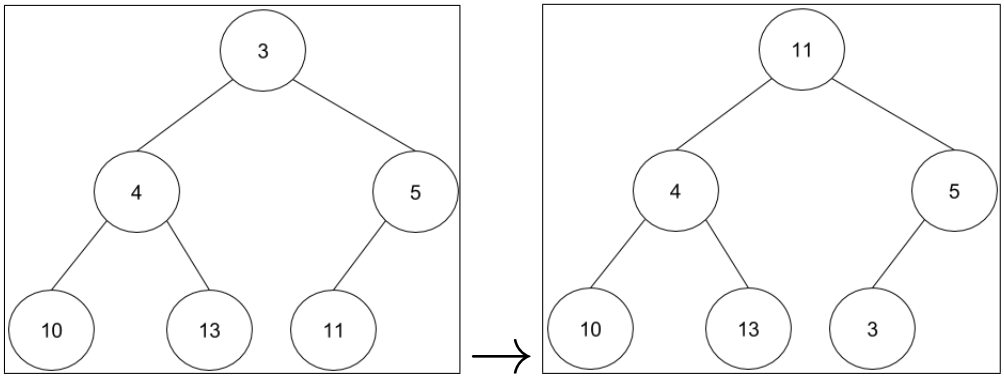
- Any child must be greater than its parent.
- The ”tree” structure must always be filled from top to bottom and left to right.
- Any node can have a maximum of two children.

Now we will build a heap from scratch for the input as follows: 3, 4, 2, 1.

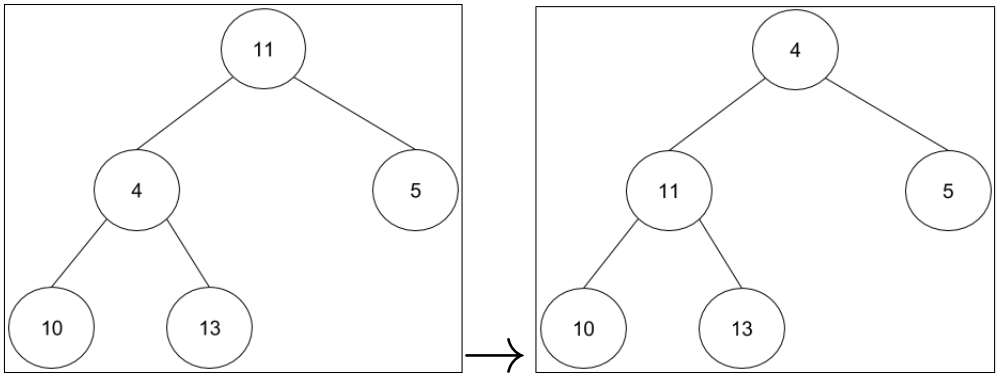


Note how when we find an element that is less than its parent, we ”swim” it up and switch it with its parent. Now, let’s go over how we can delete from a min-heap.
For priority queues in general, we only remove the minimum element, or the element with the highest priority.

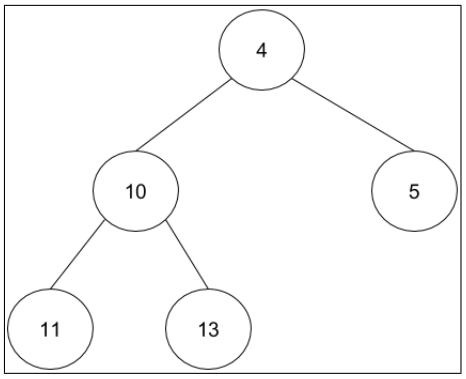
Now that we know how to add to a heap, we will go over how to delete from them. In priority queues, it is only sensible to delete the item with the maximum priority. In order to do so, we do a somewhat unintuitive approach, but one that must be done. We will start with the following heap:



You may be asking yourself, how did we get to this point? Well, when deleting the item with the highest priority, we switch the root with the item that is the rightmost item on the lowest level.



At this point, we delete the item with the highest priority (which we know is the rightmost item on the lowest level). Following this, we sink the root node. We check to see if the root is less than one, or both, of its children, if so, we choose the smallest child and swap it with the root. We then repeat this step with the same node until we cannot swap the item anymore.



We finally swap 10 and 11 because 10 is the only child of the node 11 that is less than it.

Since heaps have a consistent ordering. We can represent them in an arraylike fashion using a formula: specifically, the child of a node will always be either $2k$ or $2k + 1$ where k is the parent index. The

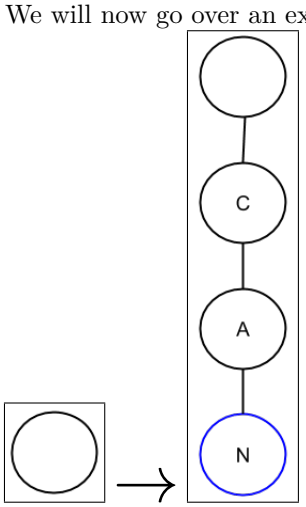
assumption being made is that the root is always at index 1. This means that 1's children will be 2 and 3, 2's children will be 4 and 5, 3's children will be 6 and 7, and so forth. *getSmallest* takes $\Theta(1)$, as it will always be the root. *add* and *removeSmallest* however, take the height of the tree- which is $\Theta(log(n))$.

4.11 Tries

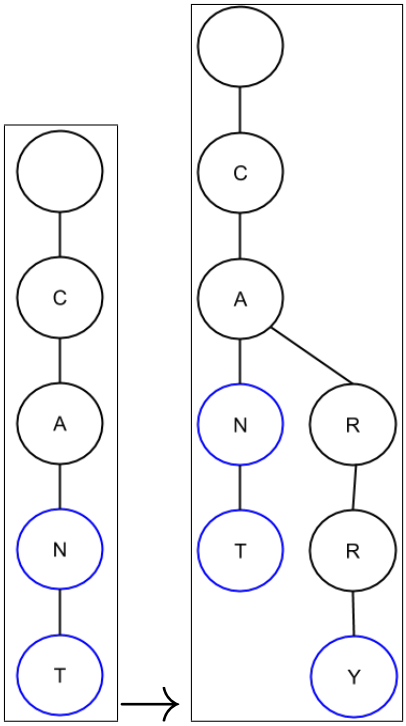
A **Trie** is a type of data structure that is used for storing strings, numbers, or just about anything that has some sort of "prefix". These are very niche data structures; however, they are very useful at what they do. Tries, like most data structures, are composed up of nodes. Each Trie node has an array of children, the array's size is dependent on how many letters are in your alphabet. Additionally, each node will have a boolean value that will tell us if the letter we are at marks the end of a word, we will call nodes at the end of words, blue nodes. The API for a trie node , assuming we are dealing with words, is as follows

```
1 public class Trie(){
2     private class TrieNode{
3        char value;
4        char[] children;
5        boolean isbluenode;
6     public void insert(String s);
7     public boolean search(String s)
8     public TrieNode root;
9 }
```

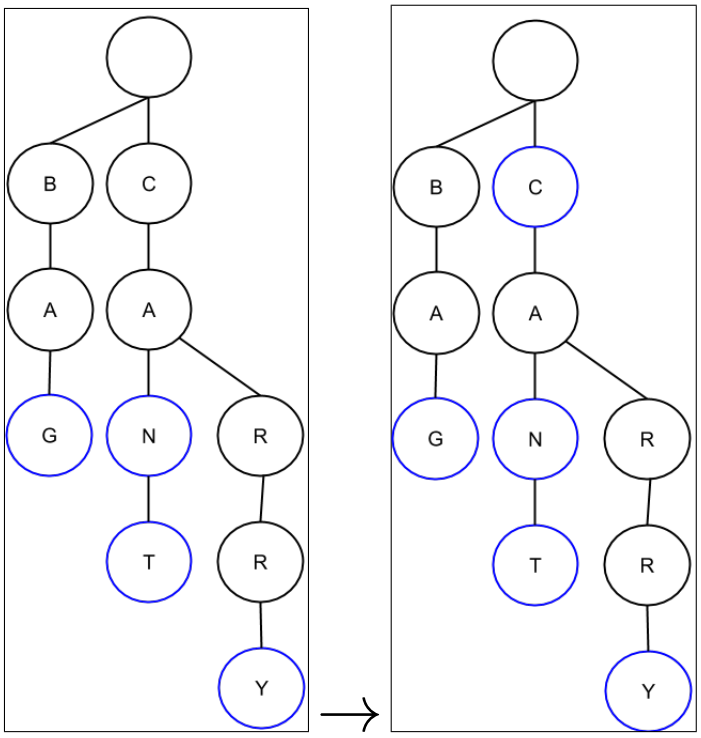
We will now go over an example of how to construct a trie.



To start off with a trie, we will have one empty "parent" node. We will see why this is necessary a bit later. We will start off by inserting the word "CAN". The first node that we insert is one which has a value of "C". Under it we put a node that has a value of "A", and finally, we have a node that has a value of "N". This last node will have it's isbluenode boolean made to true so that we know that "CAN" is a word.

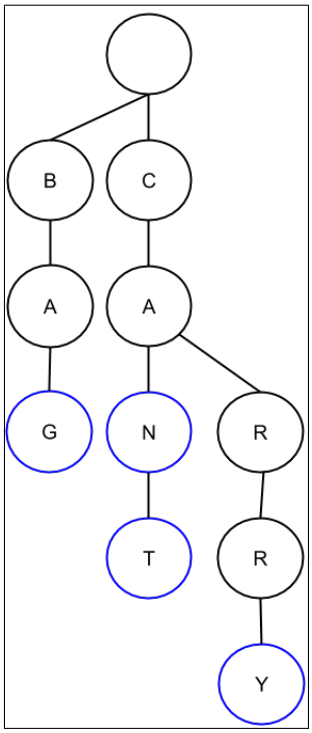


Now we will insert the word ”CANT”. To do this, we trace down our trie to see how long the longest prefix. We move through ”C”, ”A”, ”N” and realize that there is no more nodes to trace through so we create a new node for ”T” and mark it blue as it marks the end of the word ”CANT”.



Next, we insert ”CARRY”. We trace through ”C” and ”A” then realize that our ”longest prefix is done”, now we insert the rest of the nodes ”R”, ”R”, and ”Y”, marking the ”Y” node blue. Finally, we will insert

"BAG". We note that there is no prefix that starts with "B" yet. This means that we need to create our word from scratch. We create "BAG" with the "G" node being blue. Now, we will insert the string "C" into our trie. What we notice is that "C" exists as a part of "CAN"/"CANT". So instead of inserting a new C, we simply color our "C" node blue.



The final string that we will add to our trie is "FUN". We simply insert "F" as a child of the root node and then after that, we insert "U" and then "N".

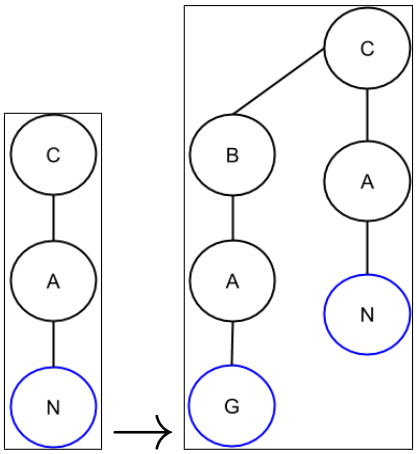
The power of the trie can be seen in the above example for "C". We do not have to create a new node for C since it is a part of "CAN"/"CANT". Other searching data structures that we have dealt with have some form of comparison. With the trie data structure, we no longer need to compare. We don't see if "CAN" is less than "CANT", we just base our searching/insertion off of prefixes. This leads to an enhanced search time.

Tries do have a pretty good runtime, specifically, their worst case runtime for insert is $\Theta(M)$ where M is the length of the string that you are attempting to insert. This runtime comes at a trade off of memory. Each node will have to have an array that is the size of the alphabet, which we will represent by R. For N keys of length L, this means that you will have $N * L * R$ space total, quite a large amount.

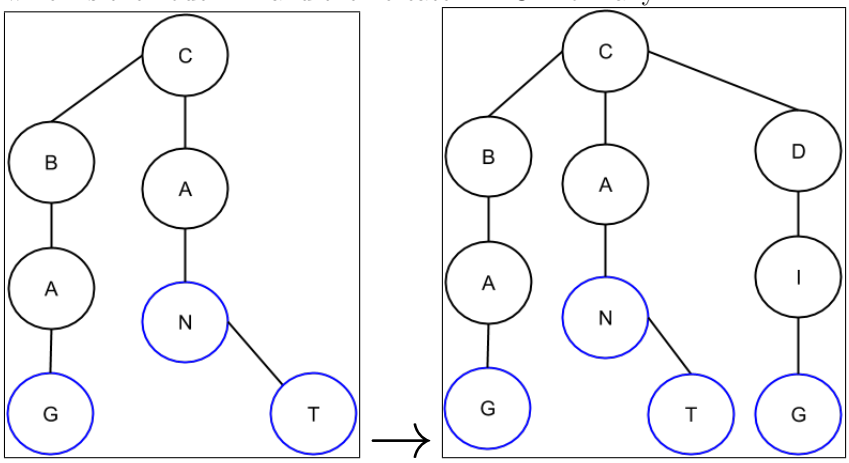
4.12 Ternary Search Tries

Ternary Search Tries are a data structure that serve the same purpose as Tries; however, they save a lot of memory, they do this by having a worse overall runtime. Instead of having an array that is the size of the full alphabet, each node has 3 sets of links- items that are less than, equal, or greater than the current node. In this case, less than means that the edge points towards a character that is less than the current one, greater than means the edge points towards a character that is greater than the node and an equal child refers to the next character in the sequence.

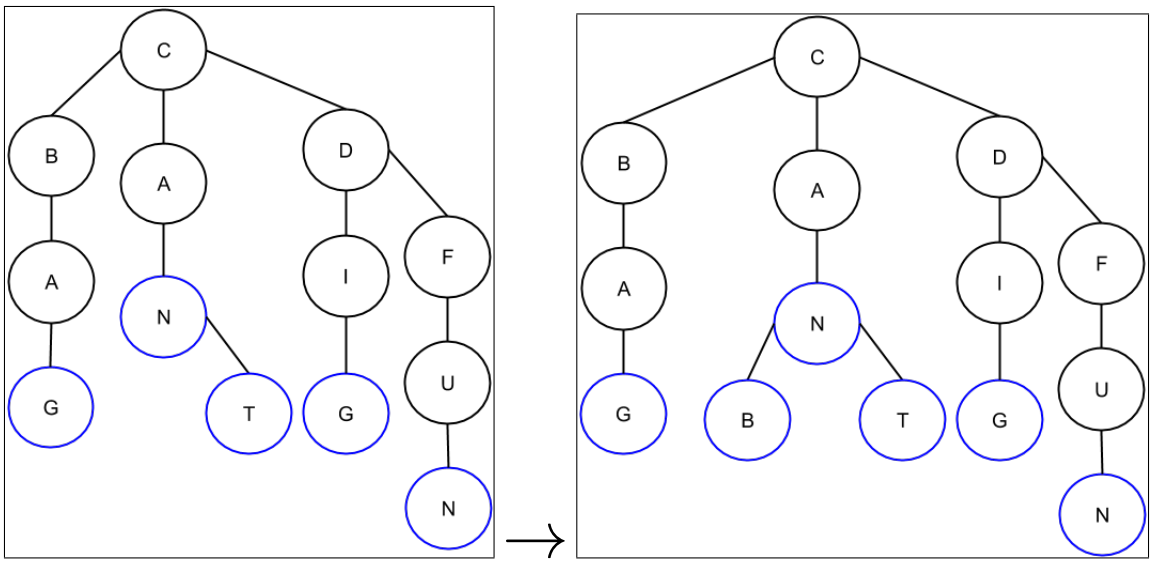
We will now go over how to construct a Ternary Search Trie:



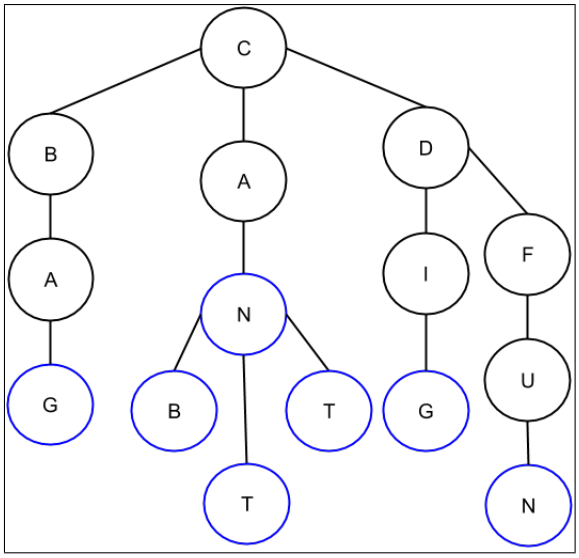
We initially start by inserting the string "CAN". The marking of a word via a blue node is true for ternary search tries just as it is for normal tries. We then insert the word "BAG". We insert a left child for "C" which is the node "B" and then create "BAG" normally.



We now insert the word "CAT". To do so, we go through our current "C" and "A" nodes. We realize "T" is not the next item in the sequence, so we move to "N" and look to the right. There is no "T" so we mae one. It is very important to see that the "T" is the right child of the "N" from the word "CAN". Following this, we insert the word "DIG". We realize that our first letter is not "C" so we look to it's right child and see that one does not exist. To fix this we create some "D" node and then follow through to normal process to complete "DIG".



The next insertion that is performed is ”FUN”. We start at C and see it has a right child ”D”. At the ”D” node, we realize we need to go one more node right but there isn’t one that exists yet. To fix this, we insert an ”F” node and then we do the normal process to finish the rest of ”FUN”
We’ll now insert ”CAB”. We trace down the ternary search trie and go down the ”C” and the ”A. At this point, we need to insert the character ”B”. We move down to the node ”N” and see it has no left child. We then insert a ”B” and our work is done.



The final insertion we do is ”CANT”. We follow the same steps as before and at the ”N” node, we see we are continuing and make a new bottom link ”T”. After this, our process is done.

In the above example, there are two ”maxed out” nodes, the ”C” at the start of ”CAN” and the ”N” in ”CAN”. Both of these nodes have a left child, an element less than them, a right child, an element greater than them, and a middle child, which continues the same word. The left child and right child of a node are continuations of the prefix above the node, but does not include the word itself.

The structure of ternary search tries are quite different from normal Tries, and while this method does save memory since we are not allocating memory for the entire alphabet, it comes at a cost of time. We can

compare the example from the Trie secton to the one in this section. The word "FUN" can be completed in 4 levels in the normal trie; however, in a ternary search trie, we must use 5 levels. This difference be even more extreme if we have words that all start on different alphabets.

In the worst case, Ternary Search Tries can take $O(N)$ time for all the insertion and deletion operations where N is the amount of keys. This can happen in the case where your ternary search trie takes on the shape of many left/right leaning linked lists. On average; however, the insertion and deletion operations take $\Theta(nlogn)$ time because that happens to be the average height of a tree/trie. The memory saved compared to a normal trie is substantial. Ternary Search Tries only take up $O(N * L)$ space where N is the amount of keys and L the maximum length of a key.

4.13 When to use which Data Structure?

As you know from reading this section, there are quite a few data structures. When tackling design questions, it can be overwhelming to think about which data structure you need to use. We will briefly discuss the use cases for the various categories of data structures.

Lists are generally used when you need to just store data in some ordered manner. Runtime isn't too important when you're using a list. The two main types of lists are **LinkedLists** and **ArrayLists**.

1. **LinkedList**
   - get() = $O(N)$
   - add() = $\Theta(1)$
   - remove() = $\Theta(N)$
   - space = $O(N)$
2. **ArrayList**
   - get() = $O(1)$
   - add() = $\Theta(N)$
   - remove() = $\Theta(N)$
   - space = $O(N)$

Sets are generally used when you have unordered data. In sets, you are not allowed to have any duplicates and you can only store keys! Essentially, sets are just maps with some dummy value. Data Structures that use sets are used when you just need to store elements.

Maps are data structures that store a key and a value. If you insert a key value pair into a map where that key already exists, you replace that key's value with the value of the key value pair you are inserting. You declare Maps in the following manner:

```
1 Map<Key type, Value type>
```

In maps, you have an immutable key and values that can be any data type. Frequently, maps can be used to see how frequently a key turns up.

Trees can be used whenever the keys in questions are implementing comparable. Trees can be used when you are attempting to find some sort of order. Specific types of **self balancing** trees are **2-3 Trees** and **Left Leaning Red Black Trees**. You would choose to use trees over hashing when the key is hard to hash. This can occur when a key is very long, and would take a lot of time to perform arithmetic on it or analyze it.

1. **Binary Search Tree**

- search() = Average: $O(logN)$ Worst Case: $O(N)$
- insert() = Average: $O(logN)$ Worst Case: $O(N)$
- delete() = Average: $O(logN)$ Worst Case: $O(N)$
- space = $O(N)$

2. **Balanced Search Trees (2-3 Trees and Left Leaning Red Black Trees)**

- get() = $\Theta(logN)$
- insert() = $\Theta(logN)$
- remove() = $\Theta(logN)$
- space = $O(N)$

Hashing data structures are used when you need $\Theta(1)$ runtime for all operations assuming that you have a good hashcode. You would also want to use hashing data sets when your data is unordered.

1. **Balanced Search Trees (2-3 Trees and Left Leaning Red Black Trees)**

- get() = Average: $O(1)$ Worst Case: $O(N)$
- insert() = Average: $O(1)$ Worst Case: $O(N)$
- remove() =Average: $O(1)$ Worst Case: $O(N)$
- space = $O(N)$

Stacks are used when you want to look at the most recent thing that has been done. Stacks are known as first in last out data types (FILO). This can be applied to search history, delivery items etc. Stacks are often used in depth first search in trees and also graphs, which we will go over in the next chapter.

1. **Stack**

- push() = $\Theta(1)$
- pop() = $\Theta(1)$
- peek() = $\Theta(1)$
- space() = $O(N)$

Queues are used when we need to view items by the order in which they were done. Queues are known as first in last out data types (FIFO). In the real world, this is used when we have to keep track of waitlists. Additionally Queues are used for breadth first search in trees as well as graphs.

1. **Queue**

- add() = $\Theta(1)$
- remove() = $\Theta(1)$
- peek() = $\Theta(1)$

Heaps or Priority Queues can be used whenever the keys we are examining implement Comparable. Anytime you need the most or least of something, use a Heap. These are not for overall order, unless it's taking out one item at a time.

1. **Heap**

- search() = $\Theta(1)$
- insert() = Average: $O(1)$ Worst Case: $O(logN)$
- delete() = $O(log(N))$
- peek() = $O(1)$
- space = $O(N)$

Tries are used when you want to perform searches and insertions using prefixes. There are 2 types of tries, normal tries and **ternary search tries**. Normal tries have a very fast speed but in exchange they take up a lot of space. Ternary search tries on the other hand are relatively slow but take up much less space.

1. **Tries**
    - M is the length of the string you are attempting to insert/search for, N is amount of keys, L is the length of the longest key, and R is the size of the alphabet.
    - search() = $\Theta(M)$
    - insert() = $\Theta(M)$
    - space = $O(N * L * R)$
2. **Ternary Search Tries**
    - N is the amount of keys and L is the length of the longest key
    - search() = $\Theta(N)$
    - insert() = $\Theta(N)$
    - space = $O(N * L)$

4.14 Conceptual Questions

1. What data structure can be used to implement a Stack and a Queue? How would this be done?

2. What is the relationship between 2-3 trees and Left Leaning Red Black trees?

3. If you wanted to find the maximum element in a hashmap, how much faster would it be than a LinkedList? A 2-3 Tree?

4.15 Difficult Questions

1. Use or modify a data structure or set of data structures to do the following operations in the provided time:

   - insert: O(log(N))
   - getSmallest: O(1)
   - getLargest: O(1)
   - deleteSmallest: O(log(N))
   - deleteLargest: O(log(N))

2. Imagine you are implementing a basic social media. You want to be able to check if 2 people are friends in constant time. You also want to be able to see if you can see if 2 people have at least 1 mutual friend in O(N) time where N is the total amount of people on the social media. Find an efficient solution to this problem

3. You are a ticket ripper at the happiest place in the world, Kartik land. Each family in your land can rip exactly one ticket, no more no less. Additionally, you want to rip the tickets of younger people before older ones, after all, Kartik land is a place for young ins to drag their parents to spend lots of money. Families may attempt to get around this by sending multiple family members with tickets, but it is up to you, the brave ticket ripper to prevent them from getting more than one ripped ticket. You are provided the familial connections of all families in whatever representation you would like. You should be able to rip the tickets of all people in O(N) time where N is the number of people in line to get their tickets ripped.

4.16 Chapter 4 Conceptual Question Solutions

1. A LinkedList can be used to implement a Stack and a Queue.

   In your Stack class, you would have a class level LinkedList. For the add operation, you would simply just call addFirst() on your linked list and for your pop operations, you would call removeFirst(). For a Queue, you would also have a class level LinkedList; however, for the add operation you would call addLast() and the remove operation would just be a call to removeFirst().

2. There is a one to one correspondence between any given 2-3 tree and a Left Leaning Red Black Tree and vice versa.

3. Trick question, a LinkedList and Hashmap would take the exact same time to find the maximum item, $O(N)$, this is because you have no clue what the maximum item is, and as a result, have to look through every element to see if it is the max. A $2-3$ tree would actually be faster, $\Theta(log(N))$ because the elements are sorted

4.17 Chapter 4 Difficult Question Solutions

1. We will fulfill all of our requirements by using a basic Balanced Search Tree. By construction, Balanced Search Trees insert in O(log(N)) time and can also delete in O(log(N)) time. As a result, the only operations that we need to account for is getSmallest and getLargest. We can trivially keep a pointer that keeps track of the smallest element and the largest element. Whenever we delete the smallest element or update the Balanced Search Tree to have a smallest element, we will update our pointer. We can do a similar process for the largest element.

   The reason why 2-3 trees are not used in the place of normal min-heaps or max-heap are really easy to implement, as they can be visualized using arrays. This also saves a lot of memory compared to Balanced Search Trees which can tend to be a bit expensive.

2. To solve this problem, we will associate each user with a weighted quick union with path compression. To check if 2 people are friends, we would go to one of the users and do a simple find operation. This would take almost amortized O(1) time. To see if people have a mutual friend, we would need to go through each one of the first person's friends and see if they are friends with the second person. We would perform find in each one of the first person's friends. This upperbounds the work by O(N).

