GOEL'S REVIEW OF ADVANCED COMPUTER SCIENCE

For ALL Levels

1st Edition

Anish Goel

Most Up to Date Version Yet!


Goel

GOEL'S

Advanced Computer Science

A Clear Review of the Concepts, Materials, and Code

1st Edition

Anish Goel
Current Student, Cypress Ranch High School
Cypress, Texas, United States of America
Support me at: paypal.me/GoelAnish

Notes

Table of Contents

6. Stacks	Stack Push and Pop Connect to Queues Connect to LinkedList	
7. Queues	Queue Enque and Deque Priority Queues Connect to Stacks Connect to LinkedList	
8. LinkedList	Nodes Connect to Stacks Connect to Queues Methods	
9. Trees	Nodes Transversal Binary and other types	
10. Greedy Algorithms and Graphs	Adjacency List/Matrix Graph BFS DFS Dijkstra	
11. Test	Test of all materials Expertise measure	

Preface

This book has been written to provide a proper summary of advanced computer programming algorithms, data structures, and a guide into the mindset of an advanced computer programmer. This book will contain code and ideas meant to be implemented in Java, however the ideas can be taken and implemented into any programming language. For those reading this book to learn new concepts, as well as those who are trying review topics they are unsure on, space marked for notes, as well as the margins are available to jot down short notes for future reference.

This book may contain for its algorithms and data structures:

1. The code for the algorithm or data structure
2. An explanation on how it works
3. Questions to test your understanding of the material consisting of:
4. A concluding review of each topic

The expectation of someone reading this book is that they are capable in coding as the material will move rather quickly. You are expected to have mastery of simpler material such as ArrayLists, Arrays, Strings, and other primitive data types. One using this book should also know how to create methods and classes in Java as well as techniques such as recursion.

The code in the book will be based on what you will be expected to produce in Netbeans, a free programming software that can be downloaded at:

https://netbeans.org/downloads/

Number Systems

Number systems are a method to organize and understand numbers. There are multiple bases, methods, and operations available to manipulate numbers into forms better suited for your needs. While there are methods to manipulate these number systems into negative numbers such as ones-complement and twos-complement, this textbook will not cover that.

Different Bases:

A base is the what a digit is multiplied by to the power of its place to get the number. For example, the number "1234", is $(4x10^0) + (3x10^1) + (2x10^2) + (1x10^4)$. An equation to remember this is

"Number = Digit x Base^Place".

By replacing the ten with a different number, the number changes and is placed into a separate base. However, it is important to remember that the base must be larger than the largest digit. For example, base 6 cannot have any number above 6, including 6.

There are infinitely many bases, but the ones of importance to learn are base 2, base 10, and base 16. While knowing all bases is helpful, these are the 3 bases you will come into contact the most.

An example of base 2 looks like 1010 1010, it is common to separate it in chunks of four for easier comprehension. An example to base 10 is 1234, this is the common number system in use today. An example of base 16 looks like 19AA4, the range is 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F representing the numbers 0 through 15.

Converting a Base:

In order to change a base, the simplest way is to convert it into base 10, then convert it from base 10 into the ending base. If you start or end in base 10, the job becomes significantly easier to accomplish.

To start take use the equation highlighted above and it will solve the number into base 10. Be sure that you apply this equation to all numbers and add them up. This will result in a number being converted into base 10.

Then once in base 10, to convert into any base of your choosing, you must divide the base 10 number by the base. The first divide's remainder becomes the first digit on the right. The second divide will take the quotient of the code and divide it by the

base again, the remainder being used as the second number from the right. This will repeat until the quotient is 0.

An example provided will use the number “163” in base 7 will be converted to base 11. To start, “163” will be converted to base 10. The steps are:

$$(3x7^0) + (6x7^1) + (1x7^2) = 94$$

1^{st} divide: 94/11 = 8 R 6

2^{nd} divide: 8/11 = 0 R 8

163 in base 7 is 94 in base 10 is 86 in base 11

In programming base changes can be done easily through a method:

“Integer.parseInt(“insert number here”);”

This line can come in handy in many situations and easily converts a number in the form of a string into base 10. If the number is in a different base, add a comma after the closing quotation marks and add which base the number is from.

There are examples as to how the method works provided below.

```
Integer.parseInt(“20”);   // 20
Integer.parseInt(“+20”); // 20
Integer.parseInt(“-20”);  // -20
Integer.parseInt(“2247483649”); // NumberFormatException
Integer.parseInt(“20”, 16); // 32, the base 10 value of 20 in base 16
```

To convert a base from base 10 to another base, use:

“Integer.toString(number to be converted, new base)”

This line will change a base and return in it the form of a String. Its parameters are both ints. To convert the String returned into a int, simply use Integer.parseInt();

There are examples as to how the method works provided below.

```
Integer.toString(15, 2);   // 1111
Integer.parseInt(+15, 2); // 1111
Integer.parseInt(-15, 2);  // -1111
Integer.parseInt(15, 16); // f
```

Bitwise Operations:

There are multiple operations that can be done on numbers in order to manipulate them.

()	**HIGH**
! ++ --	
*** / %**	
+ -	
<< >> (bitwise shifts)	
< <= > >=	
== !=	
& (bitwise and)	
^ (bitwise xor)	
| (bitwise or)	
&&	
||	
= += -= *= /= %=	
,	**LOW**

The chart above lists all the operations and the order they will happen in. The following list is of the operators: their names, and their function as well as an example on their use. They will be in the order of operations from high to low. This will only include ones that can be truly helpful when coding with different bases, binary in particular.

1. Parentheses
    a. These function to separate certain sections away from the rest of the order of operations and prioritizes them to be completed first. The

inside of the parentheses still follows the order of operations, but its contents will take place prior to the contents of the outside. They may be placed inside another set of parentheses. They also may be used to help a reader better understand the code by providing clear divisions through separating chunks of code.

b. Two examples are provided below

"(15 & 3) == (1 | 2)", if this code did not have the parentheses, the result would be false as the "==" has a higher priority than the & or the |. However, the parentheses allow the & and the | to happen first, changing the result to true.

i. "(3 ^ 1) & 2", if this code did not have the parentheses, the result would be 3, as the & has a higher priority than the ^. With the use of the parentheses the result becomes 2.

2. Not, increment, and decrement
   a. "Not" functions to put the opposite of the contents out. This will take the number, in binary, and change all the zeros to ones and all the ones to zeros. Increment will add 1 to the number, however, it may cause an overflow into the largest negative number. Decrement will subtract 1 from the number, however, it may cause an overflow into the largest positive number. You can increment and decrement prior or after the number.
   b. Two examples are provided below for each type
      i. Not
         1. "!(15)" this will convert 15 to binary which is 1111 and replace all the ones with zeros resulting in the output being 0.

2. "!(2)" this will convert 2 to binary which is 0010 and replace all the ones with zeroes and all the zeros with ones resulting in the output being 13.

ii. Increment

1. "15++" this will increment the number by 1, and the output is 16.
2. "(int) ++2147483647" this will increment by one, but as it is an int, it will overflow and become -2147483647.

iii. Decrement

1. "15--" this will decrement the number by 1, and the output is 14.
2. "(int) -2147483647--" this will decrement by one, but as it is an int, it will overflow and become 2147483647.