   An interesting way to solve this problem is a more algorithmic approach. You can compute a unique prime number for each person on your social media upon construction. To find N prime numbers takes Nlog(log(N)) time. This would be faster than the weighted quick union with path compression implementation as we would construct a data structure for each item, which would take $O(N^2)$ time.

   In addition to the associated prime number, each person would have a "friend product" attribute which would be the product of all of it's friends' primes. To check if two people are friends we can just divide the friend product of one person by the prime number associated with the other. If it the result is an integer, we know that they are friends, if not, they are not friends. To see if two people have mutual friends, we will find the greatest common divisor using Euclid's Algorithm. This will take $O(log(min(a, b))$ where a is the friend product of the first person and b is the friend product of the second person.

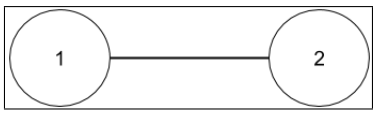
3. Use a modified Weighted Quick Union With Path Compression to store family relationships. Create a wrapper class for each person in line and store a pointer to their corresponding Weighted Quick Union with Path Compression. When reaching a person in line, check their family Weighted Quick Union With Path Compression. If the root is marked do not rip their ticket, otherwise rip the ticket and then mark the root.

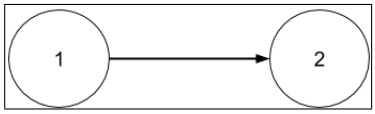
Chapter 5

Graphs

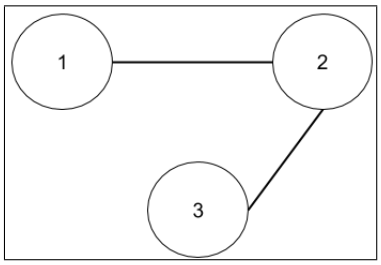
5.1 Basics of Graphs

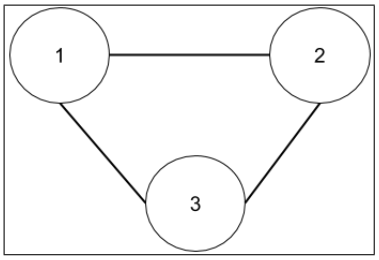
Graphs, in the most simple way, are a set of nodes with edges between them. There are a few types of graphs, one way of categorizing is **directed** and **undirected**. An undirected graph is one where the edges is "2 way" which means if there is an edge between vertex A and B you can go from A to B and B to A. A directed graph is one where the edge only goes "1 way" so an edge between A and B will either allow you to go from A to B or B to A. Below is are examples of undirected and directed graphs respectively.





Another two categories of graphs are ***cyclic*** and ***acyclic***. A graph is cyclic if there is a cycle. A cycle exists if at any point there is a path where the start and end vertex are the same. Below are examples of acyclic and cyclic graphs respectively.



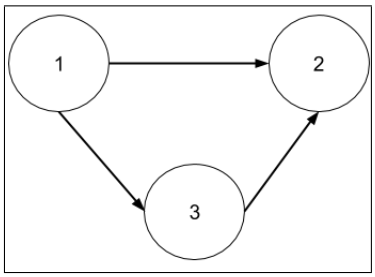


Two nodes or vertices are ***adjacent*** to each other if they have an edge between them. Sometimes, edges can have some ***weight*** on them which represent some sort of cost that it takes to cross the path. Vertices are ***connected*** if there is an edge between them; graphs are **connected** if all vertices are connected.

5.2 Representations of Graphs

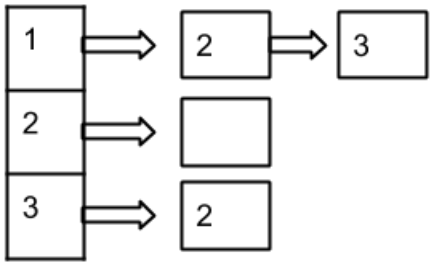
There are a 2 main ways to represent graphs: ***Adjacency Matrices*** and ***Adjacency Lists***.

An Adjacency Matrix is represented with a 2d array that is N x N. The x and y axes of this 2d array will be the vertices and the value at each "block" at row r and column c will be 0 or 1 (or some other true or false distinction value) if the vertices r and c are connected. Below is an example for how an Adjacency Matrix would look for a graph.



	1	2	3
1	0	1	1
2	0	0	0
3	0	1	0

The other representation of graphs, which tends to be used more frequently, is the Adjacency List. An Adjacency List is similar to a HashMap in that they both are represented with an array of "buckets". Below is a an example of how an Adjacency List would look for the same graph as above.



To compare the above two representations, we will take a look at the Graph API and see what functions what we need to be able to account for.

```
1 void addEdge(int v, int w) //adds an edge between the vertices v and w
2 Iterable<Integer> adj(int v) // returns an iterable that has all the certices adjacent to vertex v.
3 int V(): //number of vertices in the graph
4 int E(): // number of edges in the graph
```

To add an edge in the Adjacency Matrix, we would simply just go to the proper index in the 2-d array and replace the false value with a true one. For the Adjacency List, you would go to the proper index and add an element to the end of the bucket. As a result, this takes $\Theta(1)$ time.

Finding the vertices adjacent to a certain vertex in an adjacency matrix would require you to go to the proper row and go through each one of its columns, adding the vertex if there is a true value, as a result, this takes $\Theta(V)$ time since you need to go through V buckets. In an Adjacency List, you simply go return the corresponding bucket since the bucket contains all the adjacent vertices, as a result, this takes $\Theta(1)$ time.

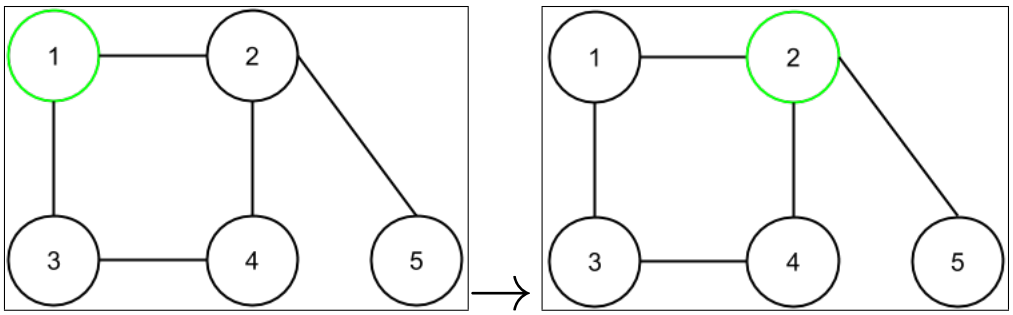
Finally, we will look at the space used. For the Adjacency Matrix, regardless of how big it is, there will always take $\Theta(V^2)$ space because the 2d array is V x V. The Adjacency List will take up $\Theta(E + V)$ space (whichever term is bigger).

So which representation should be used? Well it really depends on your graph. If the graph is very sparse, it does not make sense to use $\Theta(V^2)$ space, so an Adjacency Matrix would be the superior choice. However, to see if two vertices are connected, it can be faster to use and Adjacency Matrix. A specific case that this may occur is when the graph is fully connected, that is the edges amount is equal to V^2. This means that a bucket would be V length long and you may need to iterate over all of it in an Adjacency List representation; however, and Adjacency Matrix would allow you to plug in indices and see connectivity in constant time.

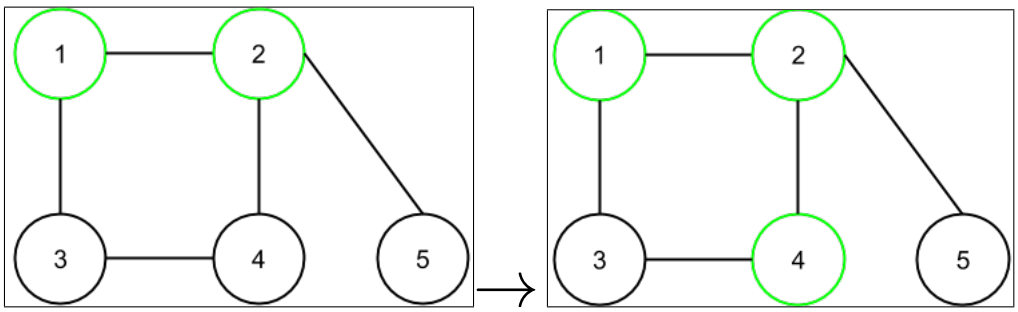
5.3 Depth-First Search

We will cover two broad ways of searching, ***Depth-First Search*** and ***Breadth-First Search***. Let's start off with Depth-First-Search.

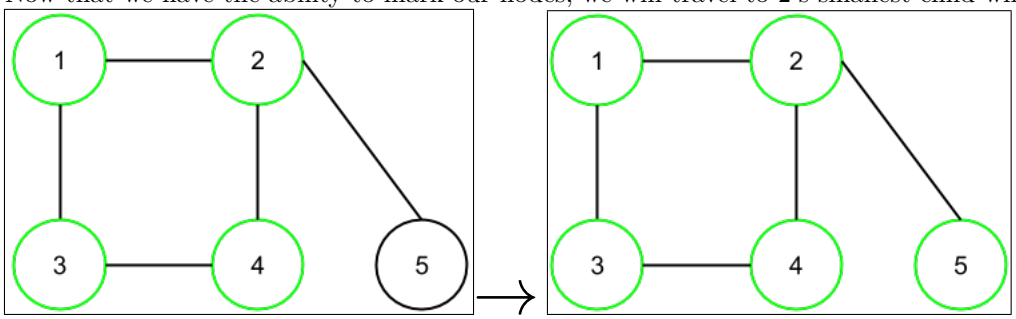
Depth-First Search is a typical recursive strategy that involves exploring the subtrees of children. Let's step through the steps of Depth-First Search and gain some intuition for the process. The green node is the node that we are currently on



The first two steps are relatively straightforward, we start off at the node 1 and then we travel to its smallest neighbor which is 2. The problem is that smallest neighbor of 2 is 1, which is the node we just came from. We will end up having some sort of infinite loop. To solve this problem, we will keep track of nodes that we have visited already. So this is how the graph would now look.



Now that we have the ability to mark our nodes, we will travel to 2's smallest child which is 4.



These are the final steps to Depth-First Search. We go to 4's smallest neighbor which is 3. After this, we realize that there are no neighbors of 3 that are not marked as a result, we recurse to our parent node 4. There are no unmarked neighbors of 4 so we recurse to its parent, which is 2. 2 does have an unmarked neighbor, 5, so we visit it. 5 has no more neighbors so we go back to 4 and we realize it has no more kids so it goes to 1. 1 no longer has anymore neighbors that are unmarked (since 3 was marked through 4) so our process is done.

Now that an understanding of what Depth-First search is has been established, let's discuss how to properly implement this. The main data structure that will be used is a **Stack**. For our earlier graph, our stack would initially be the element $\{1\}$. After 1 is popped, we place it's children in the stack making the stack $\{2, 3\}$. Following this, 2 is popped which leads to a stack of $\{4, 5, 3\}$. Once 4 is popped our stack looks like $\{3, 5, 3\}$. When the final 3 is the only element in the
stack, it is noted that we have already visited the node, since it is marked, so there is no need to visit it again.

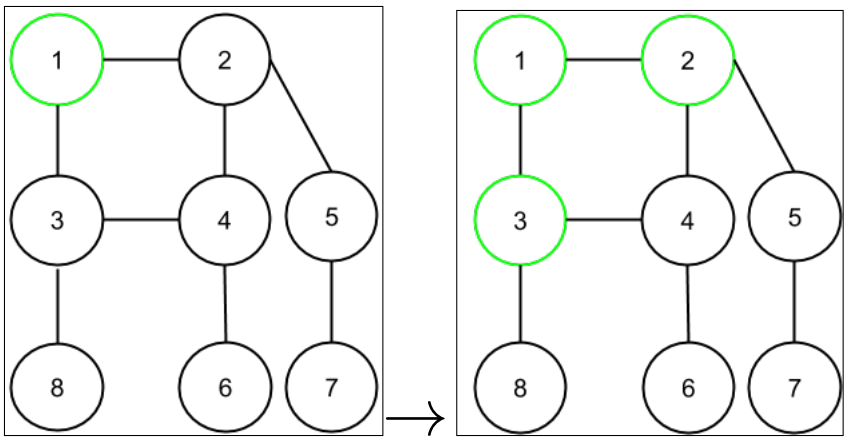
As we mentioned earlier, if we go with a naive approach of Depth-First Search, it has the possibility of failing because you may end up getting into an infinite loop. To solve this problem, we suggested the "marking" of a node, but now we will go over how to actually implement it. A nice intuitive way of approaching this problem is having a boolean array that keeps track of if a node has been visited. All elements in the array are initially False, and they are marked as True once they are visited. The code for this traversal can look like the following:

```
1  public class DepthFirstDemo{
2      private boolean[] marked;
3      private int[] edgeTo;
4      private int s;
5      public DepthFirstDemo(Graph G, int s){
6          marked = new boolean[G.V()];
7          edgeTo = new int[G.V()];
8          this.s = s;
9          dfs(G,s);
10         }
11     private void dfs(Graph G, int v) {
12         marked[v] = true;
13         for (int w : G.adj(v)) {
14         if (!marked[w]) {
15             edgeTo[w] = v;
16             dfs(G, w);
17                 }
18         }
19     }
20 }
```

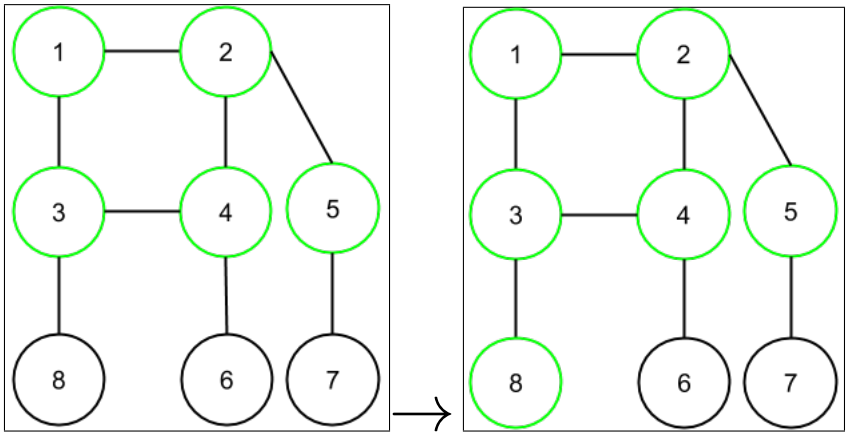
This code runs in $O(V + E)$ time when represented by an Adjacency List. We can derive this by realizing that a vertex may only be visited once (V) and in worst case, you iterate over all the edges (E) which gets us to $O(V + E)$. The amount of memory that this takes up is worst case $O(V)$ or the size of the stack.

5.4 Breadth-First Search

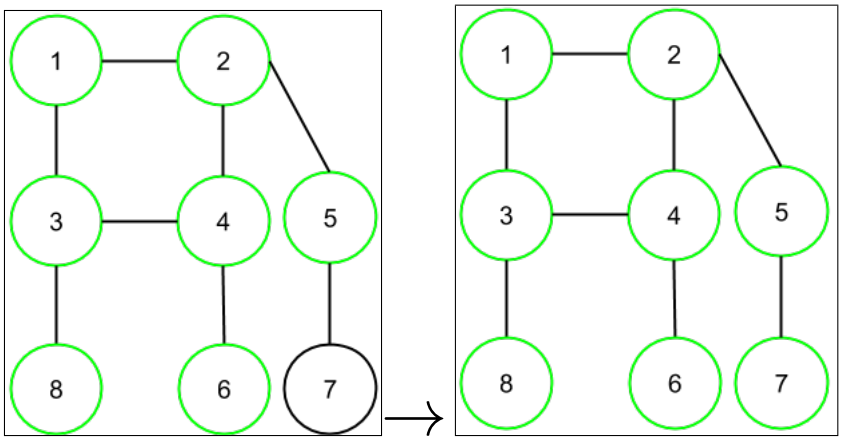
To start off this section, let's consider the problem "find the shortest path between 2 vertices" and let's approach it with our current method of Depth-First Search. Running the algorithm, you can see that it fails in certain cases such as the following problem, where we are trying to find the shortest path from 1 to 3: In this case, using Depth-First Search, we would start from 1 then go to 2 and from there go to 3; however, the shortest path would have been to go directly from 1 to 3. So solve this problem, we can use and algorithm called . Breadth-First Search differs from Depth-First Search as instead of going through one child's sub children, we visit all of a node's neighbors before exploring the neighbor's of those nodes. Let's walk through an example, in it, we will keep a similar way of marking nodes that we did for Depth-First Search:



In the initial step, we start off at node 1. We then mark the node and go to its neighbors, which are 2 and 3.



We first go to 2's neighbors, 4 and 5; however, before we visit their neighbors, we go to node 3 and visit its neighbor. 3's neighbors which are 4 and 8.



These last few steps involve our last level. We go to 4's neighbor, 8, and after that, we go to 5's neighbor 7. After this, we have finished our Breadth First Search.

Unlike Depth-First Search, we will not use a stack as our underlying data structure for this process.

Instead we use a queue. The queue for the above graphs starts off as $\{1\}$, we then dequeue it and enqeue its neighbors $\{2, 3\}$. We dequeue 2 and enqueue its neighbors which makes the queue $\{3, 4, 5\}$, after this, we enqueue 3's neighbors $\{4, 5, 4, 8\}$. Dequeuing 4, we get $\{5, 4, 8, 6\}$, then after 5 $\{4, 8, 6, 7\}$. We dequeue 4 again; however, since it has already been visited, we do not need to visit it. this leaves us with $\{8, 6, 7\}$. We dequeue each of the following and realize that they do not have any neighbors which leaves us with an empty queue.

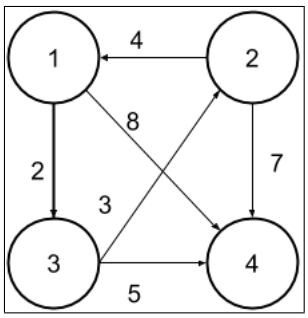
We will now have an example implementation of a Breadth-First Search

```
1  public class BreadthFirstDemo {
2      private boolean[] marked;
3      private int[] edgeTo;
4      private final int s;
5
6      public BreadthFirstDemo(Graph G, int s) {
7          marked = new boolean[G.V()];
8          edgeTo = new int[G.V()];
9          this.s = s;
10         bfs(G, s);
11     }
12
13     private void bfs(Graph G, int s) {
14         Queue<Integer> q = new Queue<Integer>();
15         marked[s] = true;
16         q.enqueue(s);
17         while (!q.isEmpty()) {
18             int v = q.dequeue();
19             for (int w : G.adj(v)) {
20                 if (!marked[w]) {
21                     {
22                     edgeTo[w] = v;
23                         marked[w] = true;
24                         q.enqueue(w);
25                         }
26                 }
27             }
28         }
29     }
30 }
```

Breadth-First Search, like Depth-First Search runs in $O(V + E)$ when we use an Adjacency List for the exact same reason.

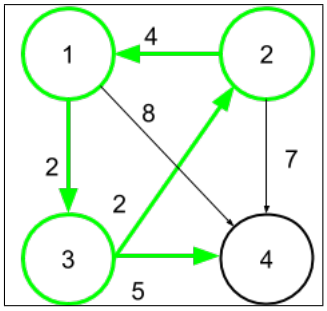
5.5 Shortest Paths: Dijkstra

As we mentioned earlier, graphs can have "weights" associated with an edge so it would look like the following:



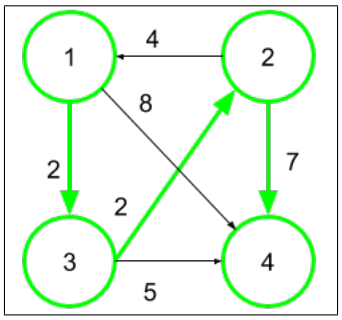
Now let's consider the following problem for the above graph: What is the shortest path (or least weight path) from the node 1 to all the other nodes?

Before we go further with this problem, let's gain some more intuition. Take the following path, is it ever possible that this would be part of our Shortest Path Graph?

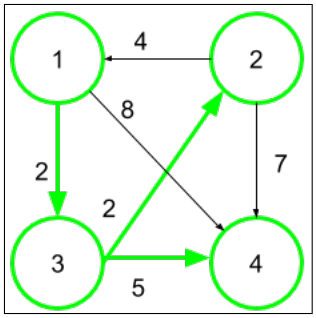


The answer is no! The reasoning is that if we have a shortest path to a vertex, there should never be a reason to visit it again because we would have reached it in less distance earlier. If we are finding the shortest distance from 1 to every other node, there is no purpose to visit 3, go to 2, and then back to 1. This leads us to make an interesting observation, **the shortest path from a node to any other node will always be a tree- in other words, there are no cycles**.

Now let's consider an approach to solving this problem. One approach that may be intuitive is perform a Depth First Search Like traversal. We can perform a small modification where at each step of the algorithm, we pick the edge with the minimum weight assuming that the node has edges coming out of it that do not result in a cycle. Assuming we start at the node 1, our resulting graph would become:



Is this correct? Well let's take a closer look. It seems like there is a shorter path from 1 to 4 if we go through from 1 to 3 to 4. The cost of this path will be 7 as opposed to the path that goes from 1 to 3 to 2 to 7 which has a cost of 11. This graph is pictured below:



So we know that we need to do something more intelligent than just picking the best edge from a certain node.

Let's try a new approach- one that lets us capitalize on nodes that we have already visited. The approach that we will take involves growing the Shortest Paths Tree by one node and one edge at each step . We do this by a a node that fulfills 2 constraints:

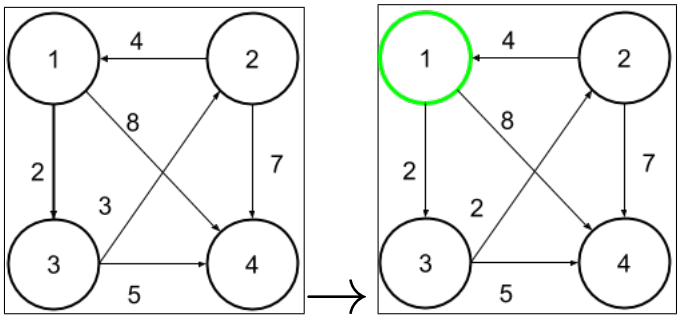
1. The node is reachable from our current Shortest Path's Tree. That is, by crossing 1 edge from some vertex in our current Shortest Path Tree we can reach the node.
2. The node is the least distance away from the start (taking into account the distance traveled so far). This idea is different than the initial Depth First Search Idea because instead of using information from only one node, we use information from all the nodes that we have explored so far.

The reason that this works is because any node that is in our Shortest Path Tree will be closer than any vertex not in it. This means that we can be sure that when we add a vertex to this shortest path tree that there is no possible way for us to reach it in less distance.

Now that we have an idea of what we want to do, let's think about how we can implement it. We need to keep track of 3 things, the actual shortest path, the minimum distance to each element, and the node that is the shortest distance away from the source. To keep track of the path, we can have an array of the "parent edge" for each vertex (what vertex leads to it) and backtrack until we reach the source, which would have no parent edge, this array would be called an **edgeTo** array. To find which element has the minimum distance at a certain step, we could use a ***Minimum Priority Queue*** that contains all the vertices and perform a deleteMin operation and add the returned node and its "parent edge" to the shortest path graph. When we explore a node and find edges to its children we want to see if we need to update the value for those children in the Priority Queue; however, looking through a Priority Queue for an element can be expensive O(N) time. To get around this inefficient query time, we can keep a **distTo** array which will keep track of our current minimum distance to an element. This has the added benefit of keeping the distance to each vertex after our Priority Queue is completely empty.

Before going through an example, let's first narrate the steps of our new algorithm. We wuill
When checking neighbors, we check to see if the distance to it is less than the current distance we have noted for it (the distTo element for that vertex), this process is called **edge relaxation**. If the distance to the node is greater than the current distance we have noted, then we do not change anything. Otherwise, we change the node's priority in the Priority Queue and the entry in the distTo array to be the smaller distance. We would also change the entry in the edgeTo array to be the edge that creates the smaller distance. To think about edge relaxation, consider some rubber band. At some given moment, we are "stretching" the rubber band across a path that connects 2 vertices. When we find one that is better, we can stop considering the other path and as a result are "relaxed" since we no longer are stretching our rubber band over 2 paths.

This approach is called ***Dijkstra's Algorithm***. We will walk through an example of Dijkstra's, the left image in every sequence is the graph at the step we are on and the table next to it is the distTo array and the edgeTo array. An important note to realize is that entries in the distTo array are the priorities of elements in the Priority Queue.

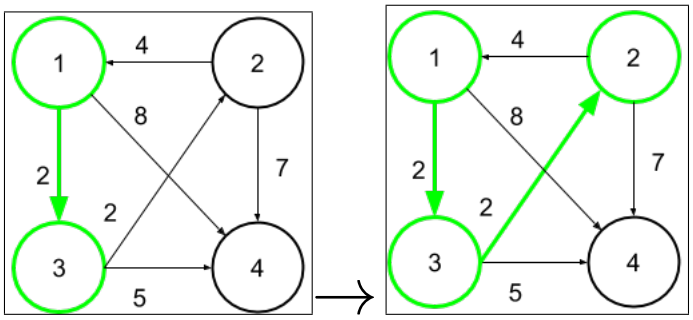


Vertex	DistTo	EdgeTo
1	0	--
2	∞	--
3	∞	--
4	∞	--

→

Vertex	DistTo	EdgeTo
1	0	--
2	∞	--
3	2	1→ 3
4	8	1→ 4

We start off at the node 1 and have its priority be 0 while all of the others nodes have a priority of infinity. The distTo array would only have an entry, 0, for the vertex 1 while all other entries would be infinity. The edgeTo array would be null for all elements. In the next step, we perform a removeMin operation and delete the vertex 1 from the Priority Queue. When this occurs, we add an entry for all of its neighbors in the distTo array and the edgeTo array. The distTo element would be equal to the edge length from the source (an entry 2 for vertex 1 and entry 8 for vertex 4). The edgeTo element would be the edge between the source and the neighbors.