3. Bitwise Shifts
   a. There are two kinds of Bitwise shifts that divide or multiply by a power of two. ">>" will divide the first the number by 2 to the power of the second number and "<<" will do the opposite.
   b. Examples:
      i. "16 >> 2" will equal 4 because 2 to the power of 2 will be 4. 16 divided by 4 is 4, and the output will be 4.
      ii. "3 << 3" will equal 24 because 2 to the power of 3 is 8 and 8 times 3 is 24.
4. Bitwise And
   a. A bitwise and will look at both numbers in binary and for each digit and place a one only if both numbers have ones in that place.
   b. Examples

        i. “5 & 3” will equal 1 because 5 is 101 and 3 is 11 and since 5 doesn’t have a 1 in the second place and 3 doesn’t have a 1 in the 3rd place, the only common 1 is the 1 in the first place. As a result, the answer is 1.

5. Bitwise Exclusive Or
    a. A bitwise Exclusive Or is similar to how a bitwise And functions because they check both numbers in binary, but an exclusive Or will take the one only if only one number has a digit in that place.
    b. Examples
        i. “5 ^ 3” will equal 2 because 5 is 101 and 3 is 11 and since 5 doesn’t have a 1 in the second place and 3 has a 1 in the 2nd place, the only non-common 1 is the 1 in the 2nd place. As a result, the answer is 2.
6. Bitwise Or
    a. A bitwise Or is similar to a bitwise exclusive Or except that if both numbers have a 1 in a place, it will still accept the 1.
    b. Examples
        i. “5 | 3” will equal 7 because 5 is 101 and 3 is 11 and since 5 doesn’t have a 1 in the second place and 3 has a 1 in the 2nd place, the numbers get placed together as 111. As a result, the answer is 7.

These are all of the critically important operations to know for number systems. Mastery of all operations will be of great use in coding, but the ones listed above will be commonly seen and tested and needed.

Questions:

1. Change 176 in base 10 to binary. Show all steps below:

2. What is 1011 in base 2 + 45 in base 8? Show all steps below:

3. How would you change the base of 1000 in base 10 to base 2 in java? Explain:

4. Write a method to change a base 10 number to binary.

5. Which of the following is not a valid digit in hexadecimal?
   A) 7
   B) 9
   C) B
   D) H

Review:

In this unit, you have learned:

1. What a base is
2. How to convert from one base to another by hand
3. How to convert from one base to base 10 in Java
4. How to convert from base 10 to another base in Java
5. What a bitwise operation is
6. What order operations are in
7. The critical operations for bitwise functions

You were later tested over this material and expected to show a strong understanding of the material. If you did not do well on the questions, it is **HIGHLY RECOMMENDED** that you go and review the topics covered again. After you feel that you have properly prepared yourself on the topic feel free to move on to the next topic.

Use the space provided for your own notes

Divide and Conquer

Divide and Conquer is when one takes a task and breaks it up into smaller parts and then solves them individually. This is most commonly used in sorting, because some sorts without it can have absurdly long runtimes while with Divide and Conquer can be NLogN.

This method of coding is also used in problems such as the Convex Hull problem or the Closest Points problem. These problems would take a long time to do without divide and conquer which makes them far more efficient. These problems arise often in code, and it is recommended to do research and learn about how to solve them.

The two sorts of interest are called Quick Sort and Merge Sort, and these will be studied below.

Quick Sort:

The code for Quick Sort is split in two methods and requires both parts to function. One method is called for the sort to take place, and the second method is a helper method that should only be called by the sort method.

The worst case scenario is the code will run at O(n^2) but it will usually run O(nLogn) which is very fast. The likelihood it will run at O(n^2) is very small so it is usually not worth worrying about.

Quick Sort in its general form doesn't require any extra storage whereas merge sort requires O(n) extra storage. Allocating the extra space used for merge sort increases the run time of the algorithm. Both types of sorts have an O(nlogn) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(n) storage space.

Code:

```
public void sort(int arr[], int low, int high) {
        if (low < high) {
            // pi is partitioning index, arr[pi] is now at right place
            int pi = partition(arr, low, high);
            // Recursively sort elements before and after partition
            sort(arr, low, pi-1);
            sort(arr, pi+1, high);
        }
}
```

```
private int partition(int arr[], int low, int high) {
        int pivot = arr[high];
        int i = (low-1); // index of smaller element
        for (int j = low; j < high; j++) {
            // If current element is smaller than or equal to pivot
            if (arr[j] <= pivot) {
                i++;
                // swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }


        // swap arr[i+1] and arr[high] (or pivot)
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;

        return i+1;
    }
```

Explanation:

This sort relies on three steps to operate:

1. Choose an element, called pivot, from the list. Pivot should be the middle element.

2. Re-order the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it. After this partitioning, the pivot is in its final position. This is called the partition operation.

3. Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

4. Profit.

Merge Sort:

Merge Sort is useful for sorting linked lists in O(nLogn) time. Unlike arrays, linked list nodes may not be adjacent in memory and we can insert items in the middle in O(1). Therefore, merge sort can be implemented without extra space for linked lists.

Unlike arrays, we cannot do random access in linked list that Quick Sort requires. In linked list to access Nth index, we have to travel every node from the first to Nth node as the objects aren't in order. Therefore, the overhead increases for quick sort while merge sort accesses data sequentially, allowing merge sort to be faster.

Code:

```
    public void sort(int arr[], int l, int r) {
        if (l < r)  {
            // middle point
            int m = (l+r)/2;

            // Sort first and second halves
            sort(arr, l, m);
            sort(arr , m+1, r);

            // Merge the sorted halves
            merge(arr, l, m, r);
        }
    }
 private void merge(int arr[], int l, int m, int r) {
        int n1 = m - l + 1;
        int n2 = r - m;
        int i = 0, j = 0;
        int k = l;
        int L[] = new int [n1];
        int R[] = new int [n2];

        for (int i=0; i<n1; ++i) {
            L[i] = arr[l + i];
        }
        for (int j=0; j<n2; ++j) {
            R[j] = arr[m + 1+ j];
        }
```

```
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
```

Explanation:

The code for Merge sort is split into 2 parts:

1. Split the list into a smaller sublist at the middle and recursively call the method until it cannot be split anymore.

2. Merge the small sublists together until only one sorted list is left.

For the first step it is crucial to add a case where if the leftmost point is after the rightmost point, to stop. When calling the method, use 0 as the left most point and the length for the rightmost point.

Questions:

1. What is the optimal, average, and best runtimes for Quick Sort?

2. Which sort requires more memory when sorting arrays: Merge Sort or Quick Sort and how much?

3. What sort is best for sorting linked lists, and what property of linked lists makes it optimal?

4. Notate how Quick Sort would sort the array, {12, 1, 43, 6, 9999, -5}, and show your steps.

5. Notate how Merge Sort would sort the array, {12, 1, 43, 6, 9999, -5}, and show your steps.

Review:

In this unit, you have learned:

1. What Divide and Conquer is
2. Why one would use Divide and Conquer techniques
3. The overall idea of implementing Divide and Conquer as well as in
4. What Quick Sort is and its runtimes
5. What Quick Sort is optimal for and why
6. What Merge Sort is and its runtimes
7. What Merge Sort is optimal for and why

You were later tested over this material and expected to show a strong understanding of the material. If you did not do well on the questions, it is **HIGHLY RECOMMENDED** that you go and review the topics covered again. After you feel that you have properly prepared yourself on the topic feel free to move on to the next topic.

Use the space provided for your own notes

Sets

Set is an interface which implements Collection. It can be instantiated through a HashSet or a TreeSet. These sets are similar to ArrayList, except that they cannot hold duplicates of a variable. The below is a table of example instantiations.

HashSet	TreeSet
Set<T> set = new HashSet<T>();	Set<T> set = new TreeSet<T>();
HashSet<T> set = new HashSet<T>();	TreeSet<T> set = new TreeSet<T>();
Set<T> set = new HashSet<>();	Set<T> set = new TreeSet<>();
HashSet<T> set = new HashSet<>();	TreeSet<T> set = new TreeSet<>();

DO NOT DO WHAT IS BELOW!!! THIS IS INCORRECT!!!

Set<T> set = new Set<T>();
TreeSet<T> set = new HashSet<T>();

The major difference between a HashSet and a TreeSet is that a TreeSet is sorted. It is crucial to learn the runtimes of these objects so you can always choose the best one.

Method	TreeSet	HashSet
Add	$O(Log_2N)$	O(1)
Remove	$O(Log_2N)$	O(1)
Contains	$O(Log_2N)$	O(1)

Methods:

Name	Use
add(x)	adds item x to the set
remove(x)	removes an item from the set
contains(x)	returns if x is in the set
clear()	removes all items from the set
size()	returns the # of items in the set

Add:

The add method will take the variable x and add it to the set. If the add is successful, it will return true. If the set already contains a variable that is the same as x, it will return false.

Remove:
The remove method will take the variable x and remove it from the set. If the set contained the object, it will return true. If the set did not have an object that matched x, it would return false, and the set would not be changed.

Contains:
The contains method will take the variable x and check if the set contains it. It will then return true if the set contains x, or false if the set does not contain x.

Clear:
This method will remove all the variables from the set and leave it completely empty. It does not return anything.

Size:
This method will check the number of variables in the set. It will return the number in the form of an int.

HashSet:

HashSet gives better performance than TreeSet for the operations like add, remove, contains, size etc. HashSet offers constant time cost while TreeSet offers Log(N) time cost for such operations. As a result, it is usually better to use a HashSet over a TreeSet unless order matters.

A HashSet has HashTables in its definition.

One critical feature the HashSet has over the TreeSet is that HashSet will allow null objects inside while a TreeSet will not.

TreeSet:

TreeSet elements are sorted in ascending order by default while HashSet does not maintain any order of elements. As a result, it is good to use TreeSet to avoid sorting the HashSet.

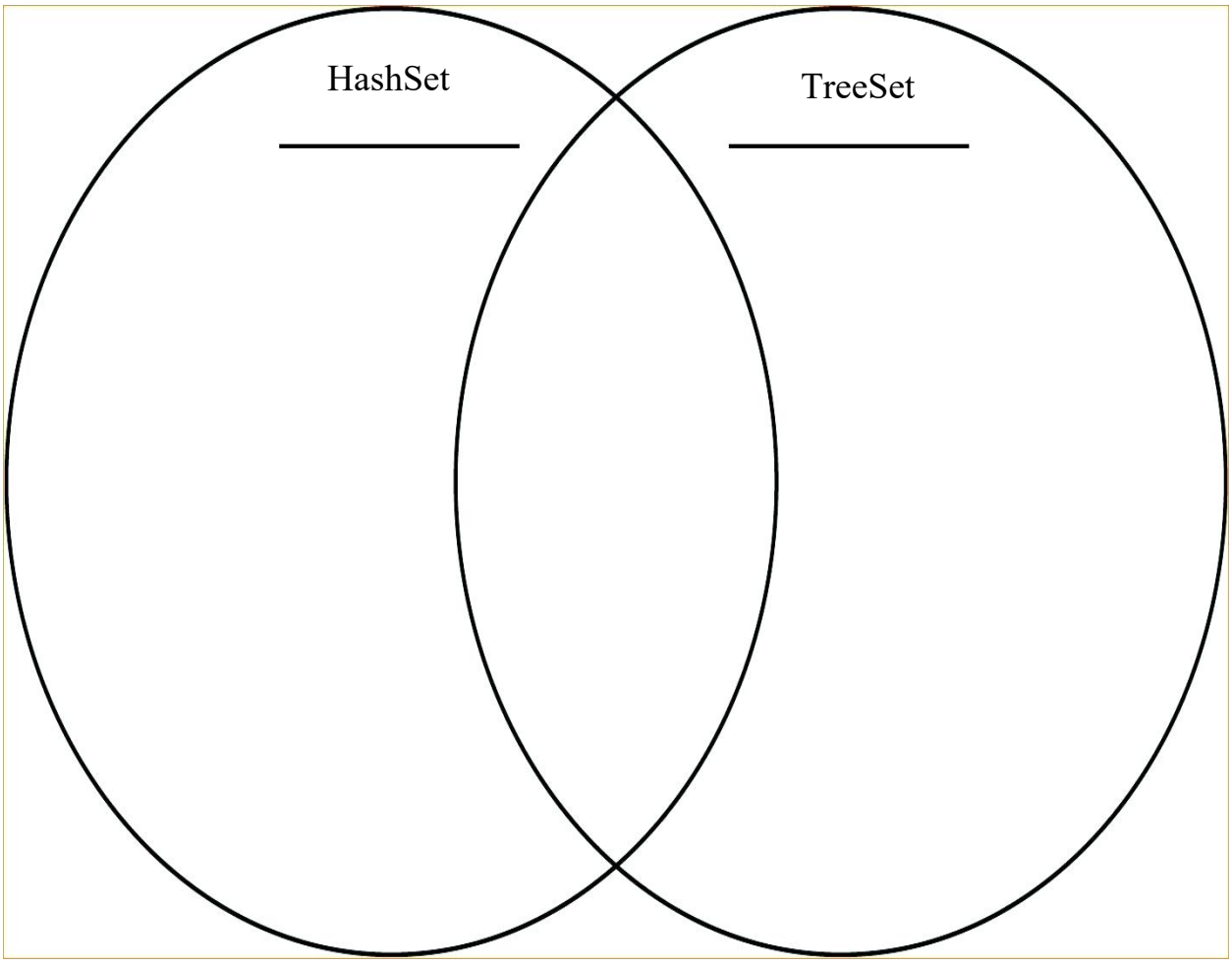
A TreeSet has Binary Trees in its definition.

A TreeSet will not accept null values and will instead throw a null pointer exception because it calls the compareTo() method in order to maintain its order.

Questions:

1. Which Set is optimal for use when efficiency is of highest priority and order is irrelevant?

2. Complete the Venn Diagram:



3. What is HashSet defined of and what is TreeSet defined of?

Review:

In this unit, you have learned:

1. What a Set is and what it is defined by
2. How to instantiate a Set and how not to
3. What the two types of Sets are (HashSet and TreeSet)
4. The main methods used in a Set
5. What the runtimes are for the different Sets
6. What the two Sets are made of (coded with)
7. Why some Sets are superior than the other in certain scenarios

You were later tested over this material and expected to show a strong understanding of the material. If you did not do well on the questions, it is **HIGHLY RECOMMENDED** that you go and review the topics covered again. After you feel that you have properly prepared yourself on the topic feel free to move on to the next topic.

Use the space provided for your own notes

Maps

A Map is an interface similar to a Set. A Map actually uses Sets in its definition. However, a Map does not implement Collections, instead it implements Java Collections Framework. Further, a map will hold 2 different values, one a key and one the value.

A Map has 2 main classifications: a HashMap and a TreeMap. They can be instantiated as shown below:

HashMap	TreeMap
Map<K,V> map = new HashMap<K,V>();	Map <K,V> map = new TreeMap <K,V>();
HashMap<K,V> map = new HashMap<K,V>();	TreeMap<K,V> map = new TreeMap <K,V>();
Map <K,V> map = new HashMap <>();	Map <K,V> map = new TreeMap <>();
HashMap <K,V> map = new HashMap <>();	TreeMap <K,V> map = new TreeMap <>();

The major difference between a HashMap and a TreeMap is that a TreeMap is sorted. It is crucial to learn the runtimes of these objects so you can always choose the best one.