Vertex	DistTo	EdgeTo
1	0	--
2	4	3→ 2
3	2	1→ 3
4	7	3→ 4

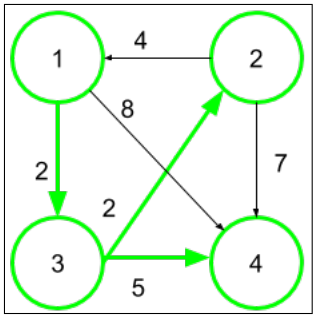
→

Vertex	DistTo	EdgeTo
1	0	--
2	4	3→ 2
3	2	1→ 3
4	7	3→ 4

We perform another deleteMin operation on our Priority Queue. This time, we remove the vertex 3. We explore 3's neighbors and we discover that we can reach a new vertex, 2. We update our distTo array and

priority to have the value dist[3]+ e(3,2), as this the the total distance that 2 is from the source. We would also update the edgeTo array for vertex 2 with the entry of the edge between 3 to 2. In addition to this, we see that we have found an alternate way of reaching the node 4. Since 4 is not part of our Shortest Path's Tree yet, we will compare the proposed cost of taking this new path to the path we currently have recorded. We compare distTo[3]+e(3,4) = 7 with distTo[4]=8. Since the former is equal to a smaller value, we change the priority of the 4 node in the priority queue and the distTo value of the entry 4 to be equal to 8. We also change the edgeTo array to have the edge from 3 to 4 as its value.

Once again we perform a deleteMin operation, this time we remove the vertex 2 as it has a lower priority/distance from the source than the vertex 4 (5 opposed to 11). We explore 2's neighbors and see that we have the vertex 1 and 4 as its neighbors. 1 is already part of our Shortest Path Tree (has been removed from our Priority Queue), so there is no way that we could find a cheaper route to it, so we do not change anything in the Priority Queue, distTo array, or edgeTo array. This can be validated by comparing distTo[1] = 0 with distTo[2] + e(2,1) = 9. Now we will check 2's next neighbor, 4. We compare distTo[2] +e(2,4) = 8 and distTo[4] = 7 and see that distTo[4] is cheaper. As a result, we will will not change anything.

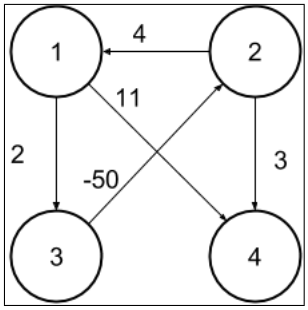


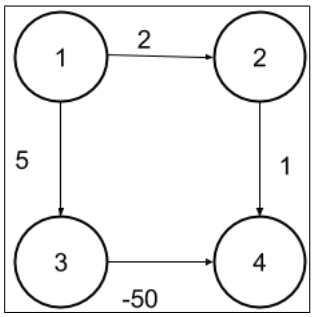
Vertex	DistTo	EdgeTo
1	0	--
2	4	3→ 2
3	2	1→ 3
4	7	3→ 4

We perform 1 more delete min and remove the node 4 and add it to our Shortest Path Tree with the edge in the edgeTo array, the edge between 1 and 4. We realize that our Priority Queue is completely empty, which is the terminating condition for Dijkstra's algorithm, meaning we are done. In the above picture, there is only 1 graph and 1 distTo array, and they correspond to the same step.

As we learned earlier, we use a Priority Queue to see which element we should add to our Shortest Path's Tree where the priority of an element is the distTo entry for the vertex that its edge is coming from + the weight of the edge between the element we are looking at and the vertex connecting to it. We have $\Theta(|V|)$ insertions, $\Theta(|V|)$ deletions, and $O(|E|)$ decreasing of priorities. A single one of any of these operations would take $O(\log(|V|)$ time. As a result, our runtime is $O(|E| \log |V| + |V| \log |V| + |V| \log |V|)$. In a connected graph, E is always between $|V| - 1$ and $|V|^2$ so we can simplify our expression to be $O(|E| \log |V|)$.

Dijkstra's is a pretty great algorithm; however, there are some pitfalls. Consider the following graphs and try to identify why running Dijkstra's would not return a valid answer.





In the first graph, if we took the path {(1,3), (3,2), (2,1) , (2,3),(3,2), (2,1)...} we could get into a never-ending cycle because our shortest path would be negative infinity. This is because we could always get a lower path weight by taking the cycle again. In the second graph, we would start at vertex 1, then we would go to vertex 2 because it is closer to the source than vertex 3. We would then travel to vertex 4 because the distTo[4] would be 3, which is less than distTo[3] (5). In the next step, we travel to vertex 3 would get a nasty surprise: since the edge between 3 and 4 is negative, there was a shorter path to the vertex 4 that we did not consider! One that actually has negative weight.

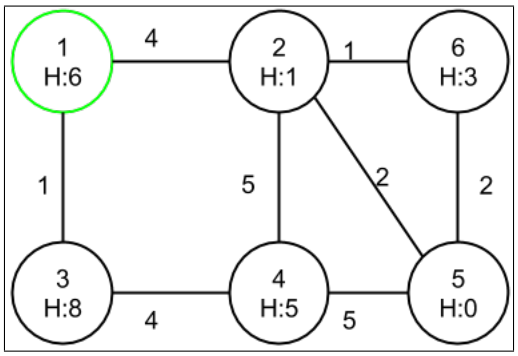
These 2 examples show us a very important requirement for Dijkstra's Algorithm: the graph we are running the algorithm on can have no negative edges. The reasoning behind this is that if an edge is negative, there could be a different shortest path to a vertex that we do not consider since negative edge weights would take away from the total weight of a path.

5.6 Shortest Paths: A*

Sometimes, we don't want to visit every vertex. Instead, we want to find the shortest path from one vertex to another. Dijkstra's algorithm would not be optimal as we would take the time to explore every vertex instead of the ones that we need. For example, if we used Dijkstra's to go from California to South America, we would go all the way to Africa! That really would be unoptimal. As a result, we would use a algorithm called ***A****. The A* Algorithm works off of the principle of a **heuristic**, which is an estimation to the cost that it will take to travel to the destination from a certain point. For A* to work, you must have:

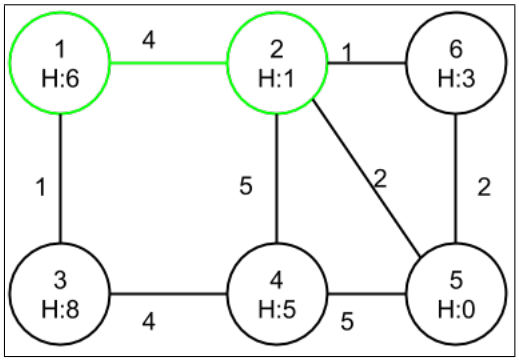
- Heuristics that are not overestimated (they are admissible)
- Consistent Heuristics

As long as the heuristics are admissible, you will always ind the correct shortest path between two nodes. Once again, we will do a walkthrough of how to run the A* algorithm. The H inside of each node refers to the heuristic. We will be running A* to find the shortest path from 1 to 5.



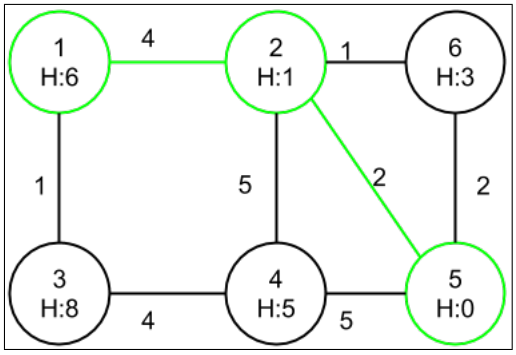
Vertex	DistTo	Heuristic	Sum
1	0	6	6
2	4	1	6
3	1	8	9
4	--	--	--
5	--	--	--
6	--	--	--

Here we start off at vertex 1. We enqueue the neighbors, 2 and 3, the distance to them, and their heuristic. The distance traveled so far is 0 from the vertex.



Vertex	DistTo	Heuristic	Sum
1	0	6	6
2	4	1	6
3	1	8	9
4	9	5	14
5	6	0	6
6	5	3	8

The distance to the node 3 is less than the distance to 2; however, since we are running A* and not Dijkstra's algorithm, we will take a look at the heuristic. We add the heuristic to the distance required to travel to the vertex and we see that going to 2 actually should lead to a shorter path. As a result, we will travel to 2.



In our final step, we take a look at the new neighbors of 2, and we see that 5 has the smallest sum of heuristic + distance traveled so far. As a result, we go directly to 5, and that concludes our A* search. As it can be seen, we avoid going to any unneeded vertices and go straight to our path. This concept is extremely powerful in the world of routing, and it saves us a lot of time.

5.7 Minimum Spanning Trees: Prim's Algorithm

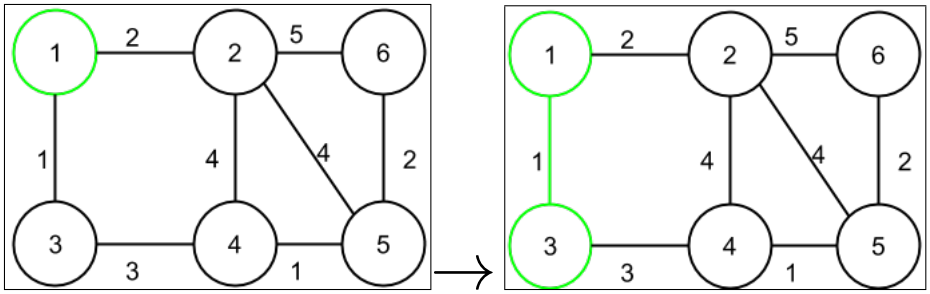
A Spanning Tree is tree (a graph with no cycles) that includes all the vertices. The **Minimum Spanning Tree** is a Spanning Tree that has the minimum weight. To find the Minimum Spanning Tree, we can use 1 of 2 algorithms, **Prim's Algorithm** or **Kruskal's Algorithm**.

Before we get started on these Algorithms, we will discuss the Cut Property. The cut property says that, given two sets, the smallest edge that connects the two edges will be a part of the Minimum Spanning Tree.

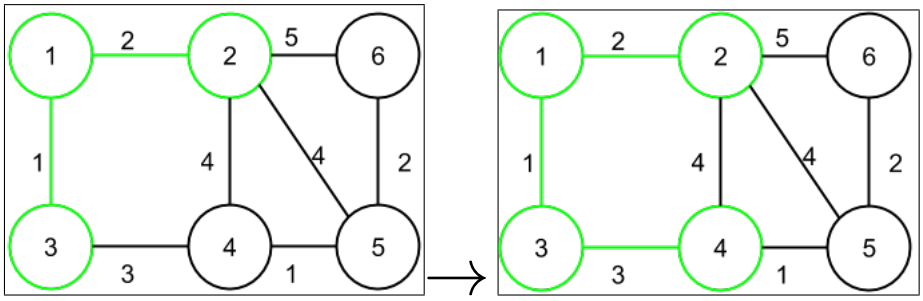
Prim's Algorithm is very similar to Dijkstra's Algorithm in general. You have a starting vertex and at each vertex that you travel to, you pick to go the the closest vertex that does not cause a cycle. The vertex that is chosen does not need to be the one that you just visited, it just has to be a neighbor to a vertex that you have visited during the algorithm.

Prim's Algorithm uses a Priority Queue in order to pick the minimum edges that are adjacent to the current vertex. This is very similar to Dijkstra's algorithm except vertices are not given the priority based

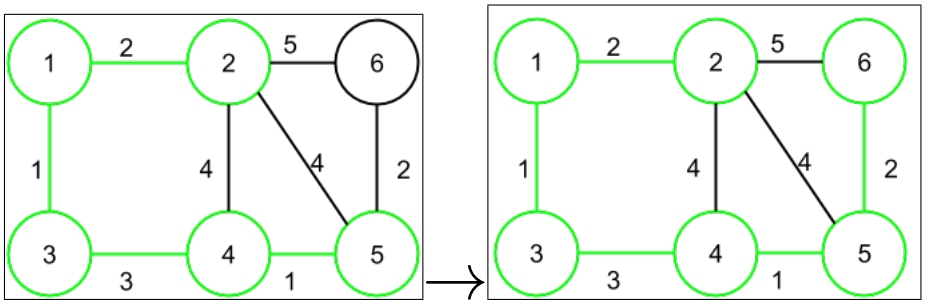
off distance from the starting point, it is given the weight of the minimum edge connected to it. Let's apply Prim's Algorithm.



We started off at the vertex 1. We enqueue the edges that are next to 1, which are of weights 2 and 1. We know that 1 is less than 2, so we go to vertex 3.



We enqueue the neighbors of 3 at this point, and we compare its neighboring edge of 3. Currently, we have 2 edges of weights 2 and 3, 2 is less than 3 so we go visit 2. We enqueue it's neighboring edges which are 4, 4, and 5. The edge of length 3 is the smallest edge, so we traverse it and visit vertex 4.



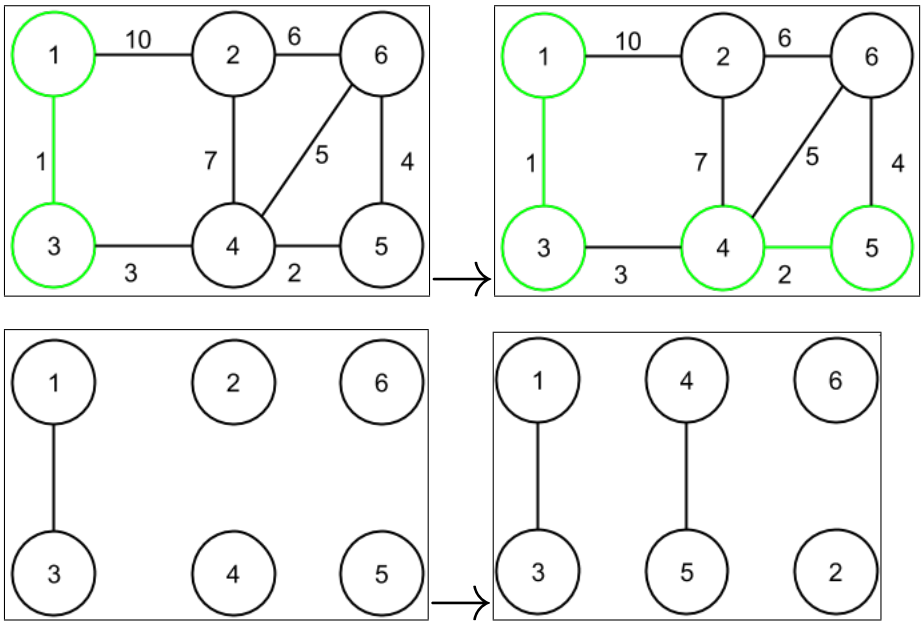
We enqueue 4's neighbors which have not yet been enqueued, which leads us to add an edge of 1 that is between 4 and 5. This edge is currently the smallest edge so we move to 5. 5 has one edge not enqueued yet and it has a value of 2, so we put it in, see it's lower than the others and we are done.

The runtime of Prim's Algorithm is $E\log(V)$ because we have $Vlog(V)$ time for inserting all the vertices into the heap, $V\log(V)$ for the deleting the min and $E\log(V)$ for changing the priority of the vertices. We sum these up to get $O(E\log(V) + V\log(V) + V\log(V))$ this can be simplified to $O(E\log(V))$.

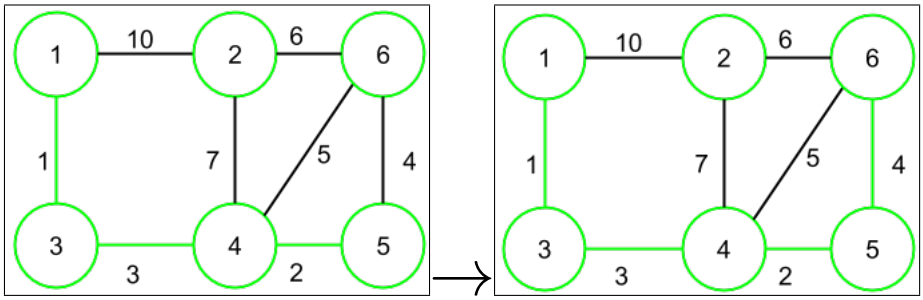
5.8 Minimum Spanning Trees: Kruskal's Algorithm

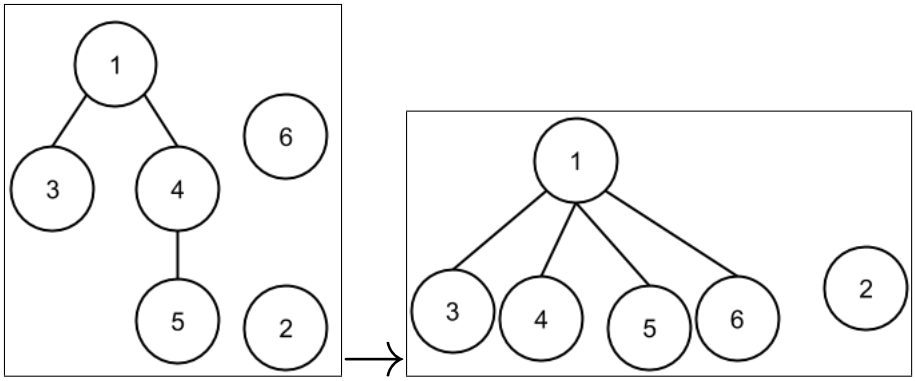
Kruskal's Algorithm runs slightly differently than Prim's Algorithm. Instead of starting from some vertex, you put all the edges into a priority queue and add the ones that are the smallest and don't cause a cycle to your Minimum Spanning Tree.

The difference that Kruskal's algorithm has with Prim's requires that a different data structure be used. The edges are sorted from least to greatest, and a core Union Data Structure is used. In this application of a Union Data Structure, each vertex is it's own "tree" and it gets connected to another vertex. This Union Data Structure allows us to ensure that we do not have cycles in a very quick manner. Here is a step by step illustration of how to Union Data Structure works for Kruskals. We will go through an example of how kruskal's works. Underneath each graph will be the Weighted Quick Union with Path Compression that corresponds to it.

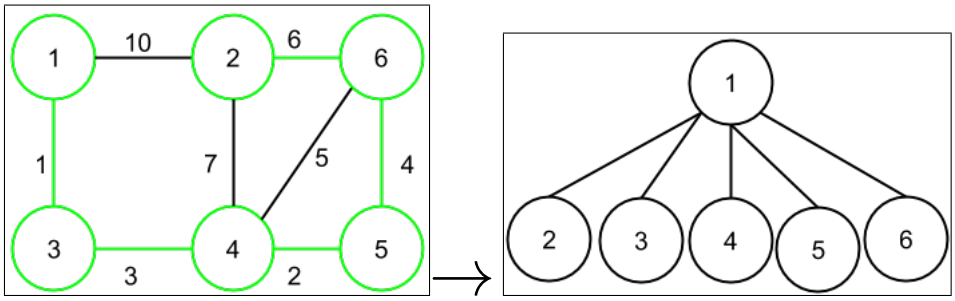


In this initial step, we find the smallest edge, which lays between 1 and 3. We then connect them in our disjoint set data structure. After this, we find the next smallest edge which lays between 4 and 5, we also connect these in our disjoint set structure.





In this step, we again found the next smallest edge which is between 2 and 6 and once again we connect them in our union data structure. At this point, there are no vertices that are not connected to one other vertex. We find that the next smallest edge is between 5 and 6. We connect the two in our data structure. Since 6 is the item that is connecting to 4, we call find on it and 2 and then make them both immediate children of 4.



In this final step we see that the next smallest edge is of weight 6 and lays between 2 and 5; however, since this would cause a cycle, we ignore it. We do the same thing for the edge weight of 7, our next choice, of weight 10 will not cause a cycle so we can go to it. We connect it in the Weighted Quick Union with Path Compression, and since 3 connects to the vertex 4, it calls find on itself and 1 which makes them both immediate children of 4.

The runtime for Kruskal's algorithm depends on what assumptions you make. If you make no assumptions, the runtime is $O(E\log(E))$. This is because you would need to make a priority queue of all the edges which takes $O(E\log(E)$ time and deleting the min will take $O(E\log(E))$ time. the sum of this is $O(E\log(E) + E\log(E))$If you make the assumption that the edges are already sorted, it will take $E\log(V)$ time because deleting the minimum will take constant time. However union and connecting in the Weighted Quick Union With Path Compression will take worst case $lg(V)$ time and this is done E times so the final result will be $E\log(V)$ time for presorted edges.

5.9 Conceptual Questions

1. Given an undirected graph, how can we determine if it is a tree?

2. True or False, given that we start from a set vertex, Prim's algorithm can create a unique Minimum Spanning Tree?

3. Explain why heuristics cannot be overestimated when we are performing A*.

4. In general, why does Dijkstra's Algorithm not work on graphs with negative edge weights?

5. What is the runtime of Kruskal's Algorithm if we are given edges in sorted order, and why is this the case?

6. If we are given our edges in presorted order, how long will it take to find the graph composed of the k shortest edges?

7. The transpose of a matrix is when the rows are interchanged with the columns. What is the result of the transposing an adjacency matrix of a directed graph? In an undirected graph?

5.10 Difficult Questions

1. Given a random vertex in a directed graph, determine if the graph is acyclic and is connected. The algorithm that you run should work in linear time, in accordance to vertices and edges.

2. Given 2 stacks how can we implement Breadth First Search?

3. If we are given a graph that has only edges of weight 1, 2, or 3, how can we modify it so that we can find the Minimum Spanning Tree in Linear Time in accordance to the amount of vertices and edge?

4. You are a TV addict and you enjoy watching television on long hot air balloon rides. You have a specific show that is N minutes long. Your hot air balloon has to start from port S and have a total of M ports. You want to see if you can have a balloon ride that is at least N minutes long that visits no port more than once so that you can finish your TV program while snacking. There is a bit of a problem- traveling between some ports u and v may cause your program to rewind- basically, you lose your time in the show. State the runtime of your algorithm as well as how you would solve this.

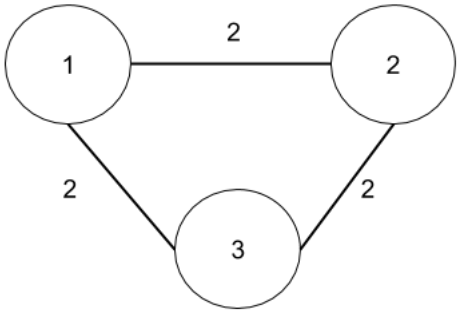
5.11 Chapter 5 Conceptual Question Solutions

1. There are two important components of a tree.

   - The graph must be fully connected
   - There must be no cycles

   To check full connectivity, we can run either Breadth First Search or Depth First Search and see if all of the vertices are reachable. This works because the graph is undirected. We can also use either one of the above traversals to see if there is a cycle.

2. False, take the following graph.



   If we start from vertex 2, we could get {[2,1]. [2,3]} or we could get {[2,1]. [1,3]}

3. The way A* works, we do not explore paths that have a higher cost than another one. If we overestimate a heuristic, we may be hiding a path that is shorter than the one that we are currently going through.

4. Dijkstra's does not generally work on graphs with negative edge weights because it can lower the weight of a path. When Dijkstra's relaxes an edge, it assumes there is no better way to get there. However, if there is a negative edge, there may have been a path that we ruled out that actually is shorter than our current path.

5. This would run in O(Elog(V)) time because we still need to make sure that when adding an edge we do not create a cycle. To do this, we can use a Weighted Quick Union with Path Compression.

6. This would take $\Theta(k)$ or O(E) time because we will just be adding the first k edges. Adding an edge is a linear operation and since we are not doing any checks for cycles we can just add with no worries.

7. In a directed graph, the transpose the adjacency matrix will result in the graph's edges being reversed. In an undirected graph no change will occur.

5.12 Chapter 5 Difficult Question Solutions

1. We can construct a new graph composed of the same graph with reversed edges. We run Depth First Search on this reversed graph from our given vertex and get to a vertex that has no children. We then run Depth First Search from this vertex in the original graph and see if all vertices are reachable and there are no cycles.

   We know this will work because the item in the reversed graph that has no children would be a root in our original graph. Finding the root takes O(V+E) time and checking if the graph is acyclic and connected takes O(V+E) time making our runtime O(V+E).

2. Breadth First Search traditionally uses a queue. Since we are given 2 stacks, we will make them act like a queue. We have 2 ways to do this, either we can make the push operation expensive or we can make the pop operation expensive.

   If we wanted to make the pop method expensive, we could do the following.

   (a) Check if the second stack is empty, if so move all elements from the first stack to the second.
   (b) Pop the first element from the second stack.

   To add items to our "queue" we would add the items to the first stack.

   If we wanted to make it so the push operation is expensive we can do the following method. Each time we add something to our "queue" we will push everything from our first stack to our second. We then push the element onto the first stack. We then pop everything from the second stack and push it onto the first stack. This ensures that the element we just added will be removed the last, the quality that we want in breadth first search. In this case, every time we wanted to remove an item we would just pop it directly from the first stack.

   We would use our version of a "queue" and then run normal breadth first search. The first method is a bit more optimal as we would only need to move the elements once in the pop operation whereas the second method moves the elements twice during the push operation.

3. This could be done by creating a new graph of dummy nodes. We would put 1 dummy node between nodes that have an edge of weight 2 between them and 2 dummy nodes between nodes that have an edge of weight 3 between them. We can then run Breadth First Search and get the correct answer.

   An important observation to be made is that in a graph with no weights, Breadth First Search finds the shortest path to any node. This means if we split our edges up so that ever edge is of weight 1 and run Breadth First Search, we will get the shortest path to all the nodes. The runtime of creating a new graph with all the vertices and dummy vertices is $O(V + E)$ and the runtime of running Breadth First Search on this graph is $O(V + E)$ getting us to a total running time of $O(V + E)$

4. In this problem, we run a modified version of Depth First Search while keeping track of the current distance from S as well as the current ports on our path. If the distance ever becomes greater than N, we have found a path that will let us finish our television program.

   At each step we check to see if the port we are at is already in our path. If so, we can simply ignore the edge, otherwise, we add the edge and add the edge weight to our sum so far. If, from a port v, we reach another port u, where no ports can be reached, we backtrack. We remove u from our path array and also subtract the edge weight of (u,v) from our sum so far. We keep backtracking until we can add an edge that was not previously on our path.

   The runtime of this algorithm is O(M!) as in the worst case the graph is complete and there is no

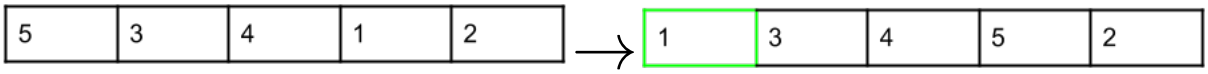
way to get over N minutes. S can initially go to M-1 ports. Then from each of these M-1 ports we can go to M-2 ports and from those M-2 to M-3. This process keeps continuing and yields a recurrence as follows M+M(M-1)+M(M-1)(M-2)+... . This is upper bounded by M(M(M-1)(M-2)...) which in turn is bounded by M!. Note that because we are checking paths of every single length, or the edges that reduce our point in the program, the negative edges weights do not matter.

Chapter 6

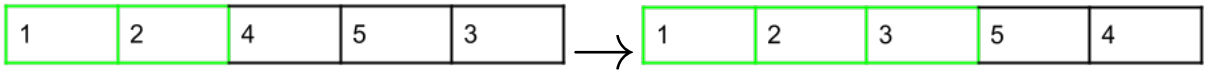
Sorting

6.1 Selection Sort

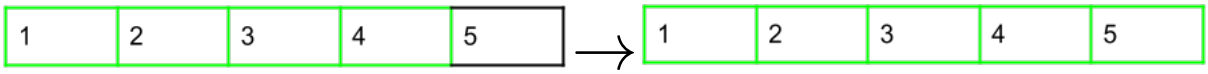
The first sort that we will go over is ***Selection Sort***. Selection sort, n the most basic terms, is finding the minimum element in the list and swaps it to the front to be in its proper places.



We start off with our unsorted array, then we go through all of the unsorted array and we look for the minimum element, which is 1. Once we find 1, we swap it with the 1st element in the array which is 5. We have now created 2 subarrays, one sorted and one unsorted.



We have now go and search for the tiniest element in the subarray which is 2. We then swap it with the 1st element in the subarray. We do a similar process for the element 3. We now have a sorted subarray of size 3 and an unsorted array of size 2.



We finish off by finding the minimum element between 4 and 5. We pick 4, put it in the ordered array which leaves us with just 5. After this, we are left with only 1 element, 5. We just add that element to our ordered array and we are finished.

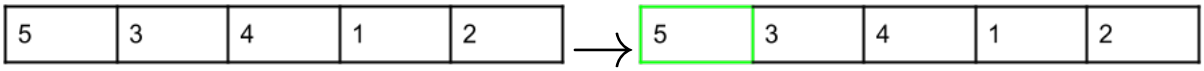
Let's analyze the runtime of this sort. For any given list, we will have 2 subarrays, one that is sorted and one that is unsorted. Each iteration of the sort, we find the smallest element in the unsorted array and then move it to the end of the sorted array. To do this, the first index of the unsorted array is swapped with smallest element. We always need to go through the full unordered array to find the minimum element, this leads us to do N work for the first iteration $N-1$ the second and so forth. We are left with the sum of $N+N-1+N-2....+3+2+1$ which can be rewritten as $1+2+3+...+N-1+N$ which is just N^2. This means that the runtime in all cases would be $\Theta(N^2$.

The space complexity of selection sort is just $\Theta(N)$ because all we need to do is manipulate pointers (keep track of the end of the sorted array/the start of the unsorted). We can do all our work in the same array so no extra space would be needed.

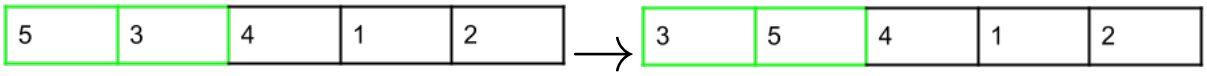
Selection sort is a stable sort because you look for the minimum each time. If the minimum happens to occur two times in a list simply pick the item that came first to come earlier in your sorted sub array.

6.2 Insertion Sort

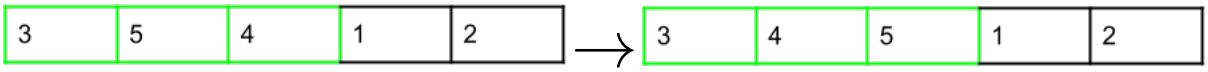
Insertion Sort is another relatively basic sort. It works as follows, you have 2 subarrays, similar to selection sort. Instead of looking for the minimum each time, you create the ordered array as you move through your unordered one. The process will be illustrated rather extensively in the following example:



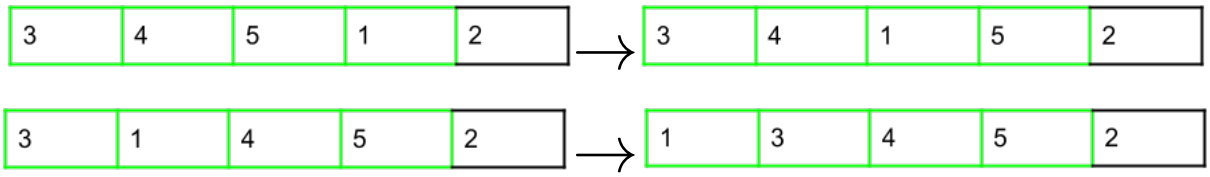
We start off with the unsorted array. We then create 2 subarrays, one sorted and one unsorted. The sorted array would be composed of the first element, 5.



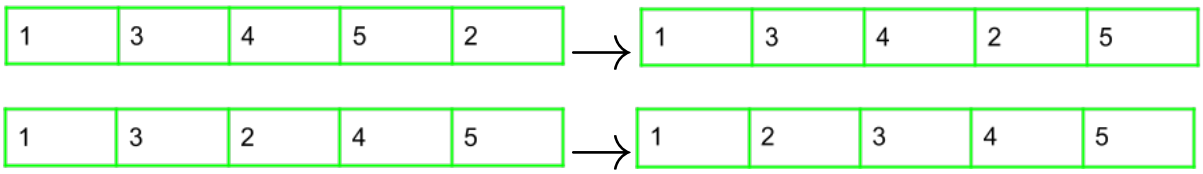
We have now expanded our sorted array to contain 3 and 5. The element 3 is less than 5, so we swap it with 5 in order to get proper ordering.



We will now expand the sorted array to include 4. We then swap 4 with 5 since it is less than 5. After this, we compare it to 3. 3 is less than 4, so we do not swap 4 with it.



We add 1 to the sorted array now. We have to swap it with 5 since 1 is less than 5. We swap it with 4 and then 3 because 1 is less than both of them. We now have 1 at the start of our array.



We now will put our
final element, 2, into our sorted array. We need to swap it with 5,4, and 3 in order to get it to its rightful spot.

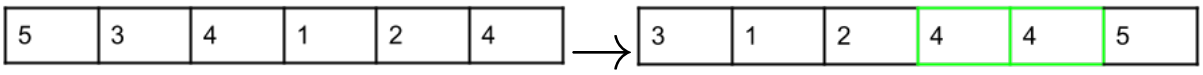
The amount of time that insertion sort takes directly relies on the amount of "inversions" that must be done. to get an idea of what exactly an inversion is, let's take a look at 1. 1 is 3 spots away from its rightful spot, its means that it adds 3 inversions to our total amount in our sort. This means that the runtime of insertion sort is $\Theta(N + K)$ where K is the amount of inversions. In the worst case, there are $\frac{N * (N-1)}{2}$ inversions.

No additional space must be added for insertion sort because everything can be done with pointers, similar to selection sort. Thus, the total amount of space that insertion sort uses is $\Theta(N)$

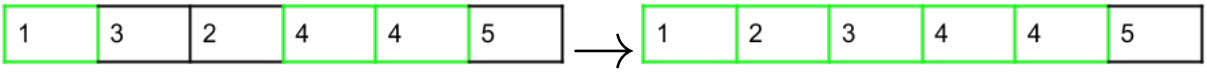
Insertion sort is a stable sort because anytime you find 2 elements that are equal, just do not swap them. That way the one that came earlier originally stays where it was.

6.3 Quicksort

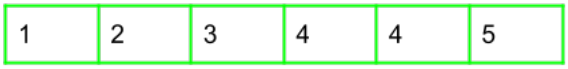
We will now discuss one of the fastest sorts ***Quicksort*** (who would've guessed this one would be fast). The idea of quicksort is that we choose some random pivot and create 3 subarrays. One subarray would contain all the elements less than the pivot (traditionally to the left), one subarray would contain elements equal to the pivot, and one subarray would contain elements greater than the pivot (usually to the right of the pivot). Here is an example of quicksort.



We started off with the unsorted array, we randomly choose a pivot of 4. We then move all the elements, in the order that they appear, to the left or right of the pivot based off their quantity. Note that our sorted array contains 2 elements. This is because there are 2 4's. From now on, the position of the 4's will not change.



We start off by going to the left unsorted array and choose 1 as a pivot. Nothing in its subarray is less than it, so we move both elements to its right. Following this, we choose 2 as a pivot and move 3 to the right of it. We no call quicksort on the last element 3 and recognize that there are no more elements, so quicksort has completed on this end.



Now that we have finished with the left side of the original pivot, we will go to the right side of the pivot. The only element on that side is 5. Thus we go to that element and then we are done with our quicksort.

The runtime for quicksort is $\Theta(Nlog(N)$ assuming we have a random pivot that is good (basically that we do not always end up with a pivot that has everything to the left of right of it).

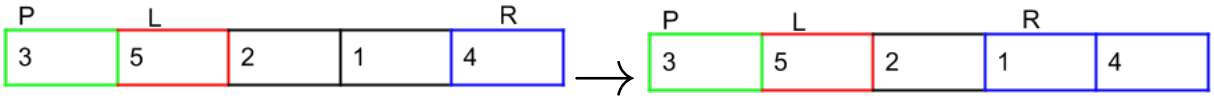
Quicksort will use, on average, $(log(N)$ extra space because there will be a call stack/recursive calls.

Quicksort is unstable because items will not keep their relative position to elements of the same type. The pivot is randomized, thus it is impossible for there to be stability.

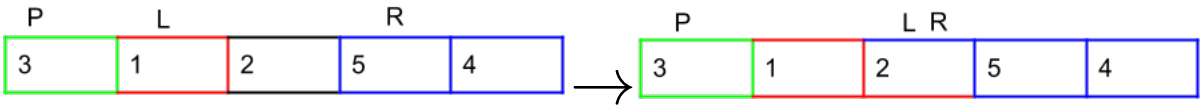
The previous way that we did quicksort was through making new arrays; however, there is a method that is almost always guaranteed to run in $\Theta(Nlog(N)$ time that is done in-place. This method is called **Hoare Partioning**. Hoare Partitioning works in the following way, you have a chosen pivot and 2 pointers, one that starts on the left end and one that starts on the right end.

The left pointer moves to the right until it finds an item that is greater than or equal to the pivot. The right pointer moves left until it finds an item that is less than or equal to the pivot. Once both pointers have stopped, the items that they are on swapped and the process repeats. This process terminates once

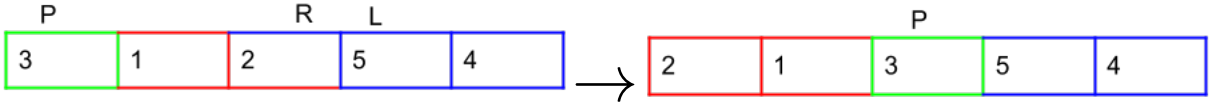
the pointers pass each other. We will illustrate this by going through 1 partition.



In this step, we started off with a pivot of 3 and we made our left pointer start on the element 5 and our right pointer start on the element 4. Immediately, the left pointer halts on the element 5 since 5 is greater than the pivot. The right pointer moves down to the element to the right of it.



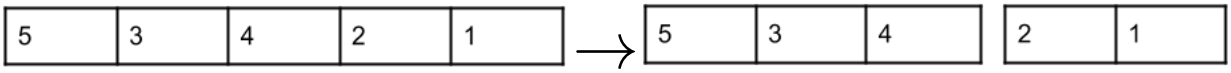
Since our right pointer was stopped on 1, because it is less than 3, we swap 1 and 5. After this, we move both our pointers to 2. At this point, the right pointer is halted since 2 is less than 3.



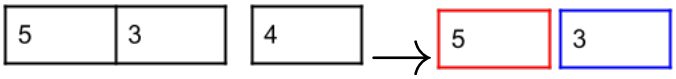
We move our left pointer one more item to the right. Here we realize that our pointers have crossed paths and our partitioning is done, well almost. As our very last step, we swap the item the right pointer is on with the pivot. Now we have a great in-place partition!

6.4 Mergesort

Mergesort is one of the most consistently fast algorithms. Mergesort works as follows: we divide the array in half, we perform mergesort on each half (recursion), we go through these steps until there is 1 element, then we move back up merging each sorted half. We will now perform a demo of Mergesort.



In this initial step, we start off with an unsorted array, we then start merge sort by splitting our array into 2 parts. Since our array is of length 5, we will make 1 side 1 element larger than the other.



We will now recurse on the left array. We will split up the elements once again so that we have 2 arrays, one of the elements { 5, 3 } and 1 that is {4}. We will once again move the the left array and call mergesort on it. We note that when we call merge sort, each item has only 1 element, so both these elements are ”sorted” in their own sub array.

In the first step listed, we ”merge” 5 and 3. Before we continue, we must call mergesort on 4. We note that 4 is a sole element and that it is sorted. We will then merge both these arrays and we get the sorted array {3, 4, 5 }

We have 2 sub arrays left, one unsorted and one sorted. We call merge sort on the right array and sort those 2 elements so now it is properly ordered. We finally have 2 sub arrays that are sorted.

We merge our 2 arrays and finally have our sorted array.

The runtime of mergesort is $\Theta(Nlog(N))$. We will utilize our knowledge of asymptotics to help us with this analysis. Each level does work proportional to the size of N (the size of the array/sub array). At each level, the array is split (the array is halved). In asymptotic notation, this looks like

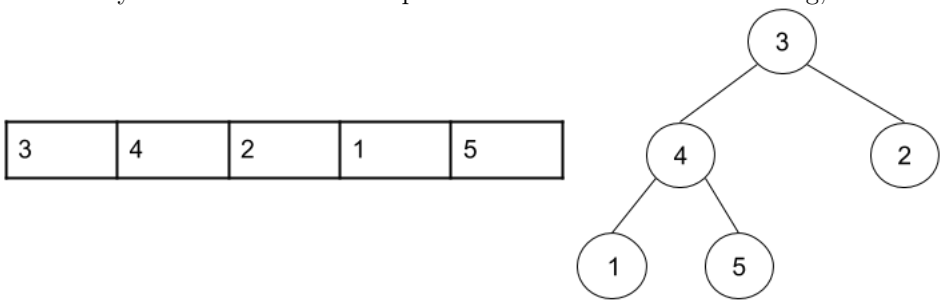
$$\sum_{i=0}^{log(n)} \frac{n}{2^i} \rightarrow \sum_{i=0}^{log(n)} n \rightarrow nlogn$$

The space complexity of merge sort is $O(N)$ because you would need to maintain a call stack for for your elements. However, once the call stack is used, you will be able to reuse it which prevents space complexity from being any worse.

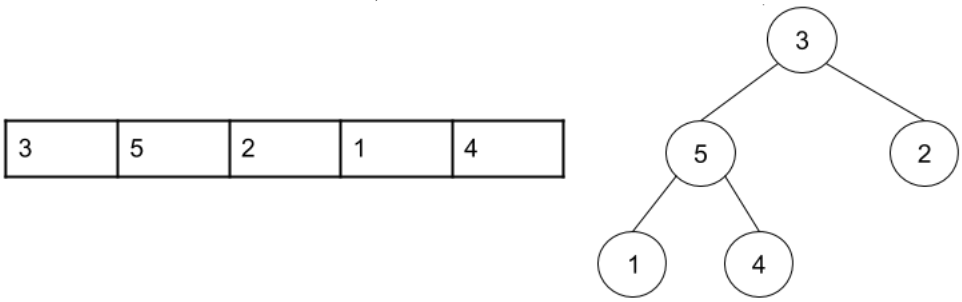
Mergesort is a stable sort because anytime we run into 2 elements that are the same, we would just choose the element in the left array to be put in first.

6.5 Heapsort

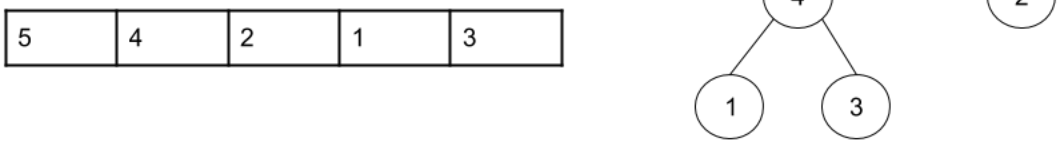
We learned about the heap data structure earlier in this book. We will now see how we can use this data structure in order to sort a list in a method called ***Heapsort***. We will create 2 subarrays within the array. One will be the heap and one will be the elements that are sorted. We will first "heapify" all our elements into a max heap then we will then perform n delete the max operations. We will put this item in the back of the array. Let's look at an example. Before we do the actual sorting, we will discuss heapifying.

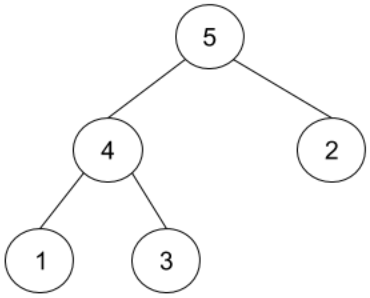


We will start off with this unsorted array. To the right of it is its corresponding heap. In order to make this into a valid heap, we will sink items in *array reverse level order*. This means that we will start at the lowest level and move to the left, then we will sink the item if needed.



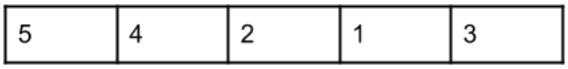
We cannot sink the first elements, 2, 1, or 5, but on the 4th element we check, 4, we realize it can be sinked since it is less than 5. We swap 4 and 5 and then we get our current heap.

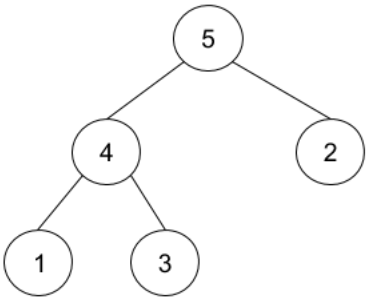




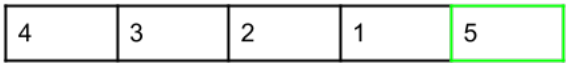
We move to the first index and then realize that we can swap 3 with 5, since 5 is bigger. We then see that 3 is less than 4 so we swap those elements as well.

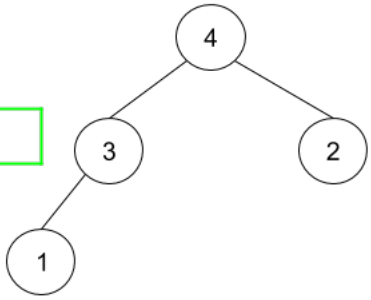
Now that we have our elements in the form of our max heap, we are ready to actually sort our array.



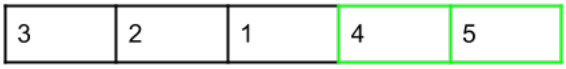


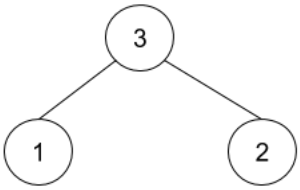
We will start off with the same heap that we did in the previous example.



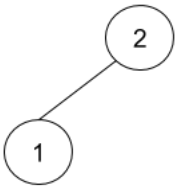


Now we have deleted our max element, 5. This results in 4 getting swapped to the top. The total amount of elements in our heap array has decreased by 1. We put the old max element in the index of the array that once has the last element.





We do a similar process with the next max element, 4. Once again, the size of the array goes down by 1. We put the item 4 where the old last element was.



We now delete the max once more and move 3 to the 3rd index, which has now been freed up. Our heap is now only 2 elements.

We repeat this process and delete the max element, 2, adjust our heap and put 2 in the 2nd index. We only have one more element left in our heap, 1.

After we delete the max one more time, our heap is completely empty. At this point, our array is totally sorted and our work is complete.

The time complexity of heapsort is $O(Nlog(N))$. As we know, constructing a heap takes $O(Nlog(N))$ time. We perform N delete the max operations, one for each element in the list. Each of these delete the max operations takes $log(N)$ time. As a result, our running time is $O(Nlog(N) + Nlog(N) = 2Nlog(N)$, simplified this is $O(Nlog(N)))$.

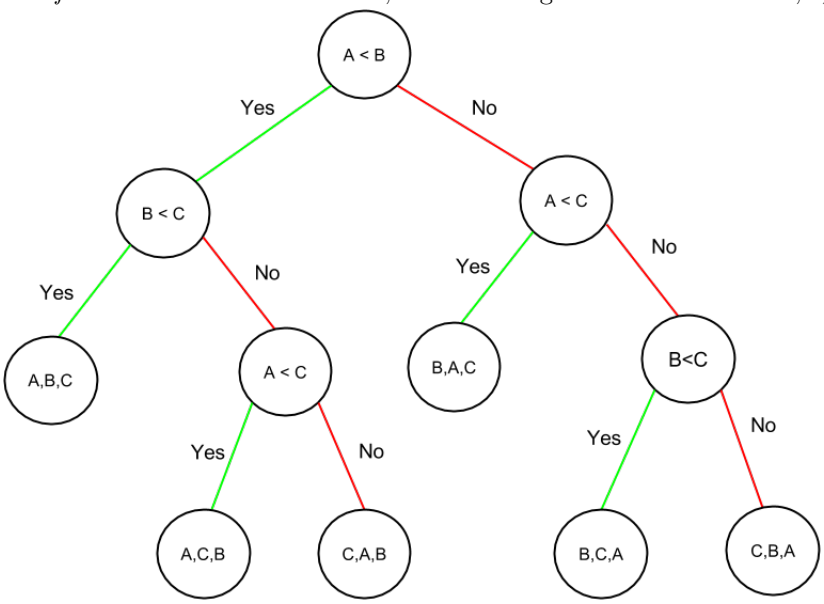
The space complexity of heapsort is $O(1)$ because everything occurs in the same array.

Heapsort is an unstable sort because when placing items into heaps, it is not guaranteed that the item in the correct order will swim or sink.

6.6 Radix Sort

Before we look at this next sort, let's take a small detour. The prior sorts we went over were comparison based sorts. These sorts have a lower bound of $O(Nlog(N))$. Why? We can visualize a decision tree for this mini proof.

Proof. In the below decision tree, we are seeing how we can order A,B, and C.

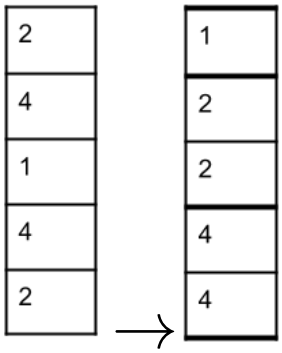


The amount of children that each node can have is equal to the amount of decisions we can make at that node. In our normal sorting, we can either say yes or no so we have 2 or 0 children per node. We know that nodes with 0 children are only those that have no more decisions to be made, which means they are fully sorted. The amount of ways we can order the lists is thus equal to the amount of nodes with 0 children, or leaves. In this case, we have 6, or 3!, ways to order our list. Since there are only 2 children per node, we can see that the height of our tree would be dependent on how many leaves we have; this leads us to $lg(6)$ in our case or, rounding up, 3 levels.

We can generalize this for all inputs. The amount of ways we can order any input would be $N!$ and the height of our tree would be upperbounded $lg(N!)$ (we will use $log(N)$ for ease of calculations). We can see that $log(N!) = log(N) + log(N-1) + log(N-2) \leq log(N) + log(N) + log(N) = Nlog(N)$. □

Sometimes, we may want something faster than $Nlog(N)$, but how can we get around this? Is there any way we can avoid "comparing" items? Well if there wasn't there this section of the book wouldn't be written- we can use a category of sort called ***Radix Sort*** to get around this lower bound.

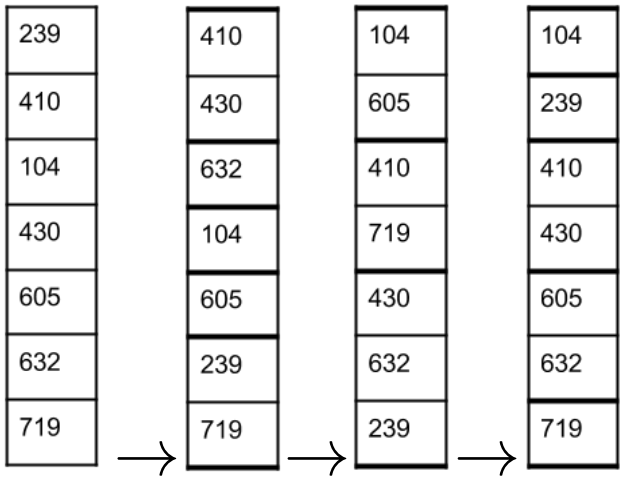
Instead of looking at the full item we are comparing, we look at a certain digit/index of all our elements and put them into "buckets". We make these buckets by assigning indices based off the amount of elements that are in the bucket. We will briefly go over an example.



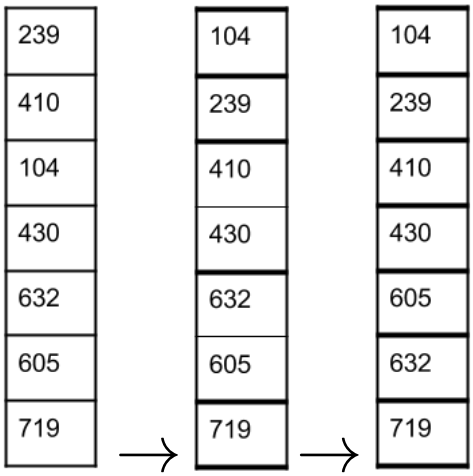
We start off with an unsorted array and then we end up with a sorted array. How is this done? We will start out by finding the amount of times each element appears in our unsorted array. We see that there is 1 1, 2 2's, and 2 4's. Using this information, we will calculate the starting index for each element. We know that 1 is the smallest element, so it will be the first set of buckets, which means that it starts at the 0 index. 2 is the next largest element, and since there is only 1 1, 2 will start at index 1. There are 2 elements in 2, so there will be 2's until index 2- this means that we should start our last bucket 4, at the index after, so 3.