Method	TreeMap	HashMap
put	O(Log2N)	O(1)
get	O(Log2N)	O(1)
containsKey	O(Log2N)	O(1)

Methods:

Name	Use
put(K,V)	adds the <K,V> pair to the map
get(K)	gets the value for key
clear()	removes all items
size()	returns the # of items

keySet()	returns a set of all keys
containsKey(x)	checks if key is in the map

Put:
This method is the equivalent of the add method. It will take K – the key – and create an entry for it in the map. This entry will house the K and the V – the value. The key will be the only way to get to the value from that point on.

Get:
This method requires the key value to operate. From there it will take the key and check what value the key has. Once it has the value, it will return it for the client code to use.

Clear:
This method will remove all the variables from the map and leave it completely empty. It does not return anything.

Size:
This method will check the number of variables in the map. It will return the number in the form of an int.

KeySet:
This method will return a set of all the keys in the map. This is useful as it helps checking what keys exist, and is useful when iterating through the map.

ContainsKey:
This method will take a variable x and return true if it contains an entry where x is the key, or false if x is not a valid key. This is useful for debugging, especially in scenarios where an end user will not access the client code.

HashMap:

A HashMap is defined using a HashTable, just like a HashSet.

A HashMap is a map ordered by each item's hash that is extremely time efficient.

It works by placing items into a large array through a formula, if there is a collision, it chains or buckets the variable using a linked list. Because of this equation, searching is very efficient, as it just solves for where the entry should be and goes to that location.

A HashMap is beneficial as it is faster and also can accept null values. It can only accept a single null key however as duplicate keys are not allowed.

TreeMap:

A TreeMap is defined using a Binary Tree, just like a TreeSet.

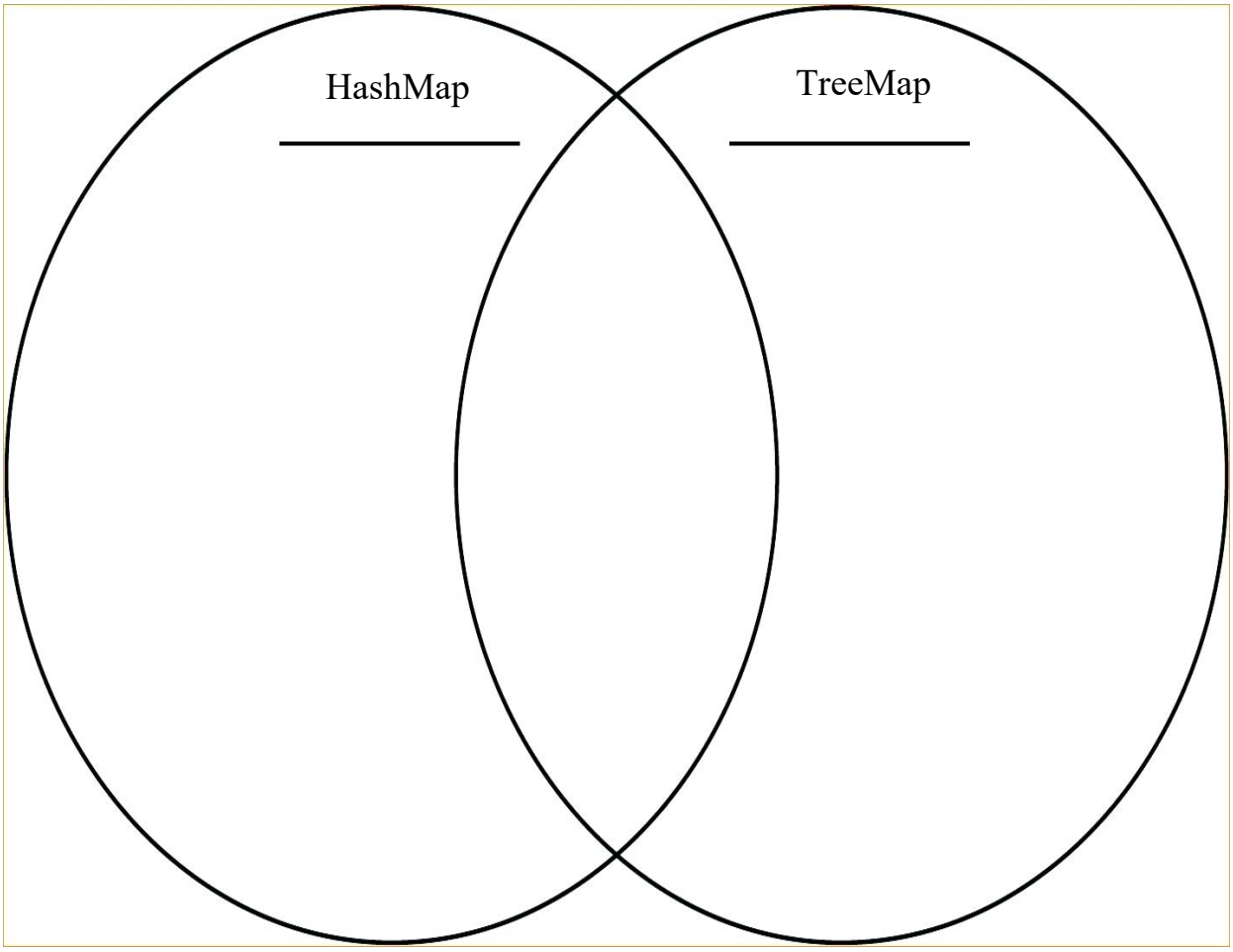
A TreeMap is a naturally ordered map that is very efficient, but not as efficient as HashMap. It works by having a node with a data entry, and a left and right reference.

As a result, it is able to place the central node and place objects moving left and right from the node to end node. It also uses this to quickly navigate to the data entry once given a key as it does not have to check every possible point.

TreeMap is beneficial as it is sorted. It cannot accept null keys because it cannot sort the null. However, it can hold a null value.

Questions:

1. Complete the Venn Diagram



2. What is HashMap defined of and what is TreeMap defined of? Elaborate.

3. What does the Map interface implement: Collections or Java Collections Framework? How do you know and what does this mean for the interface?

Review:

In this unit, you have learned:

1. What a Map is
2. How to instantiate a Map
3. The two types of Maps and their benefits
4. The commonly used methods in a Map
5. What a HashMap is and how it is defined
6. What a TreeMap is and how it is defined
7. What the differences and similarities in the two types of Maps are

You were later tested over this material and expected to show a strong understanding of the material. If you did not do well on the questions, it is **HIGHLY RECOMMENDED** that you go and review the topics covered again. After you feel that you have properly prepared yourself on the topic feel free to move on to the next topic.

Use the space provided for your own notes

Stacks

A stack is a data type where items are added on top of the existing stack and removed in that order. If an item was added to the stack, the next value would be the next added value. It uses different method names than most other data types in java, and as a result, can be a little strange for new users.

Stacks are made of singly linked lists, so it is not possible to go back and forth through the stack. There are also array-based stacks, but those are not as likely to be found or used as a singly linked list defined stack.

An iterable stack can be made, but that defeats the purpose of a stack in most scenarios as an arraylist would work just as well for that reason.

A stack can be best instantiated like:

```
Stack<T> stack = new Stack<>();
```

Methods:

Name	Use
push(x)	adds item x to the stack
pop()	removes and returns an item
peek()	returns the top item with no remove
size()	returns the # of items in the stack
isEmpty()	checks to see if the stack is empty

Push:

This will take a singular value and add it to the top of the stack. It will stay on the top until removed or a new value is added on top. The size variable is then increased but the method does not return anything.

Pop:

This will take the top value of the stack and remove it. Then it will decrease the size variable and return the removed item.

Peek:

This will take the top value of the stack and return it. It does not change the size variable or remove the top variable, instead keeping the stack exactly the same.