The basic idea that we can use for this is that there you keep a starting index for a set of elements that, in our eyes, are equal. We then add our size to the bucket's index to find the end of it. The next bucket will be found by adding 1 to the end of the previous bucket.

There are 2 specific radix sorts that we will go over, ***LSD*** (Least Significant Digit) and ***MSD*** (Most Significant Digit). Least Significant Digit works as follows: we sort from right to left, at each step placing items in buckets that are ordered from least to greatest. We will go over an example:



The bolded lines refer to the "boundaries" of each digit in the process. The first list is unsorted, the second list is sorted by first digit {0, 2, 4, 5, 9 }, the third list is sorted by second digit {0,1,3}, the last list is sorted by final digit {1,2,4,6,7}. How do the items stay in order? After all, sorting with just 1 digit in mind does not make sense. To keep things as sorted as possible we go from top to bottom. For example, say we were on the second digit and we had 2 numbers, 239 and 231. Using the previous step, we would place 231 earlier in the list than 239 because 1 comes before 9. The general rule is that, within the same bucket, items that appeared earlier in a previous iteration will keep their same relative position.
MSD can be thought of as the reverse process as LSD sort. We start from the leftmost digit then we move to the right, making a new subbucket each time. Once every element is in its own element, the entire list is sorted.



We start off with an unsorted list then we move every item into buckets based off their highest digit. The way we order within buckets is based off relative position in the
list from the previous iteration. After 2 iterations, every item is in its own bucket so the algorithm terminates.

Now why would we use LSD over MSD and vice versa? MSD tends to be more useful because LSD must use every single digit. MSD on the other hand has the chance of terminating earlier.

In general, the worst case runtime for both LSD and MSD is $\Theta(WN + WR)$ where W is the size of the longest string, R is the size of the alphabet (for the creation of buckets) and N is the amount of strings. As we said earlier, MSD can also have terminate early, this means that it has a best case performance- this is $\Theta(N + R)$ in the case everything gets done in 1 iteration so the width of the key doesn't matter.
The runtime of radix sort does not come without cost. There is a much higher memory cost for radix sort- LSD uses $\Theta(N + R)$ space and MSD uses $\Theta(N + WR)$ the extra cost for MSD comes from the usage of collecting the buckets in the recursion of the algorithm.

Radix sort is a stable sort, in fact that is the whole base of how radix sorts (from top to bottom). If this property did not exist radix sort would return the wrong result.

6.7 Conceptual Questions

1. In what scenario would it be preferable to use Mergesort over Quicksort? When would it be better to use Quicksort over Mergesort?

2. Describe how Selection Sort and Heapsort are similar. Compare runtimes and how the algorithms actually function.

3. How could Quicksort be made stable?

6.8 Chapter 6 Conceptual Question Solutions

1. It is better to use Mergesort over Quicksort when we know that we will have a bad pivot for Quicksort. This is because if we have a bad pivot, we will approach our worst case runtime for Quicksort, $O(N^2)$. Additionally, we may want to use Mergesort over Quicksort when we need a stable sort, when we need to keep the relative ordering of items.

   We would want to use Quicksort over Mergesort in any time where we can be sure that we would not be approaching the worst case scenario. This is because even though Mergesort and Quicksort have the same asymptotic runtime, $O(Nlog(N))$, Quicksort has smaller constant factors making it better in real world scenarios.

2. Before the Kth iteration of Selection Sort, there are 2 subarrays, 1 of the minimum K-1 elements elements in sorted order and 1 of the other N-(K-1) elements in unsorted order. The algorithm would scan the unsorted subarray and find the minimum element and then place it at the end of the end of the sorted subarray. This operation would take O(N), for all N elements, this takes $O(N^2)$

   Heapsort follows a similar process. Before the Kth iteration of Heapsort there are 2 subarrays, 1 subarray of the maximujm K-1 elements in sorted order and 1 subarray of the array representation of Max Heap of the other N-(K-1) elements. The algorithm performs a removeMax operation and adds the element to the start of the sorted subarray. This operations takes $O(log(N))$ time and $O(NLog(N))$ time total.

   Essentially the only difference between Heapsort and Selection Sort is how they find the minimum at each iteration. This shows that Heapsort is an improved variation of Selection Sort.

3. Quicksort could be made stable if one makes a pass to go through the array given array and note the relative orderings of all the elements. This can be done by storing a separate array that corresponds to the indices of the actual elements. When an element swaps in the original array swap it in the array of indices. If two items are equal, compare their indices and put the one that comes later originally to the right of the element that comes earlier. The reason why this tends not to be done in practice is because it takes $O(N)$ additional space.

Chapter 7

Extra Topics

7.1 Introduction

This section delves into some more advanced topics of Computer Science at a relatively superficial level. These topics will seem slightly irrelevant to topics we have been going over the rest of this book, and they are. If you are reading this for Computer Science 61B you probably do not need to worry about the test going as in depth as this textbook does for these sections. For any other class, sorry I have no clue what needs to be done for you.

7.2 Huffman Encoding

Eh eventually this will be gotten to

7.3 The World of NP

A relatively large emphasis of this textbook was both analyzing runtime and trying to make solutions that fit under some upperbound on time complexity. We have explored problems that we know have some polynomial time algorithm- that is they are upperbounded by some function $O(c^x)$ where c is some constant and x is some variable. However, there a a great deal of problems that cannot be solved in polynomial time and instead need to be solved in exponential time. So we can't always find a solution efficiently but it would still be useful to see if a proposed solution solves our problem. We will use those 2 problems: verification of a solution efficiently and finding a solution efficiently to define a few terms. Problems referring to seeing if our solution exists, or questions with a yes or no answer, will be referred to as a **Decision Problems**.

1. **P :** The set of Decision Problems that can be solved in polynomial time. An example of this is finding the Minimum Spanning Tree of a graph because there is a known algorithm (Prim's or Kruskal's) that solves this problem in polynomial time, specifically $O(Elog(E))$

2. **NP :** The set of all Decision Problems which can be verified in polynomial time. Basically given an instance of some proposed solution, we can check to see if it is a solution to our original problem. This category contains all problems in P because all problems that can be solved in polynomial time can be verified in polynomial time.

3. **NP Hard :** Problems which are at least as hard as the hardest problem in NP. This definition is a bit hand wavy for now, but we will go more in depth in the next section.

4. **NP-Complete :** Problems that are in NP but cannot be solved in polynomial time. This means they can be verified in polynomial time but the solution cannot be found in anything less than exponential time. This definition, just like NP Hard, will be solidified in the later section.

7.4 Reductions

Let's take a look at **3SAT**. 3SAT is a boolean formula with a specific form. Any instance of 3SAT would have a series of groupings, where each grouping has at most 3 **literals**. A literal is a variable x_1 or its negation, $\neg x_1$. Each grouping in 3SAT is called a **clause** and contains at most 3 literals that have an or relationship- that is if any of the literals evaluates to true, the entire expression evaluates to true. An example of this a clause follows: $(x_1 \vee x_2 \vee \neg x_3)$ where "$\vee$" means "or"/"||" and "$\neg$" means "not". In 3SAT we deal with a series of clauses that all have an and relationship between them such as the following example: $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_4 \vee x_5 \vee \neg x_1) \wedge ...$ where "$\wedge$" means "and"/"&&"

Our job is to find a satisfying assignment, what we can set each variable to such that there are no contradictions in logic and the entire statement evaluates to true.

3SAT is given to us as a problem that is in NP-Complete, and using a technique called reductions we can prove that other problems are in NP-Complete as well.

Now let's look at another problem, **Clique**. Clique is a problem where, given a goal g and and a graph G, we want to find a subset of g vertices in G such that all the edges exist between these vertices. Think of it like a complete graph for a subset of g vertices. We will prove this is problem is NP-Complete.

To prove a problem is NP-Complete, we must prove 2 things:

1. We can verify if a proposed solution is correct in polynomial time. This follows as the definition of NP.
2. If we can solve this problem, we can solve any problem that is NP. To do this

If we have some proposed solution for Clique we need to see that there are edges between all the vertices in the solution. This is trivially verifiable, just go to each vertex in your proposed solution and make sure that in your adjacency list/matrix, there exists an edge to every other vertex in your solution.

Now comes a bit of a trickier part, we want to prove that we can solve some NP-Complete problem. The process of doing this is called a **reduction**. Call the original NP-complete problem A and the problem that we are trying to prove as NP-Complete is B. Basically what we will do in this process is transform, some instance of A, called I, into an instance of B, we will call this instance I'. We then claim we have a solver for B and when we run it with an argument of A'. The solution to this would be G'- we then transform G' to a solution of A, call the solution of A as G. For a reduction to be valid the following must be satisfied:

1. The transformation between instances of problems and solutions must be shown to occur in polynomial time.
2. If there exists a solution to I' in B then there exists a solution for A
3. If no solution exists to I' in B then there is no solution for I in A. This can be a bit complicated to prove so we can use the contrapositive- prove that if there is I is a solution to A then I' is a solution to B.

For this particular case, we only have one known NP-Complete problem, 3SAT, so we will transform an instance of 3SAT into Clique. Now reductions usually take some ingenuity and practice, so don't be too intimidated if it seems a bit crazy. There are 2 qualities about 3SAT that we will use for this reduction.

1. 3SAT is a set of clauses each with a maximum of 3 literals ad that in any clause only 1 literal has to be true for the clause to evaluate to true.
2. If x_1 evaluates to true then $\neg x_1$ cannot be true as that would be a contradiction.

Now to make an input to 3SAT look like an input for Clique we will do the following:

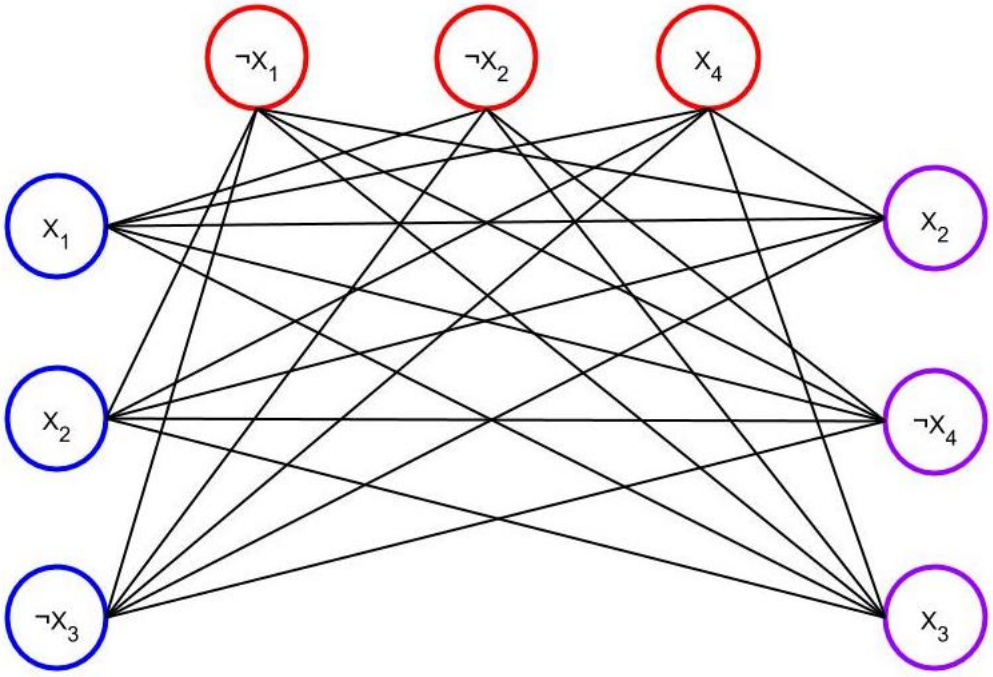
1. We will separate every clause into vertices, with 1 vertex corresponding to one literal in the clause.

2. Add an edge between vertices in different clauses that do not contradict. In other words add edges between every vertex except for vertices that are
   (a) In the same clause.
   (b) Contradict, for example there cannot be an edge between any vertex for x_1 and any other vertex for $\neg x_1$.

For example, we would transform the 3SAT input:

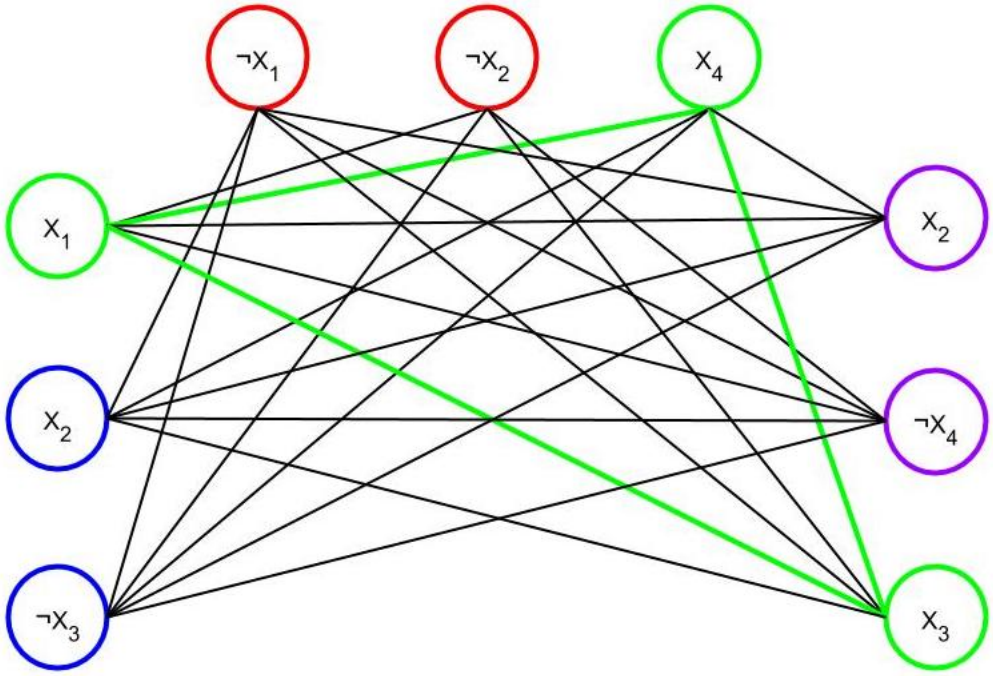
$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_4 \vee x_3)$$

into



The left column corresponds to the first clause, the middle row on the top corresponds to the second clause and the right column corresponds to the third clause. Note how there are no edges between vertices that correspond to literals in the same clause, for example there are no edges between any vertex in the left column. Also there are edges between vertices that correspond to literals that are the same in different clauses but there are no edges between literals and their negation. For example, there is an edge between x_2 in the left column and x_2 in the right column; however, there is not an edge from either of them to $\neg x_2$ in the upper middle row. The construction of this graph will be polynomial in the amount of clauses and literals.

Now that we have constructed a graph that corresponds to 3SAT, we will then run our Clique solver on this graph with our goal being the amount of clauses, in this case 3. Our solver then returns 3 vertices, a plausible solution is below.



A valid clique is one that contains the vertices x_1 from the left column, x_4 from the middle row, and x_3 from the right column. This corresponds to the $x_1, x_3, and x_4$ being set to true in our 3SAT instance.

We will prove the validity of this reduction now. For this explanation take k to be the amount of clauses in I.

1. As stated earlier, transforming the 3SAT instance into an instance for Clique is polynomial in the amount of clauses and literals. Transforming the solution is simply getting the vertices in the Clique, so it is polynomial in the vertices in the Clique solution.

2. If there is a solution to I' for Clique then there is a solution to I in 3SAT. Our I' asks for a solution of size equal to k. Since literals are only connected if they are not part of the same clause, our clique will only have one literal per clause. We can then say if we have a Clique of size k, then we can set all the literals in the Clique to be true and then each one of our 3SAT clauses would be satisfied, making the overall 3SAT satisfied.

3. We will no prove if I has a solution in 3SAT then there is a solution to I' in Clique. From the satisfying assignment for 3SAT create a set S that contains exactly 1 element that is true from each clause. Then in our instance I', we can have the vertices corresponding to the elements in S be the vertices in the Clique. The Clique will be exactly size k because every literal chosen are from different clauses.

We have now proven that Clique is in NP-Complete, but let's discuss a little bit about the difficulty. Since 3SAT can reduce to Clique that means that we can solve 3SAT in the amount of time it takes to solve Clique with some extra polynomial factor of transforming the inputs and outputs. We can generalize this by saying **if A reduces to B then A is no harder than B**. Assume that this is not the case, then if A was harder than B there would be a contradiction as we could solve B in time better to solve A; however, when we solve B we also can get a solution for A. Additionally, we can say **if A reduces to B then B is at least as hard as A**. If we are given A's difficulty, then if B can solve A, it must take at least as much time as A does otherwise there would be a contradiction.

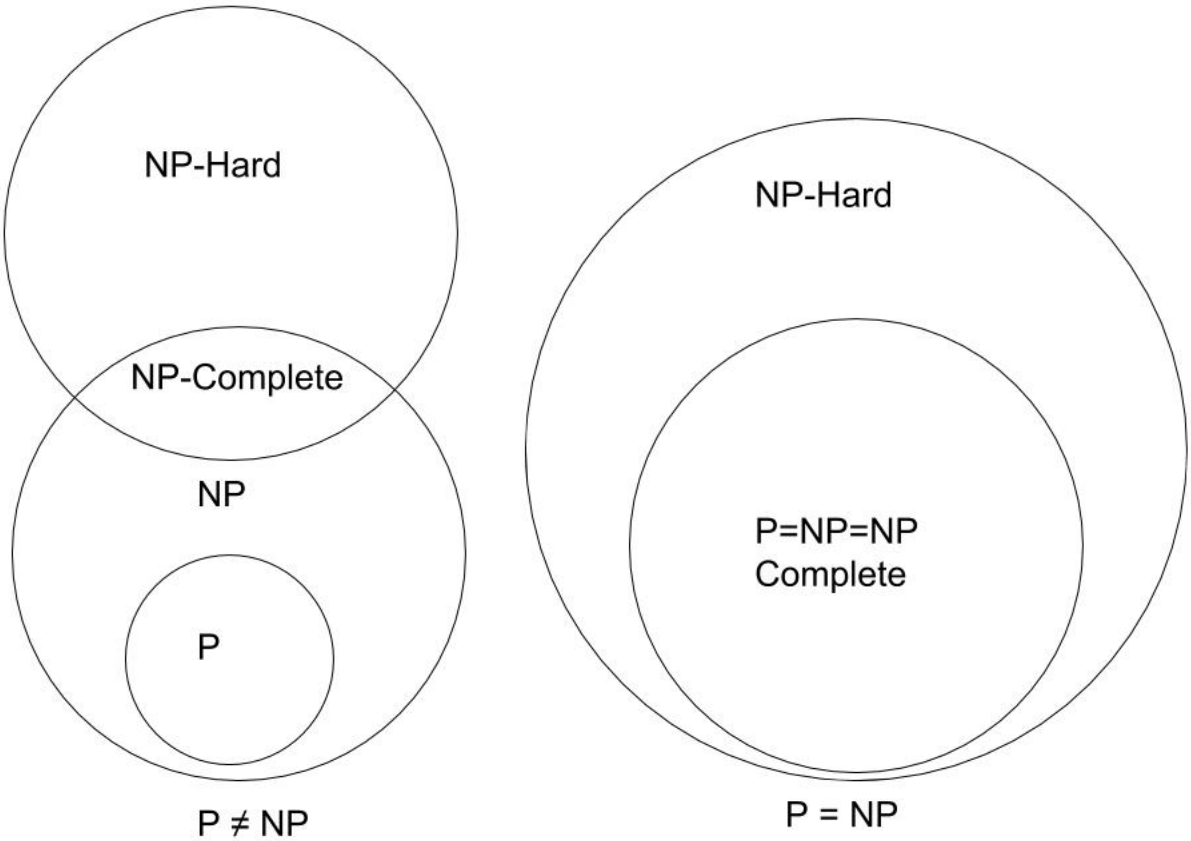
We will now use our newfound knowledge of reductions to clarify the terms of NP-Complete and NP Hard.

1. A problem A is NP Hard if every problem in NP can reduce to it. You can get an instance of an NP problem, transform it to an instance of the problem A then solve A and transform it to a solution for the solution of the NP problem.

2. A problem is NP-Complete if it is in NP, that is that you can see if a proposed solution is a solution to the problem and it is NP Hard.

7.5 Consequences of P and NP

Say we could show that we could reduce Clique to Minimum Spanning Tree. That would be a bit strange because we proved that Clique is NP-Complete and we showed earlier that Minimum Spanning Tree is a P problem. As we said earlier, if A reduces to B then A is at most as hard as B. We just showed we could make an NP-Complete problem at most as difficult as a P problem. In other words, we proven that P = NP. If P = NP then we can transform any NP problem into a P problem by a series of polynomial reductions. Below, on the left there is an image that shows the complexity distribution of problems if $P \neq NP$ and on the right there is an image that shows the complexity distribution if $P = NP$



7.6 Chapter 7 Conceptual Questions

1. Do all NP-Complete problems have the same asymptotic complexity? Explain your reasoning.

7.7 Chapter 7 Difficult Questions

7.8 Chapter 7 Conceptual Question Solutions

1. Yes all NP-Complete problems run in exponential time. We also know that we can reduce all NP-Complete problems to each other in some polynomial transformation. Polynomial terms are automatically lower order than the exponential ones so we can disregard the transformation asymptotically. Then all we really care about is the running time of the algorithm. However for any reduction from A to B, A is no harder than B. Anytime we reduce A to B we can also reduce B to A because both are guaranteed to be NP-Complete. Thus they have the same asymptotic running time.

Chapter 8

Cheat Sheets

8.1 Introduction

This section serves as cheat sheet worthy material. This does not serve as a replacement for either this textbook or the course information as a whole. Rather, it just serves as a quick fact sheet.

You will find runtimes for operations for data structures (best, worst, and average case) as well as runtimes for the various sorting algorithms. You will also be given basic descriptions about each data structure and algorithm. This description will include how to spot certain algorithms and when to use certain data structures.

If you need to review anything about Data Structures revisit chapter 4. If you need to review anything about Sorting revisit chapter 6.

8.2 Data Structure Cheat Sheet

Name	Description	Runtime of operations
Map	Similar to a dictionary from Python. There are Key Value pairs. If a key that already exists is inserted, the value of the existing key is replaced. Syntax is Map<Key,Value> Use when keeping count of how many times something occurs use a map.	Depends on implementation
Set	Used for unordered data. There can be no duplicates in a set. There are only keys (maps with dummy values) Use Sets when you need to just store elements	Depends on implementation
LinkedList	Can be used when given sorted data. Though is not always most efficient way to store.	get = O(N) insert = O(1) remove = O(N)
Arraylist	Can be used when given sorted data. Though is not always most efficient way to store.	get = O(1) insert = O(N) remove = O(N)
Queue	First in First Out Data Structure. Use when you want to take out least recent element.	enqueue = O(1) dequeue = O(1)
Stack	First in Last Out Data Structure. Use when you want to take out most recent element.	pop = O(1) push = O(1)
Weighted Quick Union	Used whenever you need to check if 2 things have some sort of relationship.	union(w/o p/c) = O(lg(N)) find(w/o p/c) = (O(lg(N)) union (w/pc) = Almost O(1) amortized find (w/pc) = Almost O(1) amortized
Tree	All keys have to implement comparable.	Best Case O(1) for all operations Average Case O(log(N)) for all operations Worst Case O(N) for all operations
2-3 Tree/Red Black Tree	Used when you need to use some sort of order. Any key has to implement comparable. Use over hashing when the key is expensive to hash.	find, remove, insert = O(log(N))
Hashtable	Use when you need really fast runtimes assuming a good hash. Unordered Data Structure	Amortized O(1) for all operations Average O(N/M) = O(L) for all operations Worst O(N) for all operations
Priority Queue	Use whenever you need the most or least of something. Can be represented as a heap. Not for overall order, just the least/greatest. Items must be comparable	add = O(log(N)) getLargest/Smallest = O(1) removeLargest/Smallest= O(log(N))
Trie	Use whenever you want to check something about objects with a certain prefix.	O(M) for insert in normal Trie O(Nlog(N)) for Ternary Search Trie

8.3 Sorting Cheat Sheet

Name	Description	Best Case Running Time	Worst Case Running Time	Average Running Time	Space	Stable?	How to Spot?
Selection Sort	Find the minimum element and swap it with the front. Move front pointer 1 index right.	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$	Yes	Minimum elements are the one in front.
Insertion Sort	Sort the list as you move down. You have 2 subarrays, one that is sorted, and one yet to be sorted.	$\Theta(N)$ When there are no inversions	$\Theta(N^2)$	$\Theta(N+K)$ K is the number of inversions	$\Theta(1)$	Yes	If the first x elements are sorted.
Heapsort	Make a heap by either: 1) Inserting into a new heap 2) in place heapify: Sink in reverse level order/Swim in level order. Remove max n times until you reach the end of the array.	$\Theta(N)$ When all the elements are the same	$\Theta(N\log(N))$	$\Theta(N\log(N))$	$\Theta(N)$ for first way $\Theta(1)$ for in place.	No	If the minimum elements are at the end and the largest elements are in the front.
Mergesort	Group elements by 2 then sort each group until it is fully merged.	$\Theta(N\log(N))$	$\Theta(N\log(N))$	$\Theta(N\log(N))$	$\Theta(N)$ total sub arrays	Yes	Look for multiple defined sorted regions.
Quicksort	Choose a pivot then move all items less than the pivot to 1 subarray, all greater to another then all equal to another. Repeat.	$\Theta(N)$ All the elements are the same	$\Theta(N^2)$ Always pick the greatest or least element	$\Theta(N\log(N))$	$\Theta(\log N)$ for in place call stack $\Theta(N)$ not in place	No in most cases	Look for pivots that stay in the same spot.
Counting Sort	Find the frequencies of each element in a list and create starting pointers. Insert into a list.	$\Theta(N+R)$ R: number of unique items	$\Theta(N+R)$	$\Theta(N+R)$	$\Theta(N+R)$	Yes	Don't really need to
Least Significant Digit Sort	Counting sort by digit from right to left (least significant to most). Prefered over Most Significant Digit Sort when given small widths.	$\Theta(W(N+R))$ R: length of alphabet W- Width of word	$\Theta(W(N+R))$	$\Theta(W(N+R))$	$\Theta(N+R)$	Yes	Look if digits are sorted and if least significant digits are sorted by buckets
Most Significant Digit Sort	Counting sort by digit from left to right (most significant digit to least). Prefered over Least Significant Digit Sort when large numbers.	$\Theta(N+R)$ If all elements are in the same bucket	$\Theta(W(N+R))$	$\Theta(W(N+R))$	$\Theta(N+WR)$ W is possible buckets for digits.	Yes	Look if digit's are sorted and if the most significant digits are sorted by bucket.