Size:
This returns the size variable as an int. This method is O(1) because the size is kept as a variable throughout the code.

isEmpty:
This method checks if the stack has any contents. If it does not, it returns true, else it returns false.

Example:

There are many problems that can be more easily solved with stacks such as postfix and expression solver. The example problem in this textbook will be postfix, because that problem is often put on the written portion of computer programming tests.

Problem:
Let the expression be "2 3 1 * + 9 –". We scan all elements one by one.

Scan '2', it's a number, so push it to stack. Stack contains '2'. Scan '3' and push it to stack, stack now contains [2,3]. Scan '1' and push it to stack, stack now contains [2,3,1]. Scan '*', it's an operator, pop two numbers from stack, use the * operator, we get 3*1 which results in 3. We push the result '3' to stack. Stack now becomes [2,3]. Scan '+', pop two numbers from stack, use the +, we get 3 + 2 which results in 5. We push the result '5' to stack. Stack now becomes [5]. Scan '9', and push it to the stack. Stack now becomes [5,9]. Scan '- ', pop two numbers from the stack, use the '– ', we get 5 – 9 which results in -4. We push the result '-4' to stack. Stack now becomes [-4]. There are no more elements to scan, we pop the top element from stack (which must be the only element left in stack).

The code provided on the page below can be used in order to perform these calculations. It requires a string called "exp" which will be the expression. In the example above would be "2 3 1 * + 9 –".

Use the following page to write your answer.

Code:

```
Stack<Integer> stack=new Stack<>();
for(int i = 0; i < exp.length(); i++) { // goes through expression length
        char c = exp.charAt(i); // capture char
        if(Character.isDigit(c)){
            stack.push(c - '0');
        } else {
            int n1 = stack.pop();
            int n2 = stack.pop();
            switch(c) { // do the operation
                case '+': stack.push(n2+n1);break;
                case '-': stack.push(n2- n1);break;
                case '/': stack.push(n2/n1);break;
                case '*': stack.push(n2*n1);break;
            }
        }
    }
    return stack.pop();
  }
```

Explanation:

This code works in 4 steps.

1. Capture the next value, and check if it is a digit or not. It captures the value as a char and uses the Character class to check if it is a digit.
2. It then splits into two different parts:
    a. If it is a digit, it is added to the stack. Then calls the next variable, returning to step one.
    b. If it is an operator, it goes to step 3
3. 2 variables are popped from the top of the stack. It then does the operation to the variables, checking for the right operation through a switch. It then goes back to step 1.
4. Finally, when all variables are exhausted, the stack pops the final value which is the answer.

Questions:

1. What is the end result of the postfix expression “4 9 – 7 8 * +”? Show all steps.

2. Instantiate a stack correctly. Write 3 ways that are incorrect and explain.

3. What is the method name to check what the top of a stack is without removing it?

4. What two types of stacks are most common? (Hint: What are they defined by?)

5. What is the runtime for the size method and why?

Review:

In this unit, you have learned:

1. What a stack is
2. How to instantiate a Stack
3. The two types of stack and their definition
4. The commonly used methods in a stack
5. What a postfix expression is
6. How to solve a postfix expression on paper
7. How to solve a postfix expression with code

You were later tested over this material and expected to show a strong understanding of the material. If you did not do well on the questions, it is **HIGHLY RECOMMENDED** that you go and review the topics covered again. After you feel that you have properly prepared yourself on the topic feel free to move on to the next topic.

Use the space provided for your own notes

Queues

There are two different types of queues. One is the standard Queue based on the queue interface. The other is a variant called Priority Queue that operates differently. Both of these are important to know and are very useful to know about, so both will be covered in this textbook.

Queue:

A queue is a data type where items are added to the back of the queue. They are removed from the front of the queue. Therefore, the first item added to a queue will be the first item removed from the queue. This style is commonly referred to as a first in first out method, or FIFO for short.

The LinkedList class implements the Queue interface. A LinkedList is a data structure not yet covered. It will be covered in the next chapter. What is necessary to know about a LinkedList in this chapter is that it is a collection of points that have arrows pointing at a different node, and those are used to go through all the points(each point being a data value).

Since a queue is an interface, do not try and instantiate the queue on its own like:

```
Queue<T> queue = new Queue<>();
```

This is wrong and will throw an error. Double check that you have instantiated this method correctly or else you are likely to have major problems in your code.

When making your own queue, try and keep the queue as an interface so it will be easier to understand for others, as well as your future self. 2 years down the line, you will be far more experienced at coding, and when you look back at your code, you will be incredibly confused if you do not create a strong foundation now.

If you want to instantiate a queue, the best way is to do:

```
Queue<T> queue = new LinkedList<>();
```

Methods:

Name	Use
add(x)	adds item x to the queue
remove()	removes and returns front item
peek()	returns the front item with no remove

size()	returns the # of items in the queue
isEmpty()	checks to see if the queue is empty

Add:
The add method will add a variable to the end of the queue. This will make it the last variable to be returned and the size is increased by one. It returns true or false depending on if the action was successful.

Remove:
The remove method will remove the front variable of the queue. It will then return the variable removed. The size is then decreased by one. This method may also be called a dequeue by other programmers.

Peek:
The peek method checks the front item of the queue and returns it. It does not remove the front item, and keeps the queue completely the same.

Size:
The size method will return the size variable. This method is O(1) because the size variable is kept updated in other methods.

IsEmpty:
The isEmpty method checks if the queue has any variables stored inside. If the queue has a variable inside the isEmpty method returns false, otherwise the method will return true.

Priority Queue:

A Priority Queue works by sorting data in a specified manner. In Java, the Priority Queue class is a min heap, which means it sorts returns the smallest variables first. In order to sort the items it is passed, it uses Comparable. In order to instantiate a Priority Queue use:

```
PriorityQueue<T> priorityQueue = new PriorityQueue<>();
```

Methods:

Name	Use
add(x)	adds item x to the queue

remove()	removes and returns front item
peek()	returns the front item with no remove
size()	returns the # of items in the queue
isEmpty()	checks to see if the queue is empty

Add:
The add method will add a variable to the queue. It will get sorted and return true if the add succeeded, false if the add fails. The add will fail if you pass it a data type it should not receive.

Remove:
The remove method will remove the front variable of the queue. It will then return the variable removed. This method may also be called a dequeue by other programmers.

Peek:
The peek method checks the front item of the queue and returns it. It does not remove the front item, and keeps the queue completely the same.

Size:
The size method will return the size of the queue. This is really useful for creating an iterable priority queue, so make sure you have a strong understanding of this method.

IsEmpty:
The isEmpty method checks if the queue has any variables stored inside. If the queue has a variable inside the isEmpty method returns false, otherwise the method will return true.

Questions:

1. What are the differences between a priority queue and a regular queue? What are the similarities between a priority queue and a regular queue?

2. Instantiate a queue correctly and explain why each part is important.

3. What is the method name to check what the top of a queue is without removing it?

4. What is a priority queue defined by? What is a regular queue defined by?

5. Which type of queue has a slower add method runtime?

Review:

In this unit, you have learned:

1. What a queue is
2. What a priority queue is
3. How to instantiate a queue
4. How to instantiate a priority queue
5. The two types of queues and their definition
6. What methods the queues have and what they do
7. The very basics of a linkedlist

You were later tested over this material and expected to show a strong understanding of the material. If you did not do well on the questions, it is **HIGHLY RECOMMENDED** that you go and review the topics covered again. After you feel that you have properly prepared yourself on the topic feel free to move on to the next topic.

Use the space provided for your own notes

LinkedList

A LinkedList is a data structure made of multiple data values called nodes. These "nodes" each contain at least one pointer and a data value that is used. A LinkedList class contains a head variable that is the first node, as well as methods for manipulating the LinkedList. The node class contains only the data, pointers, and occasionally a constructor.