Chapter 9

Practice Tests

Practice Midterm 1

This test has 9 questions worth a total of 100 points, and is to be completed in 110 minutes. The exam is closed book, except that you are allowed to use one double sided written cheat sheet (front and back). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. Write the statement out below in the blank provided and sign. You may do this before the exam begins.

#	Points	#	Points
1	4.5	6	20
2	7	7	8
3	10	8	0
4	7	9	10
5	8	10	25
		Total	100

1. (Vitamin) C what's going on?: (4.5 pts) Step through the running of the following program and at each blank write the values of o1.x[0], o1.x[1], o2.x[0], and o2.x[1]. Assume that we start off with the constructor being called.

```
public class OJ{
  int[] x;
  OJ z;
  OJ(int x, int y){
     this.x = new int[2];
     this.x[0] = x;
     this.x[1] = y;
  }
}
public class Juice {
  public OJ o1;
  public static OJ o2;

  Juice() {
     o1 = new OJ(1, 2);
     o1.z = new OJ(5, 6);
     o2 = new OJ(3, 4);
     o2.z = new OJ(7, 8);
     pulpify();
     vitaminSeed();
     appleImposter();
  }

  public void pulpify() {
     o1.x[1] = o2.x[1];   _____
  }

  public void vitaminSeed() {
     o1.x[0] = o1.z.x[0]; _____

     o2.x[0] = o2.z.x[1]; _____
     o1.z = o2;
  }

  public void appleImposter() {
     o1.x[1] = o2.x[0];
     o2.x[0] = o1.x[1];
     o2.x[1] = o1.z.x[0]; _____
  }
}
```

2. Errrrr…..er: (7 pts) Find all the compilation errors and mark them with a "C". Find all runtime errors and mark them with a "R". Note, if a line relies on something that has errored out, you can assume that the prior error was fixed. Assume we start off by calling the constructor. For every error give a brief explanation why it errors out.

```
public class CorR {
  public static final int[] arr = new int[10];
  int i;
  XD xd;

  private class XD {
     private int val;

     XD(int x) {
        val = x;
     }
  }

  CorR() {
     i = 5;
     xd = new XD(i);
     diggity();
     dawg();
     coolCat();}

  private static void diggity() {
     arr[0] = 10;
     xd.val = 0;
     arr[2] = 15;}

  private void dawg() {
     for (int i = 0; i < arr.length; i++) {
        arr[i] = i * 2 + 1 - 10 + .5;
     }
     diggity();
     new int[] temp = new Integer[10];
     arr = temp;}

  static void coolCat() {
     i++;
     dawg();
     arr[2] = arr[1] + 5;
     xd = new XD(10);
     int[] temp = ((int[]) new double[10]);
     temp = arr;}}
```

3. Casts and Inheriting Broken Bones: (10 pts) Below is a set of classes. We will have a series of method calls. In the lines following each method call, write what is printed, if anything at all. If there is a compilation or runtime error please say which one it is and provide an explanation. You may assume that previous lines affect the following.

```
public class Container {
  int size; boolean haslid; String name;}
  public Container(){
     size = 0;
     haslid = false;
     name = "bad container";
     System.out.println("no constructor");}

  public Container(int size, boolean liddy, String name){
     this.size = size;
     this.haslid = liddy;
     this.name = name;
     System.out.println("here it is");}

  public void open() {
     System.out.println("there");}

  public void close() {
     if (!haslid) {
        System.out.println("Can't close what isn't there");}
    else{
          System.out.println("Closed");
}}}
public class NutellaJar extends Container {
            int sweetness;
            public NutellaJar() {
               System.out.println("I'm so hungry");
               this.sweetness = 100;}

            public NutellaJar(int size, boolean lid, String name, int sweetness) {
               super(size, lid, name);
               System.out.println("oink");
               this.sweetness = sweetness;}

            public void taste() {
               System.out.println("Just one more scoop");}

            public void taste(int scoops) {
               System.out.println("I just ate" + scoops + ". yum");}

            public void close() {
               System.out.println("This is too hard!");}}
```

Container plainjar = new Container(); _____ _____ _____

plainjar.close(); _____ _____ _____

Container tasty = new NutellaJar(); _____ _____ _____

tasty.taste(); _____ _____ _____

tasty.close(); _____ _____ _____

NutellaJar scrumptious = new Container(); _____ _____ _____

NutellaJar nutty = (NutellaJar) plainjar; _____ _____ _____

NutellaJar n = new NutellaJar(5, true, “sweet thang”, 10); _____ _____

scrumptious.close() _____ _____ _____

nutty.close(); _____ _____ _____

((NutellaJar) tasty).taste(10); _____ _____ _____

nutty.taste(); _____ _____ _____

4) It's always my de-Fault: (7 pts) Use the below interfaces to create a class that implements Viking and compiles.

```
public interface Norse {
   final static int burliness = 100;
   void breathe();
   void grunt();
}
public interface Viking extends Norse{
   final static int burliness = 300;
   void attack();
   boolean fly();
   void grunt();
   default void grunt(String t){
      System.out.println(t + "ARRRRRGH");
   }

}
```

5. Osmosis: (8 pts) We want to add a method to IntList so that if 2 numbers in a row are the same, we add them together and make one large node. For example:
1 ⇒ 1 ⇒ 2 ==>3 becomes 2 ⇒ 2 ⇒ 3 which becomes 4 ⇒ 3.
For this problem, you will not have access to any add, size, or remove method.

```
public class IntList {
  public int first;
  public IntList rest;

  public IntList(int f, IntList r) {
     this.first = f;
     this.rest = r;
  }

  public void addAdjacent() {

    _____________________________________________________________________

     if (_____________________) {

     _____________________________________________________________________
     }
     IntList s = p;
     while (_____________________) {

       _____________________________________________________________________

       _____________________________________________________________________

       _____________________________________________________________________

       _____________________________________________________________________

       _____________________________________________________________________

       _____________________________________________________________________

       _____________________________________________________________________

       _____________________________________________________________________

       _____________________________________________________________________
       }
     }
  }
}
```

6. Interfering Interfaces: (20 points) We want to write two classes, Sandwich and Pizza. Both of these classes will implement the interface ToppableFoods. A quality of all ToppableFoods is that you can add all the toppings you want that anything that extends the Ingredient class.

Two specific classes that extend the Ingredient class are Topping and Saucy.

The first item added to a sandwich must be a slice of bread and you can add anything up until you add a second slice of bread (sorry no triple deckers). For Pizzas' the first item added must be "Dough", any amount of ingredients may be added.

Regarding Sauces, Sandwiches can add any amount of sauces whenever you want. On the other hand, pizzas can have a maximum of 2 sauces added BEFORE toppings are added (you can also go sauceless like a savage).

Just like we can add toppings and sauces, we can also remove them. In both cases, only the most recently added item is allowed to be removed. Any attempt to remove any other item should result in an IllegalArgumentException. You can remove ingredients until there are no more left.

- Note for the purpose of this problem, Bread and Dough are considered Toppings.
- You may use a LinkedListDeque or ArrayList Deque
- You may find instanceOf useful
    - Implementation is as follows: <Object> instanceOf <Class name>
    - Returns a boolean value (true if it is an instance of that class, false otherwise).
- All methods that will be common to both Sandwich and Pizza should be in ToppableFoods
- You will have four pages total to write the Pizza class and the Sandwich class (2 for each). You may include any helper classes and instance variables.

Below we have implemented the Ingredient class, the Topping class, the Saucy class,an example implementation of a class that implements Topping, and the Deque interface.

```
public class Ingredient{
String name;
Ingredient(String
name){
this.name = name;}}
```

```
public class Saucy extends
Ingredient{
    Saucy(String name){
     super(name);
}}
public class Topping
extends Ingredient{
     Topping(String name){
     super(name);
}}
```

```
public class Bread
extends Topping{
  int grain;
   Bread(int grain){
   super("Bread");
   this.grain = grain;
}
```

```
interface
Deque<Item> {
void addFirst(Item x);
 void addLast(Item x);
boolean isEmpty();
int size();
void printDeque();
Item get(int index);
Item removeFirst();
Item removeLast();}
```

public interface ToppableFoods

public class Sandwich _____ {

public class Pizza _____ {

7. Troe & False: (8 pts)
a)General Colonel wants to make a method that both overloads and overwrites the method of a parent class. Is this possible? Explain your answer.

b) When would you use a comparable over a comparator? Which one is preferable?

c) An instance of a class has a broader scope than just the class. That is, calling a method or variable from a class may cause a compilation error, but an instance of it won't.

d) Does overloading a method take into account the return value?
Basically would changing public int hello(int hi){...} to public boolean hello(int hi){...} be valid? Explain why.

8) Riddle me this (0 pts): What weighs six ounces, sits in a tree, and is very dangerous?

9) A Test Within a Test?: (10 pts)

a) Corn on the Cobb wants to is attempting to use an IntList to get through dreams. At any given time, the IntList's size must be a maximum of 5. If the IntList's size ever seems like it is going to exceed 5, you must remove the first node, making the second element the new first. For example adding 5 to the following list 0 ⇒ 1 ⇒ 2 ⇒ 3 ⇒ 4 would yield 1 ⇒ 2 ⇒ 3 ⇒ 4 ⇒ 5. Write dreaming which adds nodes and makes sure that the IntList fulfills the requirements that we put on it above. (8 pts)

```
public class IntList{
   public int first;
   public IntList rest;
   public int size(){...}
}
public void dreaming(int n){

        if(_____________________________________){

        }
        else{
             _____________________________________________________________
             _____________________________________________________________
             _____________________________________________________________
             _____________________________________________________________
              ____________________________________________________________
             _____________________________________________________________
             _____________________________________________________________
             _____________________________________________________________
             _____________________________________________________________
             _____________________________________________________________
             _____________________________________________________________
        }
}
```

b) Corn on the Cobb now wants to test this code to make sure that he does not lose his way in his dreams (that would really suck). Write a basic JUnit test to make sure that your code works as expected. Note: The IntList.list(1, 2, 3 , 4, 5) would make an int list 1 ⇒ 2 ⇒ 3 ⇒ 4 ⇒ 5 (3 pts)

```
@Test
public testDreaming(){
             _____________________________________________________________
             _____________________________________________________________
             _____________________________________________________________
              ____________________________________________________________
             _____________________________________________________________
             _____________________________________________________________
             _____________________________________________________________
             _____________________________________________________________
             _____________________________________________________________
             _____________________________________________________________
        }
```

10) Arrrrghrays: (25 pts) Purplebeard and his lackey Turquoisenail are sailing the 10 seas. In order to sail well, they want to be able to create a map. They managed to create their map, but Turquoisenail tripped and put it through a paper shredder. They managed to store the scrap images into a 1d array, but they need to make it into a 10x10 map. You are lucky because on each piece you have the longitude and latitude written down. Write a short program to help put the pieces back together. The pieces should be as follows

-100, 30	-50, 30	-25, 30
-100, 20	-50, 20	-25, 20
-100, 10	-50, 10	-25, 110

In the upper left corner, -100 is the longitude and 30 is the latitude.

For this problem, you have access to IntLists, ArrayDeques, arrays, and LinkedListDeques

Note: This problem is very hard, and it is expected that you attempted earlier problems before looking at this one.

a) (2 pts) For the first part of this problem, make a Piece class that store longitude and latitude. (**2 pts)**
public class Piece{

b) (15 pts) For the second part of this, make a method that takes in an array of all the pieces and returns a 2d array that is sorted by latitude. You may use both pages to write the problem.

```
public Piece[][] sortByLat(Piece[] p) {
```

c) (8 pts) The final part of this problem is to sort the latitude-sorted array by longitude as well. Assume that the passed in array from the part b works as expected.

```
public Piece[][] sortFully(Piece[][] p) {
```

Practice Midterm 1- Solutions

This test has 9 questions worth a total of 100 points, and is to be completed in 110 minutes. The exam is closed book, except that you are allowed to use one double sided written cheat sheet (front and back). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. You may do this before the exam begins.

#	Points	#	Points
1	4.5	6	20
2	7	7	8
3	10	8	0
4	7	9	10
5	8	10	25
		Total	100

1. (Vitamin) C what's going on?: (4.5 pts) Step through the running of the following program and at each blank write the values of o1.x[0], o1.x[1], o2.x[0], and o2.x[1]. Assume that we start off with the constructor being called.

```
public class OJ{
  int[] x;
  OJ z;
  OJ(int x, int y){
     this.x = new int[2];
     this.x[0] = x;
     this.x[1] = y;
  }
}
public class Juice {
  public OJ o1;
  public static OJ o2;

  Juice() {
     o1 = new OJ(1, 2);
     o1.z = new OJ(5, 6);
     o2 = new OJ(3, 4);
     o2.z = new OJ(7, 8);
     pulpify();
     vitaminSeed();
     appleImposter();
  }

  public void pulpify() {
     o1.x[1] = o2.x[1];      o1.x[0] = 1, o1.x[1] = 4 , o2.x[0] = 3 , o2.x[1]= 4
  }

  public void vitaminSeed() {
     o1.x[0] = o1.z.x[0]; o1.x[0] = 7, o1.x[1] = 4 , o2.x[0] = 3 , o2.x[1]= 4

     o2.x[0] = o2.z.x[1]; o1.x[0] = 7, o1.x[1] = 4 , o2.x[0] = 8 , o2.x[1]= 4
     o1.z = o2;
  }

  public void appleImposter() {
     o1.x[1] = o2.x[0];
     o2.x[0] = o1.x[1];
     o2.x[1] = o1.z.x[0]; o1.x[0] = 1, o1.x[1] = 8 , o2.x[0] = 8 , o2.x[1]= 8
  }
}
```

2. Errrrr…..er: (7 pts) Find all the compilation errors and mark them with a "C". Find all runtime errors and mark them with a "R". Note, if a line relies on something that has errored out, you can assume that the prior error was fixed. Assume we start off by calling the constructor. For every error give a brief explanation why it errors out.

```
public class CorR {
   public static final int[] arr = new int[10];
   int i;
   XD xd;

   private class XD {
      private int val;

      XD(int x) {
         val = x;
      }
   }

   CorR() {
      i = 5;
      xd = new XD(i);
      diggity();
      dawg();
      coolCat();}

   private static void diggity() {
      arr[0] = 10;
      xd.val = 0; Compilation Error, cannot reference instance variable from static method
     arr[2] = 15;}

   private void dawg() {
      for (int i = 0; i < arr.length; i++) {
         arr[i] = i * 2 + 1 - 10 + .5;Compilation Error, cannot put a double in an int array
      }
     }
      diggity();
      new int[] temp = new Integer[10]; Compilation Error, cannot create an Integer Array when
the static  type is an int array
      arr = temp;} Compilation Error, cannot reassign the address of the final array

   static void coolCat() {
      i++; Compilation Error, cannot reference a reference variable from a static method
      dawg();
      arr[2] = arr[1] + 5;
      xd = new XD(10);Compilation Error, cannot instantiate within static methods.
      int[] temp = ((int[]) new double[10]); Compilation Error, cannot cast a double to an int
      temp = arr;}}
```

There was not a single runtime error on this problem

3. Casts and Inheriting Broken Bones: (10 pts) Below is a set of classes. We will have a series of method calls. In the lines following each method call, write what is printed, if anything at all. If there is a compilation or runtime error please say which one it is and provide an explanation. You may assume that previous lines affect the following.

```
public class Container {
  int size; boolean haslid; String name;
  public Container(){
     size = 0;
     haslid = false;
     name = "bad container";
     System.out.println("no constructor");}

  public Container(int size, boolean liddy, String name){
     this.size = size;
     this.haslid = liddy;
     this.name = name;
     System.out.println("here it is");}

  public void open() {
     System.out.println("there");}

  public void close() {
     if (!haslid) {
        System.out.println("Can't close what isn't there");}
    else{
          System.out.println("Closed");
}}}
public class NutellaJar extends Container {
          int sweetness;
          public NutellaJar() {
             System.out.println("I'm so hungry");
             this.sweetness = 100;}

          public NutellaJar(int size, boolean lid, String name, int sweetness) {
             super(size, lid, name);
             System.out.println("oink");
             this.sweetness = sweetness;}

          public void taste() {
             System.out.println("Just one more scoop");}

          public void taste(int scoops) {
             System.out.println("I just ate" + scoops + ". yum");}

          public void close() {
             System.out.println("This is too hard!");}}
```

Container plainjar = new Container(); no constructor _____ _____

plainjar.close(); Can't close what isn't there _____ _____

Container tasty = new NutellaJar(); no constructor I'm so hungry _____

tasty.taste(); Compilation error _____ _____

tasty.close(); This is too hard! _____ _____

NutellaJar scrumptious = new Container(); Compilation Error _____ _____

NutellaJar nutty = (NutellaJar) plainjar; Runtime Error _____ _____

NutellaJar n = new NutellaJar(5, true, "sweet thang", 10); Here it is oink _____

scrumptious.close() Compilation Error _____ _____

nutty.close(); Runtime Error _____ _____

((NutellaJar) tasty).taste(10); I just ate 10 scoops. yum _____ _____

nutty.taste(); Runtime Error _____ _____

4) It's always my de-Fault: (7 pts) Use the below interfaces to create a class that implements Viking and compiles.

```
public interface Norse {
  final static int burliness = 100;
  void breathe();
  void grunt();
}
public interface Viking extends Norse{
  final static int burliness = 300;
  void attack();
  boolean fly();
  void grunt();
  default void grunt(String t){
     System.out.println(t + "ARRRRRGH");
  }

}
```

```
public class Thor implements Viking {
  public void breathe(){
     System.out.println("breathing noise");
  }
  public void attack(){
     System.out.println("fight on with my " + burliness + " men");
  }
  public boolean fly(){
     System.out.println("I am flying!!!");
     return true;
  }
  public void grunt(){
     System.out.println("arrggh");
  }
}
```

*This problem was not too difficult, the important thing to realize is that when implementing Viking, you must use the methods from Norse (specifically breathe since grunt() was overridden.

5. Osmosis: (8 pts) We want to add a method to IntList so that if 2 numbers in a row are the same, we add them together and make one large node. For example:
1 ⇒ 1 ⇒ 2 ==>3 becomes 2 ⇒ 2 ⇒ 3 which becomes 4 ⇒ 3

```
public class IntList {
  public int first;
  public IntList rest;

  public IntList(int f, IntList r) {
     this.first = f;
     this.rest = r;
  }

  public void addAdjacent() {
  IntList p = this;
  if (p == null) {
     return;
  }
  IntList s = p;
  while (s.rest != null) {
     if (s.first == s.rest.first) {
        s.first = s.first * 2;
        s.rest = s.rest.rest;
        s = p; //addAdjacent() is also allowed
     } else {
        s = s.rest;
     }
  }
 }
}
```

The 1st error that is easy to make is checking to see if "this" is null. This can never be null in java, so we will have a pointer towards this. Since we did not allow access to size, you had to do this (non elegant) check. Now getting on to our while loop. The problem was easy to enough if you did not have to go backwards. If we initially started with the list 2 ⇒ 1 ⇒ 1 and you only went through 1 iteration of the while loop, you would end up with 2 ⇒ 2. This is not allowed under our conditions so you would set s = p, and start the process from scratch, as soon as you find 2 elements are the found to be the same . One could have done a recursive call to addAdjacent in its place, either method works.

6. Interfering Interfaces: (20 points) We want to write two classes, Sandwich and Pizza. Both of these classes will implement the interface ToppableFoods. A quality of all ToppableFoods is that you can add all the toppings you want that anything that extends the Ingredient class.

Two specific classes that extend the Ingredient class are Topping and Saucy.

The first item added to a sandwich must be a slice of bread and you can add anything up until you add a second slice of bread (sorry no triple deckers). For Pizzas' the first item added must be "Dough", any amount of ingredients may be added.

Regarding Sauces, Sandwiches can add any amount of sauces whenever you want. On the other hand, pizzas can have a maximum of 2 sauces added BEFORE toppings are added (you can also go sauceless like a savage).

Just like we can add toppings and sauces, we can also remove them. In both cases, only the most recently added item is allowed to be removed. Any attempt to remove any other item should result in an IllegalArgumentException. You can remove ingredients until there are no more left. Once you remove an item, you should return it.

- Note for the purpose of this problem, Bread and Dough are considered Toppings.
- You may use a LinkedListDeque or ArrayList Deque
- You may find instanceOf useful
    - Implementation is as follows: <Object> instanceOf <Class name>
    - Returns a boolean value (true if it is an instance of that class, false otherwise).
- All methods that will be common to both Sandwich and Pizza should be in ToppableFoods
- You will have two pages to write the Pizza class and the Sandwich class. You may include any helper classes and instance variables.

Below we have implemented the Ingredient class, the Topping class, the Saucy class,an example implementation of a class that implements Topping, and the Deque interface.

```
public class Ingredient{
String name;
Ingredient(String
name){
this.name = name;}}
```

```
public class Saucy extends
Ingredient{
    Saucy(String name){
     super(name);
}}
public class Topping
extends Ingredient{
     Topping(String name){
     super(name);
}}
```

```
public class Bread
extends Topping{
  int grain;
   Bread(int grain){
   super("Bread");
   this.grain = grain;
}
```

```
interface
Deque<Item> {
void addFirst(Item x);
 void addLast(Item x);
boolean isEmpty();
int size();
void printDeque();
Item get(int index);
Item removeFirst();
Item removeLast();}
```

```
public interface ToppableFoods{
  public void addIngredient(Ingredient t);
  public Ingredient removeIngredient(Ingredient t);
}
```

```
public class Sandwich implements ToppableFoods{
  LinkedListDeque<Ingredient> ig;
  int amountofbread;

  public Sandwich() {
     ig = new LinkedListDeque<Ingredient>();
     this.amountofbread = 0;
  }

  public void addIngredient(Ingredient t) {
     if (ig.size() == 0 && t.name != "Bread") {
        throw new IllegalArgumentException("Need to have bread first?");
     }
     if (amountofbread >= 2) {
        throw new IllegalArgumentException("No more toppings allowed, you already have 2 breads");
     }
     if (t.name == "Bread") {
        amountofbread += 1;
     }
     ig.addLast(t);
  }

  public Ingredient removeIngredient(Ingredient t) {
     if (ig.size() == 0) {
        throw new IllegalArgumentException("Can't remove nothing");
     }

     if (ig.get(ig.size() - 1).name != t.name) {
        throw new IllegalArgumentException("This is not the most recently added element");
     }
     if (t.name == "Bread") {
        amountofbread -= 1;
     }
     return ig.removeLast();
  }
}
```

```
public class Pizza implements ToppableFoods {
  LinkedListDeque<Ingredient> ig;
  int numberOfSauces;
  boolean topping;

  public Pizza() {
    ig = new LinkedListDeque<Ingredient>();
    numberOfSauces = 0;
    topping = false;
  }

  public void addIngredient(Ingredient t) {
    if (ig.size() == 0 && t.name != "Dough") {
      throw new IllegalArgumentException("Can’t add to nothing");
    }
    if (t instanceof Saucy && numberOfSauces >= 2) {
      throw new IllegalArgumentException("Too many sauces");
    }
    if (t instanceof Saucy && (ig.get(ig.size() - 1) instanceof Topping) && (ig.get(ig.size() - 1).name !=
"Dough")) {
      throw new IllegalArgumentException("Can't put a sauce after a topping");
    }

    if (t instanceof Saucy) {
      numberOfSauces++;
    }
    ig.addLast(t);
  }

  public Ingredient removeIngredient(Ingredient t) {
    if (ig.size() == 0) {
      throw new IllegalArgumentException("There’s nothing to remove");
    }
    if (t.name != ig.get(ig.size() - 1).name) {
      throw new IllegalArgumentException("This is not the most recently added element");
    }
    if (t instanceof Saucy) {
      numberOfSauces -= 1;
    }
    return ig.removeLast();
  }
}
```

7. Troe & False: (8 pts)

a)General Colonel wants to make a method that both overloads and overwrites the method of a parent class. Is this possible? Explain your answer.

Making one method that both overloads and overwrites a method is impossible. For once overloading only refers to methods in the same class, when you have different arguments for a method with the same name. Overriding on the other hand is having a class that extends another class and has a method with the same name and same arguments.

b) When would you use a comparable over a comparator? Which one is preferable?

Neither is really better than the other, it just depends how you are using them. A comparable is used for having a "natural" order, basically you want the objects to always be compared in some fashion. A comparator is more of a "one-time" use case and just used to compare for a specific time.

c) An instance of a class has a broader scope than just the class. That is, calling a method or variable from a class may cause a compilation error, but an instance of it won't.

True, an instance of a class has access to static and dynamic methods and variables. The only time this

d) Does overloading a method take into account the return value?

Basically would changing public int hello(int hi){...} to public boolean hello(int hi){...} be valid? Explain why.

No, this is not valid. You cannot change the return value because there would be no way to differentiate which method java should choose. How would it know which return value you are expecting?

8) Riddle me this (0 pts): What weighs six ounces, sits in a tree, and is very dangerous?