There are two types of LinkedLists, a singly LinkedList and a doubly LinkedList. A doubly LinkedList would have the optional pointer implemented. The difference is that a doubly LinkedList would be more efficient as it allows for adding to the end to be O(1). It also allows the LinkedList to be made cyclical, where if one was to keep calling for the next node, they would eventually be looped back – not necessarily to the beginning – and allowed to continue forever. Java's LinkedList class is a doubly LinkedList.

Methods:

Name	Use
add(x)	adds item x to the list (2 types)
get(x)	get the item at location x
size()	returns the # of items in the list
remove()	removes an item from the list (3 types)
clear()	removes all items from the list

Add (Type 1):

It only takes a variable and will add it to the end. The first type will return a Boolean.

Add (Type 2):

The second will take both a variable and a number for the index. The variable will be added into the location and nothing will be returned.

Get:

This takes in a number for location. It will iterate through the LinkedList until the location is reached and then returns the data. It will throw an error if the location number is greater than the size.

Size:
This will return the size of the LinkedList.

Remove (Type 1):
This remove method accepts no values. The remove method will remove the first node of the LinkedList. This will also return the data held in that node. The first node is set to the second node.

Remove (Type 2):
This remove method accepts a number for the index. It will iterate until it reaches that index and then remove that node. The nodes surrounding it will be joined together. The output is the data value from the removed node.

Remove (Type 3):
This remove method accepts a data value. It will iterate until it finds that data value and then remove that node. The nodes surrounding it will be joined together. The output is a Boolean.

Clear:
This will clear the LinkedList. All the nodes will be removed and the first node of the LinkedList will be set to null.

Nodes:

```
private static class Node {
    public E data;
    public Node next;
    // optional: private Node before;
    public Node (E d, Node n) {
      data = d;
      next = n;
    }
}
```

A node is the building block of a LinkedList. A better LinkedList would have more complex nodes that have private variables and methods to access and alter data values and pointers within it.

While using nodes is a rare occurrence, understanding them is very helpful as similar structures appear quite often in Java and other programming languages.

Example:

There are many common programming problems that are asked in interviews. Knowing these questions as well as possible solutions for them will make you much more prepared for a career in the software industry. One such problem uses LinkedLists as a basis for the problem, and it has a simple solution.

Problem:

Say you have a LinkedList, and you need to check if it is cyclical. How can you do so?

Think of the solution and write it in the blank space below. The answer is on the next page.

Code:

```
Floyd's Cycle Finding Algorithm
public boolean  isLoop() {
        Node slow = head, fast = head;
        while  (slow != null  && fast != null  && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if  (slow == fast) {
                return  true;
            }
        }
        return  false;
}
```

Explanation:

How this solution works is through the use of 2 different nodes. They iterate through the LinkedList at 2 different speeds. If they ever meet up, it means that they will have cycled back to the same node again. If one of them ever becomes null, it reached the end of the LinkedList, thus proving no cycle exists.

This solution works better than most other answers because of two points.

Firstly, the code does not need to create and store a list of variables, it simply has two nodes that iterate at different speeds. An answer that requires hashing every node would require a list to store every hash for comparison.

Second, the code is clear and concise. Any new members to your coding team would be able to briefly glance over it and understand its contents. It does not require much space to write and is a simple solution.

Overall, it is a simple solution that runs rather well, and it should be well known as this problem may arise often.

Questions:

1. Write a complete Node class with accesor methods.

2. How does a LinkedList work, and why is it useful?

3. What type of LinkedList does Java have? What are the pros and cons to this type of LinkedList?

Review:

In this unit, you have learned:

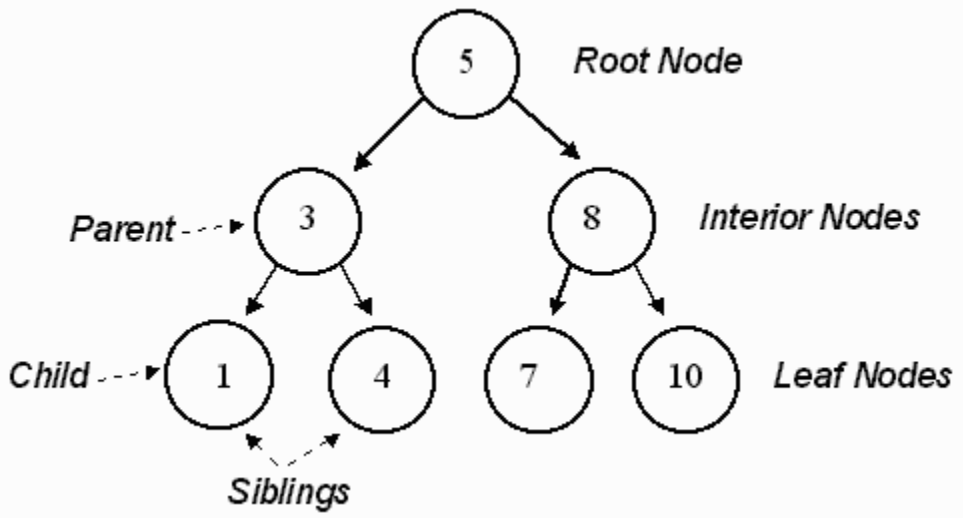
1. What a LinkedList is
2. What a Node is
3. How to instantiate a LinkedList
4. How to instantiate and operate a Node
5. The two types of LinkedList and their uses
6. What methods does a LinkedList have and what they do
7. How to find a cyclical LinkedList

You were later tested over this material and expected to show a strong understanding of the material. If you did not do well on the questions, it is **HIGHLY RECOMMENDED** that you go and review the topics covered again. After you feel that you have properly prepared yourself on the topic feel free to move on to the next topic.

Use the space provided for your own notes

Trees

A tree is a hierarchical tier of nodes. This works with a parent node that has child nodes. The first parent node, which does not have a parent node, is called the root. All other nodes in the tree must have a parent. Every child node has a parent node, and the child nodes without child nodes of their own are called leaves. The most encountered type of tree is a binary tree, where every node can have up to two children, but a scenario may occur where the children nodes have more than two children.



Creating a tree may come up in programming, but the instances are rare as you will more than likely use a TreeSet or a TreeMap in its place. Nevertheless, it is very useful to understand how to create a node for a Tree and how to traverse a Tree to output its code.

Node:

```
public class TreeNode {
    TreeNode left, right;
    Data d;

    public TreeNode(TreeNode l, TreeNode R, Data data){
        left = l;
        right = R;
        d = data;
    }
}
```

The TreeNodes are left and right for the left and right child respectively. The left child should always be smaller than the right child and the parent node. The right

child should always be greater than the left child and the parent node. The Data is the variable that the Tree is storing.

Transversal:

Preorder	print, left, right
Postorder	left, right, print
Inorder	left, print, right
Reverseorder	right, print, left

There are four major transversals. While various other types do exist, as long as you know and understand these four, you can derive the rest of the transversals.

Preorder:

```
public void preorder (TreeNode node) {
	System.out.println(node.data);
	preorder(node.l);
	preorder(node.r);
}
```

Postorder:

```
public void postorder (TreeNode node) {
	postorder(node.l);
	postorder(node.r);
	System.out.println(node.data);
}
```

Inorder:

```
public void inorder (TreeNode node) {
	inorder(node.l);
	System.out.println(node.data);
	inorder(node.r);
}
```

Reverseorder:

```
public void reverseorder (TreeNode node) {
	reverseorder(node.r);
	System.out.println(node.data);
	reverseorder(node.l);
}
```

Questions:

1. Write a complete Node class.

2. Define a complete Tree and a full Tree.

3. Write the code for a method that take the root node of a tree and outputs the preorder transversal of it.

Review:

In this unit, you have learned:

1. What a Tree is
2. What a TreeNode is
3. How to transverse a Tree
4. How to instantiate and operate a TreeNode
5. What a Binary Tree is
6. What transverses exist and how they work

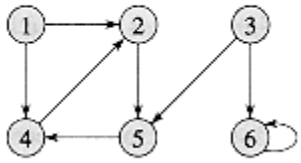
You were later tested over this material and expected to show a strong understanding of the material. If you did not do well on the questions, it is **HIGHLY RECOMMENDED** that you go and review the topics covered again. After you feel that you have properly prepared yourself on the topic feel free to move on to the next topic.

Use the space provided for your own notes

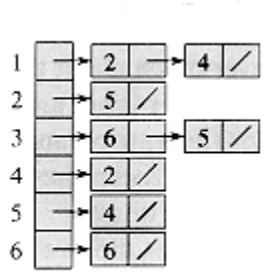
Greedy Algorithms and Graphs

A graph is a group of connected points. Not all the points may be connected directly, but all of them can be. These can be represented as either a matrix or a list or even a drawing. They can also be bi-directional or directional. All of these factors mean that graphs are incredibly versatile, and mastering them will be a crucial skill.

Transversal of a graph means to visit all vertices and edges once in an order without repeats. Sometimes when using graph algorithms, you have to guarantee that the all vertices of the structure are visited only once. The specific order that the vertices are visited is important and will depend upon the algorithm and problem at hand. During the graph traversal process, it is necessary for you to track which vertices have already been traveled through. This can be easily done by marking the vertices after traveling to them in the node or a different data structure all together.



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

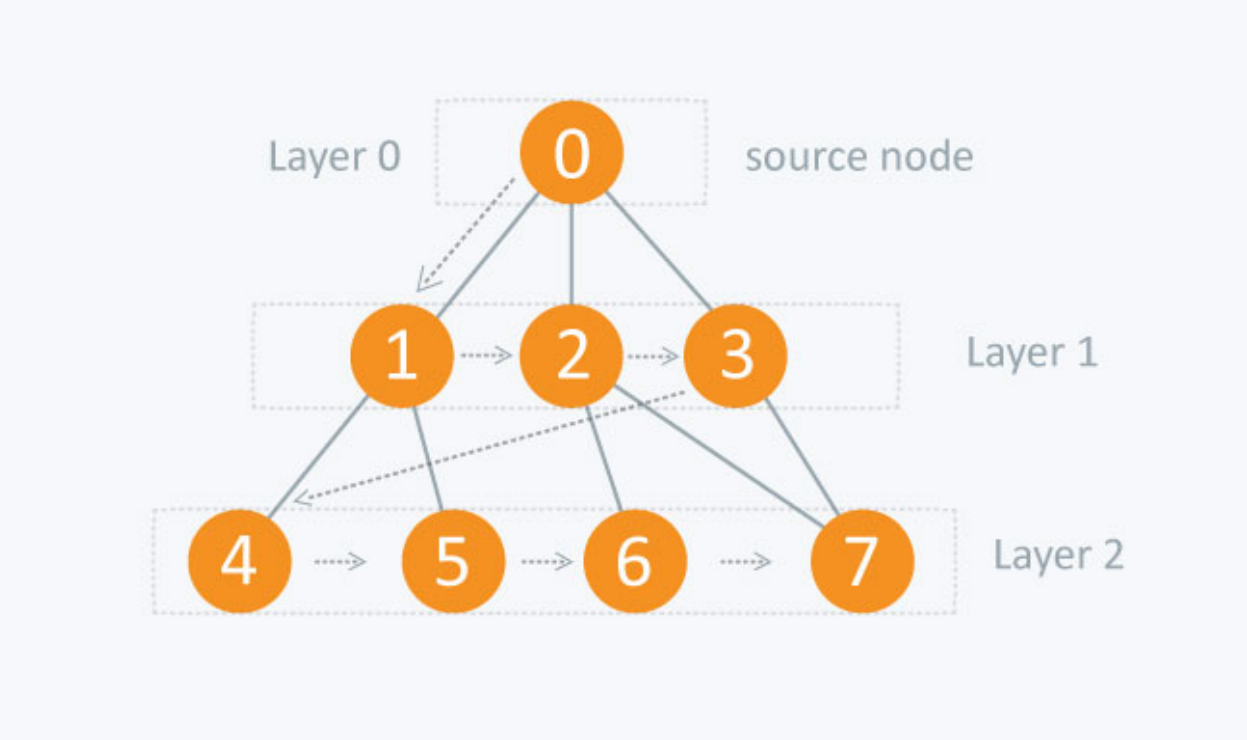
Fractional Knapsack

```
static double getMaxValue(Item[] items, int maxWeight) {
        Arrays.sort(items);
        int currentWeight = 0;
        double currentValue = 0.0;
        for (Item item : items) {
            if (currentWeight + item.weight <= maxWeight) {
                currentWeight += item.weight;
                currentValue += item.value;
            } else {
                int remaining = maxWeight - currentWeight;
                currentValue += remaining * item.getRatio();
                break;
            }
        }
        return currentValue;
    }
```

The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can which will always be the optimal solution to this problem.

BFS

There are a variety of algorithms to traverse graphs. BFS is the most commonly used technique. BFS is a traversing algorithm in which you start traversing from a specific node - usually the starting node - and traverse the graph layer wise to explore the neighbor nodes, nodes which have direct connections. You then continue toward the next level and lower nodes.



```
private static boolean inBounds(char[][] mat, Point p) {
      return Math.min(p.x, p.y) >= 0 && p.x<mat.length && p.y < mat[0].length;
}

public static List<Point> solve(char[][] mat, Point start, Point end) {
      Set<Point> visited = new HashSet<>();
      Map<Point, Point> parents = new HashMap<>();
      Queue<Point> q = new LinkedList<>();
      q.add(start);
      while (!q.isEmpty()) {
         Point cur = q.remove();
         visited.add(cur);
         if (cur.equals(end)) {
            LinkedList<Point> path = new LinkedList<>();
            Point node = cur;
            while (node != null) {
               path.addFirst(node);
               node = parents.get(node);
            }
            return path;
         } else {
            for (int[] dir : dirs) {
               Point n = new Point(cur.x + dir[0], cur.y + dir[1]);
               if (inBounds(mat, n)&&!visited.contains(n)&&mat[n.x][n.y] == '.'){
                  q.add(n);
                  parents.put(n, cur);
               }
            }
         }
      }
      return null;
   }
```

DFS

The DFS algorithm is a recursive algorithm which incorporates backtracking. It uses exhaustive searching of nodes by traveling ahead if able, otherwise using backtracking.

Here, backtrack means that when moving forward, if there are no more nodes available on the current path, you will travel back on the same path to find nodes and traverse them. All of the nodes should be marked and visited on the current path until all the unvisited nodes have been traversed. Afterwards, the next path to go down will be selected.

This recursive nature of DFS is best implemented using stacks. The basic idea is as follows:

1. Choose a start node and push all its adjacent nodes into a stack.
2. Pop a node from the stack to choose the next node to travel to and push all the adjacent nodes into the stack.
3. Repeat this process until the stack is empty.