A sparrow with a machine gun

9) A Test Within a Test?: (10 pts)

a) Corn on the Cobb wants to is attempting to use an IntList to get through dreams. At any given time, the IntList's size must be a maximum of 5. If the IntList's size ever seems like it is going to exceed 5, you must remove the first node, making the second element the new first. Write dreaming which adds nodes and makes sure that the IntList fulfills the requirements that we put on it above. (8 pts)

```
public class IntList{
   public int first;
   public IntList rest;
}
public void dreaming(int n){

        if(this.size == 0){
                this = new IntList(n, null);
        }
        else{
                if this.size() == 5{
                        this.first = this.first.rest;
                        this.rest = this.first.rest;
                }
                Intlist p = this;
                while(this.rest!=null){
                        p = p.rest;
                        }
                p.rest = new IntList(n, null);
                }
        }
```

b) Corn on the Cobb now wants to test this code to make sure that he does not lose his way in his dreams (that would really suck). Write a basic JUnit test to make sure that your code works as expected. Note: The IntList.list(1, 2, 3 , 4, 5) would make an int list 1 ⇒ 2 ⇒ 3 ⇒ 4 ⇒ 5 (3 pts)

```
@Test
public testDreaming(){
        IntList tst = new IntList();
        tst.list(2,3,4,5);
        Intlist second = new IntList();
        second.dreaming(1);
        second.dreaming(2);
        second.dreaming(3);
        second.dreaming(4);
        second.dreaming(5);
        assertEquals(5, second.size());
        assertEquals(tst, second);
}
```

10) Arrrrghrays: (25 pts) Purplebeard and his lackey Turquoisenail are sailing the 10 seas. In order to sail well, they want to be able to create a map. They managed to create their map, but Turquoisenail tripped and put it through a paper shredder. They managed to store the scrap images into a 1d array, but they need to make it into a NxN map. You are lucky because on each piece you have the longitude and latitude written down. Write a short program to help put the pieces back together. The pieces should be as follows:

-100, 30	-50, 30	-25, 30
-100, 20	-50, 20	-25, 20
-100, 10	-50, 10	-25, 110

In the upper left corner, -100 is the longitude and 30 is the latitude.

For this problem, you have access to IntLists, ArrayDeques, arrays, and LinkedListDeques

Note: This problem is very hard, and it is expected that you attempted earlier problems before looking at this one.

You

a) For the first part of this problem, make a Piece class that store longitude and latitude. (**2 pts)**

```
public class Piece{
        public int longitude;
        public int latitude;
        public Piece(int x, int y){...}
}
```

b) For the second part of this, make a method that takes in an array of all the pieces and returns a 2d array that is sorted by latitude. **(15 pts)**

```
public Piece[][] sortByLat(Piece[] p) {
  int width = (int) Math.sqrt(p.length);
  Piece[][] latSort = new Piece[width][];
  for (Piece pi : p) {
    boolean nullp = true;
    for (int i = 0; i < latSort.length; i++) {
      if (latSort[i] != null) {
        if (pi.latitude == latSort[i][0].latitude) {
          int index = 0;
          while (latSort[i][index] != null) {
            index++;
          }
          latSort[i][index] = pi;
          nullp = false;
          break;}}}
    if (nullp == true) {
      int in = 0;
      while (latSort[in] != null) {
        in++;}
      latSort[in] = new Piece[width];
      latSort[in][0] = pi;
    }}
  Piece[][] temp = new Piece[width][];
  int count = 0;
  while (count < width) {
    int maximum = Integer.MIN_VALUE;
    int maxindex = 0;
    for (int j = 0; j < latSort.length; j++) {
      if (latSort[j] != null) {
        if (latSort[j][0].latitude > maximum) {
          maximum = latSort[j][0].latitude;
          maxindex = j;
        }}
      if (j == latSort.length - 1) {
        temp[count] = latSort[maxindex];
        latSort[maxindex] = null;
        count++;
      }}}
  latSort = temp;
  return latSort;
}
```

This Solution is pretty complicated. We started by finding all the elements that have the same latitude and putting them into “buckets”. Then we sort the buckets themselves by latitude.

c) The final part of this problem is to sort the latitude-sorted array by longitude as well. Assume that the passed in array from the part b works as expected. **(8 pts)**

```
public Piece[][] sortFully(Piece[][] p) {
  for (int i = 0; i < p.length; i++) {
     Piece temp;
     for (int j = 1; j < p.length; j++) {
        for (int k = j; k > 0; k--) {
           if (p[i][k].longitude < p[i][k - 1].longitude) {
              temp = p[i][k];
              p[i][k] = p[i][k - 1];
              p[i][k - 1] = temp;
           }
        }
     }
  }
  return p;
}
```

Note: The solution in part C uses a type of insertion sort that will be looked over later on in the course.

Practice Midterm 2

This test has 10 questions worth a total of 120 points, and is to be completed in 110 minutes. The exam is closed book, except that you are allowed to use one double sided written cheat sheet (front and back). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. Write the statement out below in the blank provided and sign. You may do this before the exam begins.

#	Points	#	Points
1	15	6	12
2	9	7	12
3	0	8	9
4	6	9	12
5	20	10	25
		Total	120

1. Dat-uhhh Structures: (15 pts) Provide the solutions in the blanks under the questions. Each question is worth 3 points.

a) How many nodes would be in the left subtree of a complete binary search tree where height (h) > 0 and the root starts at a height of 0. Note, if you want to use height in your calculations, you must use the height of the full binary search tree, not subtrees.

b) How many nodes would be in the right subtree of the right subtree in a full binary search tree. The same assumptions should be made as in the previous problem

c) How many different heaps can be both max heaps and min heaps? If there are none, justify why, tell us the cases that result in heaps that fit this criteria.

d) Can red-black trees have only black links? When is this the case?

e) What is wrong with the following code snippet? Assume that it compiles properly, there is a constructor, and a hashcode method that has a hashcode function works well (spreads everything well).

```
Public class Human{
int legs;
int arms;
….
@Override
public boolean equals(Human no) {
        return legs == no.legs && arms == no.arms;
}
```

2) Flash Union: (9 pts)

a) Beary Allen, a fast UC Berkeley student wants to make trips between cities. He wants to see if they are connected in the fastest way possible- as a result, he will use the fastest union data structure that we were taught. His *travel* method acts like *connect* or *union* in a normal union data structure. Draw the Union Structure in each box after all the commands have executed. (5 pts)

Gotham Metropolis Central City Star City Atlantis Coast City
travel(Gotham, Star City); travel(Atlantis, Metropolis); travel(Central City, Coast City);
travel(Metropolis, Central City);
travel(Metropolis, Star City);

b) Beary Allen wants to determine the longest time that it would take for him to check if 2 cities are connected. Please write the runtime in terms of N, the number of cities. (2 pts)

c) Beary decided that the runtime is not fast enough for the fastest man in the world. Can he go faster by changing the way the Union Data Structure works? If so, briefly describe how you would do so. (1 pt)

3) Riddle me this: (0 pts)
Which president wears the biggest hat?

4) Pitching and Catching: (6 pts) What does the following code display? You need not use all the lines.

```
  public static void triedTooHard(String[] n){
     try{
        for(int i = 0; i < n.length + 1; i++){
           System.out.println(n[i].charAt(0));
        }}
     catch(NullPointerException E){
        System.out.println("Tried" + n.length + "hard");}
     catch(IndexOutOfBoundsException E){
        String[] t = new String[n.length+1];
        System.arraycopy(n, 0, t, 1, n.length);
        triedTooHard(t);
     }
     finally{
        System.out.println(n.length);}}

public static void main(String[] args){
  triedTooHard(new String[] {"my ", "everything ", "my", "eternal"});
}
```

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

5) B-arbor Day: (20 pts) Woody TreePecker is a bit sad because he wants a place to perch. The only problem is he doesn't have a nice bushy tree to perch in- all he has is spindly LinkedLists. He has employed you, professional tree barber (or gardner whatever you want to call it), to give him some really nice trees. Now you, being the conservationist you are, want to turn the LinkedLists into Balanced Binary Search Trees.

Write the code that will convert a sorted LinkedList (one from java's documentation) into a Balanced Binary Search Tree (that can contain generics). For consistency, your main method that converts the LinkedList to a Binary Search Tree should be labeled **LinkedListToBST**. You will be given the remainder of this page and one more page to complete your class.

6) Test making you sad? Call that A-simp-totics: (12 pts) Write the Asymptotic runtime of the following function. Use the tightest bounds possible. Each problem is worth 4 points.

1.
```
public static void louwTheWay(int N){
    if(N<=1){
        return 1;
    }
    for int(i = 0; i < N / 2 ; i ++){
       System.out.println("you");
     }
    louwTheWay(N - 1); louwTheWay(N-2);\\lie
}
```

2.
```
public static void takeCaretik(int N){
        if(N <= 1){
            return 1;
        }
        for(int i = 0; i < (N*1000+30^100)/N ; i ++){
            System.out.println("I'll take care of you.");
            }
     takeCaretik(N/4);
  }
```

3.
```
public static void weDontTalk(int N){
    if(N<=1){
        return N;
     }
     for(int i = 0 ; i < N; i++){
        for (int j = 0 ; j < i; j++){
          System.out.print("Anymore")
           }
       }
     weDontTalk(N/2) \\Like we used to
  }
```

7) MashedSet: (12 pts)

a) We have a HashSet, but a goon broke part of our code so it does not handle duplicates properly. This means that duplicates can added without care. Luckily, your hashcode, which is quite nice, is still intact (it was encrypted well), along with this, the spread of the hashcode is pretty good. Determine how long it would take in order to traverse your broken HashSet to find and delete ALL duplicates of ONE key. For example, how long would it take to find the key "macaroni". Provide the runtime (It is your choice what symbol to use) based off our prior assumptions. Give reasons for both runtimes. Correct runtimes with wrong explanations will be given 0 points.

b) You're trying to figure out how the goon managed to break your code so easily. You spent so many hours on it, and he managed to change it very quickly so that duplicates aren't handled. What is the most probable way he did this?

c) Pretend that you did not remove the duplicates from the faulty HashSet and you wanted to keep track of how many times each item was inside of the HashMap. How would one do this? We would like our solution to be as fast as possible.

8) Rotations and Flips: (9 pts)

a) What series of inserts would result in a perfectly balanced binary search tree for the following input: 1, 2, 3, 4, 5, 6 , 7 , 8 , 9 , 10, 11, 12, 13, 14, 15. Draw the resulting Binary Search Tree.

b) Delete the item 12 from the above Binary Search Tree and draw the results.

c) Insert 7 into the following red black tree. Show your steps using the red-black tree method for full credit. ½ credit will be given for converting into a 2-3 tree and back.

9) Runtimes not finished call that Not dun-times: (12 pts) Write all runtimes in Theta or Omega and O. Ignore constants but do not ignore lower order terms. Provide an explanation for your runtimes.

a) Runtime of putting N presorted keys into a binary search tree.

b)You have a minheap and you will perform deleteMin operations until the min-heap is empty. Whenever an element is deleted, it is added to a Binary Search Tree.

c) Inserting N elements into a HashMap that all have the same key and different values.

d) Given an array sorted from least to greatest put all the elements into a stack (lowest index to highest index)then pop off all the elements. Each time an element is popped off put it into a maxheap.

10) NBA: National Bongoola Association: (25 pts)

You are the coach of a prestigious Bongoola team. Your job as coach is to make sure that, at any given time, the best players on your team for each positions are on the field. Bongoola, being a great community team sport, invites all people of the community to join. Each player will be assigned a position that they can play- there are a total of **M** positions- and a unique team number. For this problem, we will use **N** to be the total amount of people on your team and **P** to be the people signed up for a certain position.

You have your starting **M** players, but as the game progresses, you will need to replace them. You will base your replacement of people using the *bodacious* factor. During the game, only 1 player's bodacious factor goes down, and once they are benched, their bodacious factor doesn't change. Every **Z** minutes, 1 player on the field will have their bodacious factor decrease and you will be able to replace the player if needed. Once the next eligible player's (a player who plays the same position) bodacious factor exceeds the current player, they are considered a better choice. You are guaranteed that only 1 of the players at a time will need to be replaced. Changing a player's bodacious factor should take $\Theta(l)$ time, replacing a player should take $\Theta(\log(P))$ time, and putting a new player in the current lineup should take $\Theta(1)$ time.

On the next page, write a detailed description of what data structures you will use to make the operations stated above run in the provided time. State any assumptions you need to regarding anything (no you cannot assume that everything is done by some magical warlock).

Practice Midterm 2- Solutions

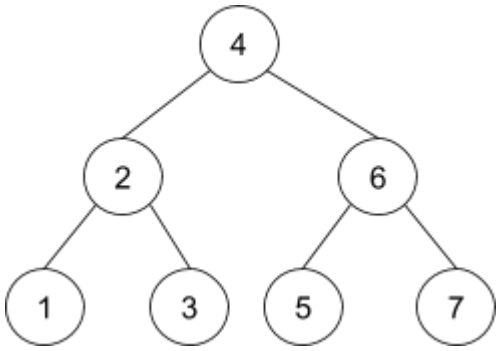
This test has 10 questions worth a total of 120 points, and is to be completed in 110 minutes. The exam is closed book, except that you are allowed to use one double sided written cheat sheet (front and back). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. Write the statement out below in the blank provided and sign. You may do this before the exam begins.

#	Points	#	Points
1	15	6	12
2	9	7	12
3	0	8	9
4	6	9	12
5	20	10	25
		Total	120

1. Dat-uhhh Structures: (15 pts) Provide the solutions in the blanks under the questions. Each question is worth 3 points.

a) How many nodes would be in the left subtree of a complete binary search tree where height (h) > 0 and the root starts at a height of 0. Note, if you want to use height in your calculations, you must use the height of the full binary search tree, not subtrees.

2^h - 1: This can be seen by drawing a basic complete binary search tree as follows:



In the left subtree, there are 3 nodes and the height is 2. You may have needed to draw a larger Binary Search Tree to see the pattern.

b) How many nodes would be in the right subtree of the right subtree in a full binary search tree. The same assumptions should be made as in the previous problem

2^{h+1} - 1: This can be seen by drawing a basic binary search tree such as the one above. It is probably a larger one than this one would have had to been used because there was only 1 node in the right subtree of the right subtree.

c) How many different heaps can be both max heaps and min heaps? If there are none, justify why, tell us the cases that result in heaps that fit this criteria.

Infinitely many. If all the elements are all the same.

d) Can red-black trees have only black links? When is this the case?

Yes, it is possible for red black trees to only have black links. This happens when there is only 1 node or with a very careful deletion. A red black tree with only black links would be the equivalent of a 2-3 tree with nodes that either have 0 children or 2 children. Additionally, a red-black tree with only black links would essentially just be a Binary Search Tree.

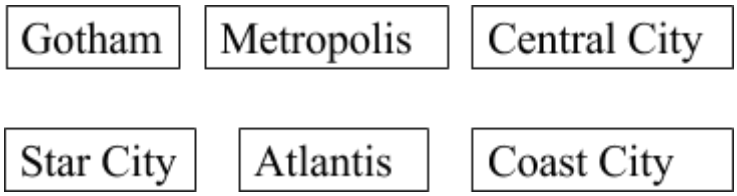
e) What is wrong with the following code snippet? Assume that it compiles properly, there is a constructor, and a hashcode method that has a hashcode function works well (spreads everything well).

```
Public class Human{
int legs;
int arms;
….
@Override
public boolean equals(Human no) {
        return legs == no.legs && arms == no.arms;
}
```

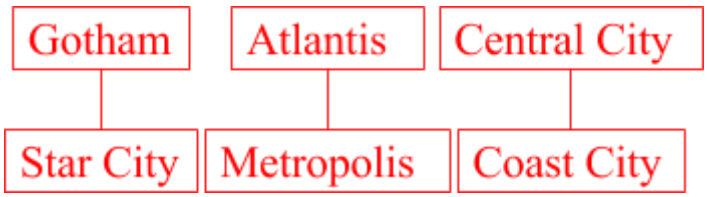
The main problem with this code is that the equals does not actually override any methods! The signature is wrong. Normally, the equals takes in an Object not a Human. This will cause errors when a “Human” is inputted.

2) Flash Union: (9 pts)

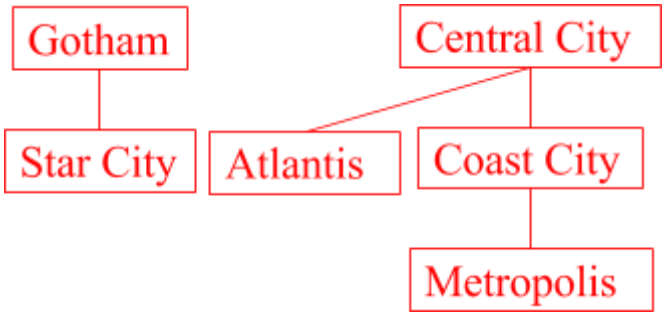
a) Beary Allen, a fast UC Berkeley student wants to make trips between cities. He wants to see if they are connected in the fastest way possible- as a result, he will use the fastest union data structure that we were taught. His *travel* method acts like *connect* or *union* in a normal union data structure. Draw the Union Structure in each box after all the commands have executed. (5 pts)



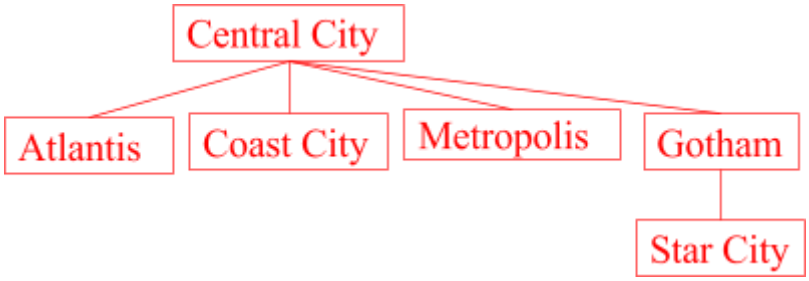
travel(Gotham, Star City); travel(Atlantis, Metropolis); travel(Central City, Coast City);



travel(Metropolis, Central City);



travel(Metropolis, Central City)



In this problem, it was important to understand the qualities of a Weighted Quick Union with Path Compression and that a Weighted Quick Union with Path Compression is the fastest Union Data Structure. When we call travel, we connect a node to its current root not the final root

b) Beary Allen wants to determine the longest time that it would take for him to check if 2 cities are connected. Please write the runtime in terms of N, the number of cities. (2 pts)
Almost $\Theta(1)$ but not quite. Amortized $\Theta(1)$ is acceptable.
c) Beary decided that the runtime is not fast enough for the fastest man in the world. Can he go faster by changing the way the Union Data Structure works? If so, briefly describe how you would do so. (1 pt)
No.
Not possible to argue in the scopes of the course. Another possible answer is that Beary can just run back in time and tell the city planners to put the cities closer together.

3) Riddle me this: (0 pts)
Which president wears the biggest hat?
The one with the biggest head.

4) Pitching and Catching: (6 pts) What does the following code display? You need not use all the lines.

```
  public static void triedTooHard(String[] n){
     try{
        for(int i = 0; i < n.length + 1; i++){
           System.out.println(n[i].charAt(0));
        }}
     catch(NullPointerException E){
        System.out.println("Tried" + n.length + "hard");}
     catch(IndexOutOfBoundsException E){
        String[] t = new String[n.length+1];
        System.arraycopy(n, 0, t, 1, n.length);
        triedTooHard(t);
     }
     finally{
        System.out.println(n.length);}}

public static void main(String[] args){
  triedTooHard(new String[] {"my ", "everything ", "my", "eternal"});
}
```

m
e
m
e
Tried5hard
Size is 5
Size is 4

The main tricky point in this problem is that the finally block happens AFTER everything in the normal try catch blocks. Because of this, the finally occurs after the recursive call has run to completion. The NullPointerException occurred in the second call because the item at the 0th index in the input array was null.

5) B-arbor Day: (20 pts) Woody TreePecker is a bit sad because he wants a place to perch. The only problem is he doesn't have a nice bushy tree to perch in- all he has is spindly LinkedLists. He has employed you, professional tree barber (or gardner whatever you want to call it), to give him some really nice trees. Now you, being the conservationist you are, want to turn the LinkedLists into Balanced Binary Search Trees.

Write the code that will convert a sorted LinkedList (one from java's documentation) into a Balanced Binary Search Tree (that can contain generics). For consistency, your main method that converts the LinkedList to a Binary Search Tree should be labeled **LinkedListToBST**. You will be given the remainder of this page and one more page to complete your class.

```
import java.util.LinkedList;

public class linkToBST {

  class TreeNode<T> {
     T item;
     TreeNode left, right;

     TreeNode(T item) {
        this.item = item;
        left = null;
        right = null;
     }
  }

  TreeNode LinkedListToBST(LinkedList l){
        return LinkedListToBST(l.size(), l);
  }
  TreeNode LinkedListToBST(int n, LinkedList l){
     if(n<=0){
        return null;
     }
     TreeNode left = LinkedListToBST(n /2 , l);
     TreeNode root = new TreeNode(l.removeFirst());
     TreeNode right = LinkedListToBST(n - n/2 - 1, l);
     root.left = left;
     root.right = right;
     return root;
```

```
    }
}
```

Since we are using the java implementation of a LinkedList, there was no need to create your own LinkedList node. Having a TreeNode class was pretty much a requirement because you needed to be able to store the left and right child. The method is relatively straightforward; however, it is very easy to trip up on the calculations.

We remove first when calculating the root because the first time that we want to make the element at index 0 into the leftmost element. We can take advantage of the fact that it is an ordered list to do this. The division by 2 allows us to “split” the list at that point so that everything remains balanced.

This is a relatively tricky problem, especially since it was hard to use your own LinkedList node (due to no private class allowed and thus no “static” head).

6) Test making you sad? Call that A-simp-totics: (12 pts) Write the Asymptotic runtime of the following function. Use the tightest bounds possible. Each problem is worth 4 points.

1. $\Theta(N*2^N)$

```
public static void louwTheWay(int N){
 for int(i = 0; i < N / 2 ; i ++){
    System.out.println("you");
  }
  louwTheWay(N - 1); louwTheWay(N-2);\\lie
}
```

We must realize that there are N layers. Additionally, the amount of work done per node is N. The nodes per layer is 2^i ,where i is the current layer, because of the 2 recursive calls. The summation would look as follows:

$$\sum_{i=0}^{N} N * 2^i \rightarrow N(2 + 2^2 ... + 2^n) = N * 2^{n+1} = 2\,N2^n = O(N*2^N)$$

2. $\Theta(\log(n))$

```
public static void takeCaretik(int N){
      if(N == 1){
            return;
      }
      for(int i = 0; i < (N*1000+30^100)/N ; i ++){
            System.out.println("I'll take care of you.");
            }
   takeCaretik(N/4);
}
```

The amount of work done in the for loop is constant (though a very large constant), so we disregard it and just consider it constant time. The amount of layers would be $\log_4 N$. Since we have constant work on each layer, 1 node per layer, and $\log_4 N$ layers, the runtime is $\Theta(\log(n))$.

3. $\Theta(N^2)$

```
public static void weDontTalk(int N){
    for(int i = 0 ; i < N; i++){
       for (int j = 0 ; j < i; j++){
        System.out.print("Anymore")
        }
     }
    weDontTalk(N/2) \\Like we used to
  }
```

The amount of work done in the second for loop is proportional to the work done in the first for loop- for the first iteration, this will be N^2; however, we have a recursive loop that we must take into account. The amount of work done per now is $(\frac{N}{2^i})^2$ or $(\frac{N^2}{4^i})$ where i is the current layer. The amount of layers is log(N) because we divide by 2 at each recursive call. This makes the summation formula $\sum_{i=0}^{log(N)} (\frac{N^2}{4^i}) \rightarrow (\frac{N^2}{4^{log(N}}) \rightarrow N^2$

7) MashedSet: (12 pts)

a) We have a HashSet, but a goon broke part of our code so it does not handle duplicates properly. This means that duplicates can added without care. Luckily, your hashcode, which is quite nice, is still intact (it was encrypted well), along with this, the spread of the hashcode is pretty good. Determine how long it would take in order to traverse your broken HashSet to find and delete ALL duplicates of ONE key. For example, how long would it take to find the key "macaroni". Provide the runtime (It is your choice what symbol to use) based off our prior assumptions. Give reasons for both runtimes. Correct runtimes with wrong explanations will be given 0 points.

Ω(1)- We have a good hashcode and a good spread, and since the key would be the same, it would always hash to the same bucket, this means that you would have to check very few elements.

O(N)- Technically this can happen if every single element in our HashMap is the same. It is not acceptable to say that the hashcode is bad.

b) You're trying to figure out how the goon managed to break your code so easily. You spent so many hours on it, and he managed to change it very quickly so that duplicates aren't handled. What is the most probable way he did this?

A change in the equals method for the item being hashed would result in the duplicates not being handled. Your equals method normally goes through the bucket to check if the element is not working, but if your equals method does not work, it won't realize there is a duplicate.

A NOT acceptable response would be that the hashcode function was screwed up as in the original problem it was stated that it remained intact.

c) Pretend that you did not remove the duplicates from the faulty HashSet and you wanted to keep track of how many times each item was inside of the HashMap. How would one do this? We would like our solution to be as fast as possible.

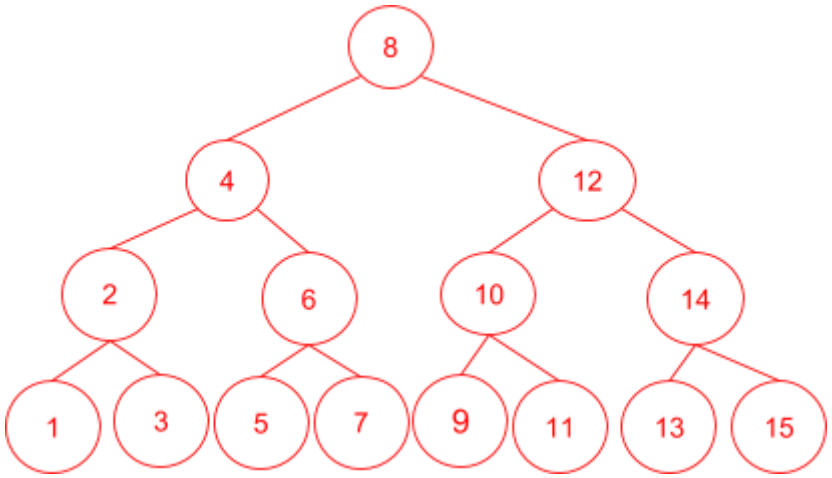
The most intuitive way would be to create a HashMap where you insert elements as you go through the original HashSet. The element would be a node that has a the key and the count. Every time you found a duplicate, you would find the node and change the value. This would take $\boldsymbol{\Theta}(1)$ because we would use the SAME hashcode that was provided to us earlier in the problem. The overall runtime as a result would be $\boldsymbol{\Theta}(N)$

8) Rotations and Flips: (9 pts)

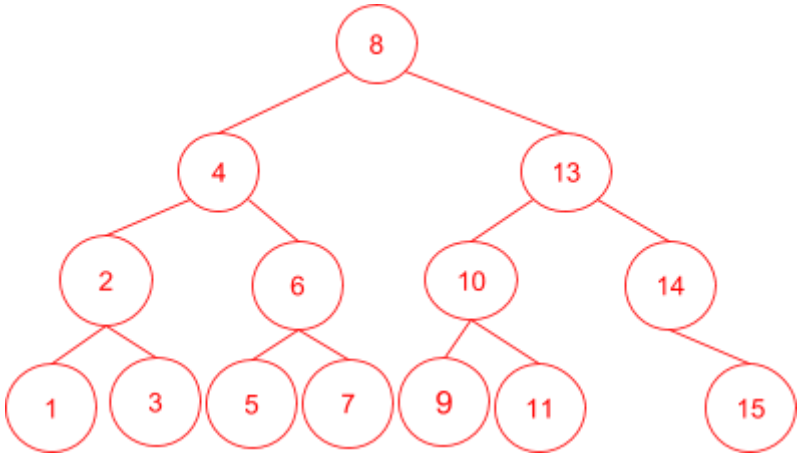
a) What series of inserts would result in a perfectly balanced binary search tree for the following input: 1, 2, 3, 4, 5, 6 , 7 , 8 , 9 , 10, 11, 12, 13, 14, 15. Draw the resulting Binary Search Tree.

8, 12,4, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15

A way that makes this a bit easier to approach is construct a 2-3 tree and see where the nodes get promoted to. It decreases the randomness and allows you to do it relatively quickly.

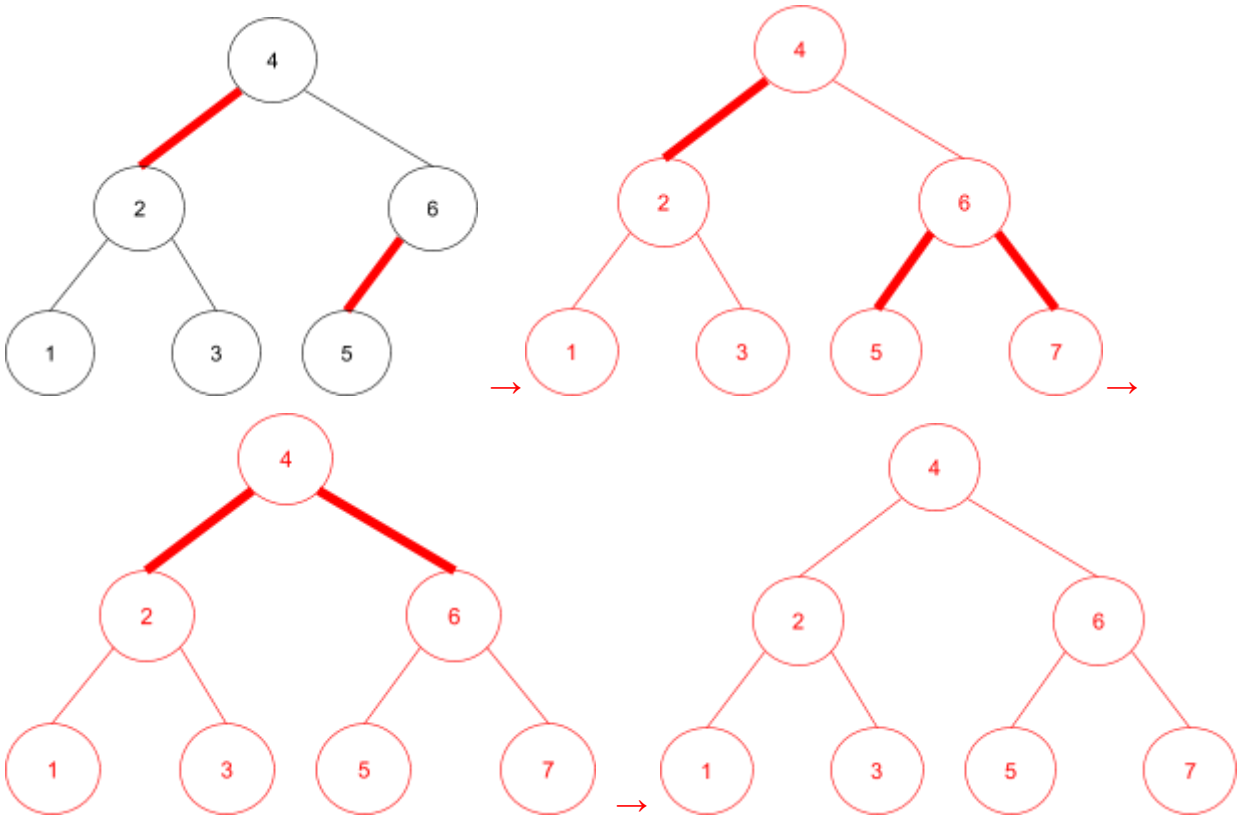


b) Delete the item 12 from the above Binary Search Tree and draw the results.



To do this, you want to do the “hibbard” deletion where you find the smallest item on the right side of the element and then swap it with the item

c) Insert 7 into the following red black tree. Show your steps using the red-black tree method for full credit. ½ credit will be given for converting into a 2-3 tree and back.



Problem is straightforward if you know the rules well

9) Runtimes not finished call that Not dun-times: (12 pts) Write all runtimes in Theta or Omega and O. Ignore constants but do not ignore lower order terms.

a) Runtime of putting N presorted keys into a binary search tree.
$\boldsymbol{\Theta}(N^2)$ The items are presorted (either least to greatest or greatest to least) and this means that the Binary Search Tree will end up looking like a LinkedList

b)You have a minheap and you will perform deleteMin operations until the min-heap is empty. Whenever an element is deleted, it is added to a Binary Search Tree.
$\boldsymbol{\Omega}(NlogN)$- Deleting the min takes log(N) time and you perform that operation N times. Then, in best case, inserting into a Binary Search Tree takes $\boldsymbol{\Theta}(log(N))$ time, it is done N times. Accounting for both operations makes the value $\Omega(NlogN + NlogN) = \Omega(2NlogN)$

$\mathbf{O}(N^2 + NlogN)$ - Deleting the min takes log(N) time and you perform that operation N times. The N^2 comes from the worst case runtime of inserting into a Binary Search Tree being $\boldsymbol{\Theta}(N)$ time and that operation is performed N times ($N*N = N^2$)

c) Inserting N elements into a HashMap that all have the same key and different values.
$\boldsymbol{\Theta}(N)$- You insert N elements and since HashMaps allow for no duplicates, you will never have more than 1 element in your HashMap.

d) Given an array sorted from least to greatest put all the elements into a stack (lowest index to highest index)then pop off all the elements. Each time an element is popped off put it into a maxheap.
$\boldsymbol{\Theta}(N)$- The items are presorted and since it is put into the stack, the greatest elements are always above the smaller elements. As a result, no swimming will ever need to be done. We put N elements into a stack, pop off all N elements and insert them into a heap with no swimming so N insert operations. This results in 3N or $\boldsymbol{\Theta}(N)$

10) NBA: National Bongoola Association: (25 pts)
You are the coach of a prestigious Bongoola team. Your job as coach is to make sure that, at any given time, the best players on your team for each positions are on the field. Bongoola, being a great community team sport, invites all people of the community to join. Each player will be assigned a position that they can play- there are a total of **M** positions- and a unique team number. For this problem, we will use **N** to be the total amount of people on your team and **P** to be the people signed up for a certain position.

You have your starting **M** players, but as the game progresses, you will need to replace them. You will base your replacement of people using the *bodacious* factor. During the game, only 1 player's bodacious factor goes down, and once they are benched, their bodacious factor doesn't change. Every **Z** minutes, 1 player on the field will have their bodacious factor decrease and you will be able to replace the player if needed. Once the next eligible player's (a player who plays the same position) bodacious factor exceeds the current player, they are considered a better choice. You are guaranteed that only 1 of the players at a time will need to be replaced. Changing a player's bodacious factor should take $\boldsymbol{\Theta}$(l) time, replacing a player should take $\boldsymbol{\Theta}$(log(P)) time, and putting a new player in the current lineup should take $\boldsymbol{\Theta}$(1) time.

On the next page, write a detailed description of what data structures you will use to make the operations stated above run in the provided time. State any assumptions you need to regarding anything (no you cannot assume that everything is done by some magical warlock).

First thing is first, you will need a **Player** class with a *bodacious* instance variable. You will then create a HashMap which will hash based off the unique player id's. Because of the HashMap, we can change the *bodacious* factor of a player in $\boldsymbol{\Theta}$(1) time. For this part of the problem, you need to state that you assume that there is a good hashcode and good spread- if this was not the case, it would be possible to have $\boldsymbol{\Theta}$(N) runtime.

To represent the players who can play in a certain position, we will use a priority queue. The top player in the priority queue will be in the current lineup. Once their *bodacious* factor is less that one of their children, we will sink the player. This results in $\boldsymbol{\Theta}$(log(P)) time to replace a player (if they truly need to be replaced).

Now it gets a little tricky. Since there are M positions, we cannot disregard the amount of positions as a constant. Two get around this, there are two possible (intuitive) solutions:

The first solution involves realizing that there are as many priority queues as there are positions. You can create 2 M sized arrays- one array will be the current lineup and the other array will be composed of priority queues in the same order. You would modify the **Player** class so that it has

an instance variable that stores the index that it would go to when replacing or being replaced. Accessing an array would occur a maximum of 2 times which would make the runtime $\Theta(1)$ time.

The second solution would be slightly more complicated. You can create 2 HashMaps, again, one of the current lineup and one of the priority queues. However, since the hashing of priority queues is not straightforward, we would create a wrapper class for priority queues and create some instance variable that was unique to each priority queue. For example, we could have a string that would be the name of the position it represents. We then hash the M wrapper classes. Accessing a priority queue would take $\Theta(1)$ time if we assume a good hashcode. If good hashcode was not assumed, then it would take $\Theta(M)$ time

The complexity in this solution (specifically when you attempt to hash the lineup) is that players who replace others would need to have the same hashcode. To do this, the key in this HashMap should be the value of its corresponding wrapper classes priority queue (in our example the name of the position) and the value should be the player. Replacing the item in the HashMap would take $\Theta(1)$ time since we know where it hashes to.

Since the *bodacious* factor only decreases at the end of the Z minutes, we can just make it so that when a sinking occurs, the player is changed.

Practice Final

This test has 13 questions worth a total of 200 points, and is to be completed in 180 minutes. The exam is closed book, except that you are allowed to use one double sided written cheat sheet (front and back). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. Write the statement out below in the blank provided and sign. You may do this before the exam begins.

#	Points	#	Points
1	16	8	0
2	12	9	10
3	9	10	15
4	25	11	20
5	8	12	17
6	25	13	30
7	9		
		Total	200

1. Starting this Test on a Positive Note (15 pts).
a) (4 pts) Give an example where Dijkstra's would fail because of a negative edge weight

b) (4 pts) Give an example where Dijkstra's **does not** fail with n negative edge weights. State assumptions that you must make in order for this to work. If you have some method to make it work, explain the method.

c) (4 pts) What is the least amount of negative edges (>0) that a graph can have and still have Dijkstra's work? What is the most amount of negative edges that we can have? State assumptions that must be made

d) (4 pts) What is the smallest weight we can have and guarantee that Dijkstra's will always work, regardless of the graph's construction?

2. Sorting and Fun (12 pts). We worked with a few various sorts. For each sort, provide the runtime and give a brief description of what it does (how it sorts). Each questions is worth 4 points

	private static void dwarfSort(int[] ar) {
	  int i = 1;
	  int N = ar.length;
	  while (i < N) {
	  if (i == 0 || ar[i - 1] <= ar[i]) {
	  i++;
	  } else {
	  int tmp = ar[i];
	  ar[i] = ar[i - 1];
	  ar[i--] = tmp;
	  }
	  }
	}

	private static int[] waffle(int[] arr){
	  int[] syrup = arr;
	  for(int i = 1; i < arr.length; i++){
	  if(arr[i] < arr[i-1]){
	  arr[i] = arr[i] + 1;
	  }
	  }
	  return syrup;
	}

	//find the runtime of flapJackSort. Note that the flip function is defined as follows flip(array, index) and reverses the entire array up until that index.
	public static int[] flapJackSort(int arr[], int n){
	  for (int curr_size = n; curr_size > 1; curr_size--)
	  {
	  for(int choco = 0, chip =0; chip < curr_size; chip++) {
	  if (arr[chip] > arr[choco]) {
	  choco = chip;
	  }
	  if (choco != curr_size - 1) {
	  flip(arr, choco);
	  flip(arr, curr_size - 1);
	  }
	  }
	  }
	  return arr;}

3. The Fruits of Your Labor (9 pts).

a) (3 pts) You own a trie that can hold many fruits; however, you want to save space so that your neighbor doesn't cut off your branches. Find how you make one modification (change, delete, or add a letter) to one of the following 3 words "Apple", "Peach, "Pear". What modification can you do to save the most space? Draw the resulting trie after your modification.

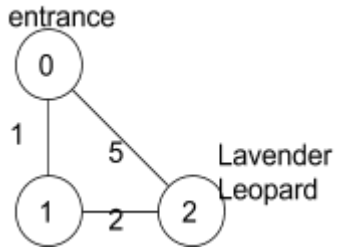
b) (3 pts) A wealthy CEO wants to encrypt all his secrets; however, he decided that he wants to store all of his secrets within a flashy hashmap. But he doesn't want a plain old hashmap, after all, that could get hacked. To be different, he decides that, for any given day, a secret will hash to different location than it would have the previous day. Is his idea valid? Provide your reasoning.

c) (3 pts) 2-3 Trees are known to be fast because they are self balancing and they have a nice low level? Why don't we just use 2-3-4 trees or 2-3-4-5-6 trees?

4. The Lavender Leopard (20 points).

You are a jewel thief starting at the entrance of a museum where all rooms are connected to each other- that is a room is connected to every room besides itself; however, hall lengths may differ. You are in search of the very expensive gem, the "Lavender Leopard", the only problem is that you have no idea what room it is in.
Your idea is to go to the room closest to the one you are in (excluding ones that you have already visited) and then as soon as you find the gem, you will immediately exit the building, going through the same rooms you did earlier; however, if there was a faster way to a room that is closer to the exit, you will go that way.

a) (20 pts) Fill in the Museum class and following code to find the gem and then escape as quickly as possible. You are guaranteed that the jewel will be in the building.



In the above example, you would travel to 1, then 2, get the gem, and go back to room 1 then the entrance.

```
public class Hall {
  public final Room room1; public final Room room2;
  public final double length;
  public Hall(Room r1, Room r2, double length){
     this.room1 = r1; this.room2 = r2; this.length = length;
  }
  public Room r1(){return room1;}
  public Room r2(){return room2;}
  public double howFar(){return this.length;}
```

```
public class Room {
  public LinkedList<__________> adjacentHall;
  boolean beenTo = false;
  public boolean hasGem;
  public int order; //provides a definite ordering for the rooms, does not tell you the location, but can be used as a reference point.
  public Room(int order, boolean gem, int rooms) {
     this.order = order;
     this.hasGem = gem;
     adjacentHall = new LinkedList<Hall>();
     for(int i = 0; i < rooms; i++){
        adjacentHall.add(null);
}}}
```

```
import java.util.ArrayList; //feel free to use these imports, if not it's okay as well
import java.util.LinkedList;

public class Museum {
public int totalRooms;
public int totalHalls;
public LinkedList<_____> adj;
public Room entrance;

//Constructor
public Museum(Room[] rooms) {
  this.totalRooms = rooms.length;
  this.totalHalls = 0;
  adj = new LinkedList();
  for (_____) {

    _____

  }
  entrance = _____
}

//method for stealing from the museum
public void stealing() {
  ArrayList<Room> visited = new ArrayList<>();

  _____

  _____

}
```

```
//You are sneaking into the museum, attempting to find the jewel, this is the
hardest method of the class just because of sheer length.
public Room goIn(_____ visited) {
   Room r = entrance;
   boolean stolegem = false;

   while (_____) {

      _____

      _____

      if (r.hasGem) {

         _____

         break;
      }
      Hall minHall = _____

      for (int i = 0; _____; i++) {

         Hall curr = _____

         if (minHall _____) {

            if (curr != null &&

            _____ )) {
               minHall = curr;}
         } else if (curr != null &&

            _____ &&

            _____) {
            minHall = curr;}}

      if (_____) {
         r = minHall.to();
      } else {
         r = minHall.from();}}
   return r;}
```

```
//You have the gem now you are leaving. You can do this part without having done
the earlier method (there are no references to it; however, you should understand
what it is doing)
public void goOut(Room r, ArrayList<Room> visited) {

    int curr_lim = _____);

    while (_____) {
        Room closestroom = null;
        double dist = Double.MAX_VALUE;

        int currentlimit = _____


        for (int i = 0; i < _____; i++) {
            Room curr = visited.get(i);
            if (adj.get(r.order).size() < dist) {
                closestroom = curr;

                _____

                _____
            }
        }

      _____
    }
  }
    System.out.println("We outtie");
}
```

b) (5 pts) What is the worst case runtime of stealing in terms of rooms (r) and halls (h)? Only keep necessary terms.

5. Trick or Tree-t (12 pts). Mark true or false for the following problems, a brief justification is required. Each question is worth 3 points

a) The fastest time for any comparison based can ever be is **Ω**(Nlog(N)) regardless of input.
☐True ☐False

b) When looking through a binary search tree, if we are looking for a number of items between 2 elements, the best case runtime is Θ(N) because we will need to go through every single element to see if it fits into our range?
☐True ☐False

c) All valid left leaning Red Black Trees can become a valid Binary Search Tree by replacing all red nodes with black ones.
☐True ☐False

d) 2-3 Trees are faster than Red-Black Trees for most major operations because of the height of the 2-3 tree is smaller than the height of a Red-Black Tree.
☐True ☐False

6. The Lavender Leopard Strikes Again (20 pts). You were examining your gem after successfully stealing it only to realize that you had stolen a fake gem, and that the real Lavender Leopard is in a much more secure museum. In this museum, not every room is connected. To avoid ridiculous amounts of backtracking, you have decided to place teleporters in every room you visit. At any point, if there is any path that is closer to the entrance than the one you are taking, you will teleport to the room before and then travel to it. Once you find the gem, you need to get out as soon as possible. To get out, you will teleport in the reverse order that you arrived so you can collect all your teleporters (they aren't that cheap!). Note that Hall and Room have the same definition that they did for #4. Additionally, you **do not** need to redefine class level variables and the constructor (you can if you want to), as they are the same as the Museum class's . You have 3 pages to complete this.

If you cannot write the code, but can write the idea behind the problem, you can get up to % of the points.

Hint 1: Use old algorithms that we've gone over and see how you can modify them to match this problem

Hint 2: Look at other data structures.

```
//Optional Node class
class Node {

  _____

  _____

  public Node(_____) {

    _____

    _____

  }
}

//Method to teleport between rooms then teleport out of the museum.
 public void teleportInNOut() {

  _____

  _____

  _____

  }
```

```
//Method to steal the jewel from the museum

public void teleportSteal(_____) {

   Hall[] edgeTo = _____

   double[] distTo = _____

   PriorityQueue pq = _____

   for (_____) {

      _____

   }
   distTo[entrance.order] = 0.0;

   pq.insert(_____);

   while (_____) {

      _____

      _____

      _____

       if (_____) {
          break;
       }

       for (_____) {

          if (_____) {

             done(_____);
          }
       }
   }
}
```

```
\\Method header is not complete
public void done(Hall h, Room temp,

_____) {
      Room r;
      if (h.from().equals(temp)) {

         _____
      } else {

         _____
      }

      if (_____) {

         distTo[r.order] = distTo[temp.order] + h.length;

         edgeTo[r.order] = h;

         if (_____) {

        _____

         } else {

            _____
         }
      }
   }
}


//Method for leaving the museum

public void teleportLeave(_____) {

   while (_____) {

         _____
   }
   System.out.println("Out again");
}
```

7. D-Arranged Lab (9 pts). You are viewing a research group, unfortunately, this research group is full of some bumbling buffoons who happens to make a lot of mistakes. Each part is worth 3 points.

a) While putting his chemicals into test tubes, one of the scientists realized that he accidentally made a duplicate of one of his n chemicals, he just doesn't know which one. However, he does know that the duplicate element will be, at most, 10 test tubes away. What sort should he use to quickly find the duplicate? Assume that all the chemicals are in their natural order, except for the duplicate.

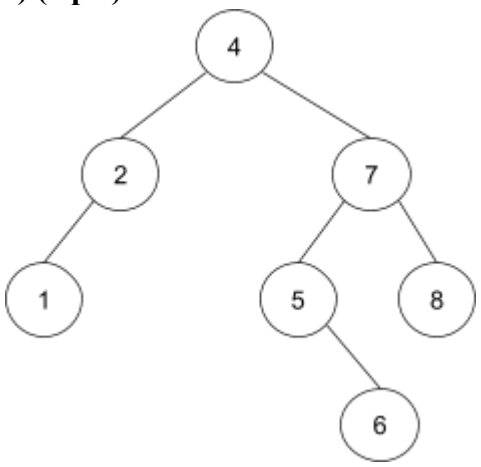
b) You have separated each of your n test tubes into k sections (so test tube 1 is now in seperated into k test tubes and so on). The only problem is that while they were all getting analyzed, you mixed them up again. Luckily, you have the name of the chemical and the time a chemical was put into a test tube. You want to sort your test tubes by chemical and then by amount of time it has been oxidizing. What sort can you use to ensure this is done as quickly as possible?

c) Now the scientists are going to input the name of their chemicals into a spreadsheet to see if they are dealing with your list of hazardous materials. The list is in alphabetical order so to easily cross reference what chemicals are dangerous, the scientists want to put the name of their chemicals in alphabetical order, what sort should they use?

8) Riddle me this:
What room can no one enter?

9. Rotations and Aberrations (10 pts).

a) (5 pts) Create the 2-3 tree that corresponds to the below Binary Search Tree.



b) (5 pts) Create a Red-Black Tree that corresponds to the 2-3 tree/Binary Search Tree in part a.

10. JJJJ UNIT (15 pts).
You are starting your rapping career as a part of the prestigious rapping group J Unit. You want to recruit new members so you can have a hip possi. To prove their worth to your group, you have decided that you will test their vocabulary. To do this, you will make a TrieMap. This will make a Trie, but for each word, you also have a note to see how many times it has been used. If any word has been used more than 5 times, the person will not be able to join your group.
You are given the TrieNode class defined as follows:

```
public class TrieNode {
   char c;
   boolean isWord;
   String word;
   TrieNode[] children = new TrieNode[26];
   int count;}
```

Additionally, you are given that the TrieMap has a root class level variable and an **addWord** and **contains** method.
*The LinkedList has an addAll method which adds all items from a collection to the end of the list (e.g. LinkedList A = {A,B,C}, LinkedList B = {D, E,F}. A.addAll(B), A = {A,B,C,D,E,F}

Fill in the below methods

```
public LinkedList getAllWords(TrieNode start) {
```

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

```
public static boolean testCount(TrieMap t){

_____

_____

_____

_____

_____

_____

_____

_____

_____
}
```

11. Median Heap (20 pts).

We have earlier dealt with Min Heaps and Max Heaps, now we are going to explore a new type of heap, the **Median Heap**. The Median Heap will have 3 methods that you will need to fill in, **median**, which finds the median element, **insert**, which inserts an element into the Median Heap, and **balanceHeap**, which makes the heap balanced. You may need to use some of these methods inside of others. Feel free to use any data structure that we have gone over.

```
public class MedianHeap {

   private _____ top = new _____

   private _____ bottom = new _____

   //find the median element, if there are an even amount of elements, take the
average of the 2 elements closest to the middle
   public int median() {

      int minSize = _____

      int maxSize = _____

      if (_____) {

        _____
      }

      if (_____) {

         _____

      }
      if (_____) {

         _____

      }
      return _____

   }
```

```
//used to insert elements into the MedianHeap
public void insert(int element) {
```

_____

_____

_____

_____

_____

_____

_____

_____

```
balanceHeap();

}
```

```
 //Balance the heap
private void balanceHeap() {
```

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

```
}
```

12. Faster than a Speeding Runtime (17 pts)
Fill in the runtimes for each of the following operations. If there is no tight bound, provide a lower and upper bound. Provide a brief explanation as to why the runtime is what it is.

a) (5 pts) How much time does it take to create a heapify an array of N elements?

b) (3 pts) How much time does it take to find the minimum element in a Binary Search Tree

c) (3 pts) How much time does it take to place all the contents of a given MinHeap into an array that is ordered from least to greatest.

d) (3 pts) How much time does it take to find the minimum value in a MaxHeap.

e) (3 pts) Find the runtime of getting all the words from a trie and then using MSD sort on them.

13) Future-Roaming (30 points).
You are a croissant delivery boy in the 31st century. You want to deliver your croissants as quickly as possible; however, because you lost your client's addresses and they live all across the galaxy, so you don't want to go door to door. You have the phone numbers of all the people who ordered croissants from you, so you will attempt to use this to figure out in where you should go. Each phone number is about 10000000000000000000000 digits long and you have a total of N clients. You will use the following information about telephone numbers to help:

1. The sector code (the leftmost 10 digits) will allow you to figure out which sector a client is in. On average, there are a total of $\sqrt{N}$ of households you need to deliver to in the sectors.
2. In the 31st century, telephone numbers have a cool feature where the difference between the last 4 digits of any 2 telephone numbers in the same sector is equal to the distance between their two corresponding households.
3. You know the address of 1 of the houses you need to go in each sector.

You want to go to sectors based off how many of your clients are in them and how close it is . If a sector has 5 clients and is 2 Megadistances away, it is more appealing than a sector that has 1 client and is 1 Megadistance away. If a sector has 6 clients and is 2 Megadistances away, it is less appealing than a sector with 9 clients and 3 Megadistances. If a sector has 4 clients and it is 2 Megadistances away, it is considered equal to a sector that has 2 clients that is 1 MegaDistance away, at this point, you just pick which one to go to randomly . You will deliver to all the households in that sector before going to the next sector. You also know how many megadistances are between each sector, and you can assume that you will start at a location that is equidistant from all the sectors.

Provide an implementation where figuring out which households are in which sector in **Θ(N)** time, figuring out the distance between a house and all the other houses in a sector, must take, on average,**O(** $\sqrt{N}$ **)** time, and delivering croissants to all the households on average, **O(** $N^{\frac{3}{2}} log\sqrt{N}$ **)** time.

Assume that the time it takes you to walk 1 Minidistance (inside of a sector) is the same amount of time it takes your spaceship to travel one Megadistance (between sectors). Justify why your design works in the given time.