Guarantee that the nodes which get visited are marked. If you forget to mark the nodes that get visited, you will end up in an infinite loop.

```
private static boolean inBounds(char[][] mat, Point p) {
	return Math.min(p.x, p.y)>= 0&& p.x < mat.length && p.y < mat[0].length;
}
public static List<Point> solve(char[][] mat, Point start, Point end) {
	Set<Point> visited = new HashSet<>();
	Map<Point, Point> parents = new HashMap<>();
	Stack<Point> s = new Stack<>();
	s.add(start);
	while (!s.isEmpty()) {
	   Point cur = s.pop();
	   visited.add(cur);
	   if (cur.equals(end)) {
	      LinkedList<Point> path = new LinkedList<>();
	      Point node = cur;
	      while (node != null) {
	         path.addFirst(node);
	         node = parents.get(node);
	      }
	      return path;
	   } else {
	      for (int[] dir : dirs) {
	         Point n = new Point(cur.x + dir[0], cur.y + dir[1]);
	         if (inBounds(mat, n)&&!visited.contains(n)&&mat[n.x][n.y] == '.'){
	            s.add(n);
	            parents.put(n, cur);
	         }
	      }
	   }
	}
	return null;
}
```

Dijkstra

Use two sets, one set holds vertices counted in the shortest path, the other set holds the vertices not yet counted in the shortest path. In all steps of this algorithm, we find a vertex which is in the set of nodes not yet counted. and has a minimum distance from the start node.

1. Create a set that keeps track of vertices included in the shortest path tree, whose minimum distance from start node has been calculated. To start, this set will be empty.

2. Assign a value of distance to all vertices in the input graph. Set all the distance values as INFINITE to start. Set the distance value as 0 for the start node so that it is chosen first.

3. While all vertices have not been visited:

    A. Pick a vertex U has not been visited and has a minimum distance value.

    B. Visit it and mark it.

    C. Update the distance value of all the adjacent vertices of U. To change all the distance values, iterate across all the adjacent vertices. For the adjacent vertex V: if the sum of distance value of U and weight of edge U-V is less than the distance value of V, then update the distance value of V.

The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated and use it show the shortest path from source to different vertices. The code is for undirected graph, same Dijkstra algorithm can be used for directed graphs also. The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from the source to a single target, we can break the for the loop when the picked minimum distance vertex is equal to target (Step 3.A of the algorithm).

```
public static void printDistances(int[][] graph, int fromVertex) {
        int numVertices = graph.length;
        int[] d = new int[numVertices];
        boolean[] v = new boolean[numVertices];
        for (int I = 0; I < numVertices; I++) d[I] = Integer.MAX_VALUE;
        d[fromVertex] = 0;
        for (int k = 0; k < numVertices - 1; k++) {
           int minDist = Integer.MAX_VALUE, vertex = -1;
           for (int c = 0; c < numVertices; c++)
              if (!v[c] && d[c] <= minDist) {
                 minDist = d[c];
                 vertex = c;
              }
           v[vertex] = true;
           for (int n = 0; n < graph.length; n++)
           if (!v[n]&&graph[vertex][n]>0&&d[vertex]!=Integer.MAX_VALUE){
                if(d[vertex]+graph[vertex][n]<d[n])
                    d[n]=d[vertex]+graph[vertex][n];
           }
        }
}
```

Questions:

1. In what situations is Fraction Knapsack beneficial? Explain in paragraph form.

2. How many vertices and edges are in the matrix below?

1	0	0	1	1
0	0	0	1	1
1	0	0	1	1
1	1	0	1	0

3. What does Dijkstra do?

4. What are two differences and two similarities between BFS and DFS?

5. Which algorithm should be used in a weighted graph?

Review:

In this unit, you have learned:

1. The uses of Greedy Algorithms and Graphs
2. What a graph is and how to identify vertices and edges
3. How to use Fractional Knapsack
4. How to use BFS and DFS
5. How to differentiate BFS and DFS
6. How to use Dijkstra

You were later tested over this material and expected to show a strong understanding of the material. If you did not do well on the questions, it is **HIGHLY RECOMMENDED** that you go and review the topics covered again. After you feel that you have properly prepared yourself on the topic feel free to move on to the next topic.

Use the space provided for your own notes

Test:

This your final review of this book.

In this test, you will prove your understanding of Java in its entirety.

Explanation:

There are two parts to the test:

1. Free Response
2. Multiple Choice

Free Response:

Your Free Response test will be 40 questions of free response questions. You will be expected to do all questions with no resources but a pencil and scratch paper.

Multiple Choice:

Your Multiple Choice test will be 40 questions of multiple choice questions. The questions will be either true or false questions or matching questions. There will be 20 true or false questions and 20 matching questions. You will be expected to do all questions with no resources but a pencil and scratch paper.

Scoring:

For grading, your test scores will be weighted properly so that they are both equal. The Free Response will be added to the Multiple Choice Test. If you get lower than an 60, you have failed.

Free Response Test:

You may write notes here once time has begun:

1. Describe the function of the peek method.

2. What does the keySet method in a Map return? Explain two ways this is useful.

3. Given two integers n and p denoting the number of houses and the number of pipes. The connections of pipe among the houses contain three input values: A, B, D denoting the pipe of diameter D from house A to house B, find out the efficient solution for the network. The output will contain the number of pairs of tanks and taps T installed in first line and the next T lines contain three integers: house number of tank, house number of tap and the minimum diameter of pipe between them.

4. Convert 100343 from base 6 to base 11.

5. What is the worst runtime for quicksort?

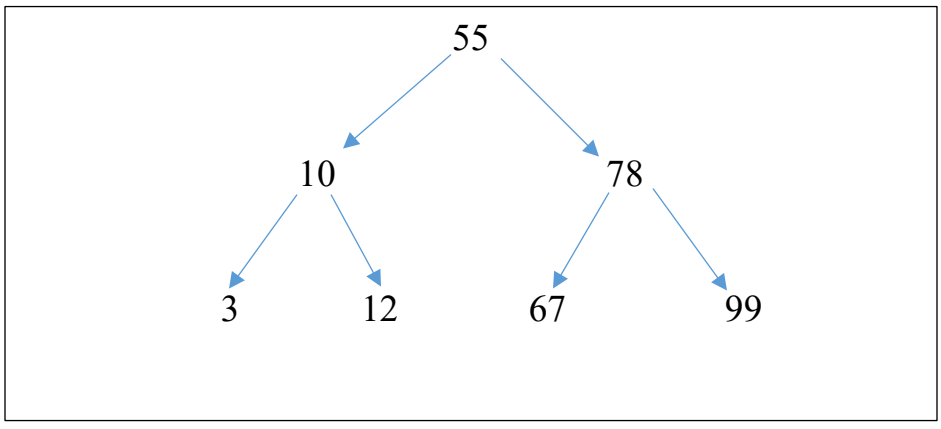
6. What sort is best for sorting LinkedLists and why?

7. Solve the postfix expression: "8 3 9 / - 22 +".

8. What is Quicksort best at sorting and why?

9. Briefly summarize how you would determine if a LinkedList is cyclical.

10. Write your code to determine if a LinkedList is cyclical.



11. Write the postorder transversal of the above tree.

12. Write the inorder transversal of the above tree

13. Write the preorder transversal of the above tree

14. Write the reverse transversal of the above tree

15. What is used in the definition of HashSets and HashMaps?

16. What is used in the definition of TreeSets and TreeMaps?

17. Define the term "Complete Tree".

18. What method is used to convert from any base to base 10?

19. What method is used to convert from base 10 to any base?

20. Write code to solve a postfix expression, if the expression is given in an array.

21. Write code to solve a prefix expression, if the expression is given in an array.

22. Write a method from scratch which will convert a base 10 number, passed as an int, into a base 13 number, returned as a string.

23. What is the best and worst runtime of QuickSort?

24. Why is HashSet more efficient than a TreeSet when adding?

25. Write code to check if a String is a palindrome using a Stack.

26. Write code to check if a String is a palindrome using a Queue.

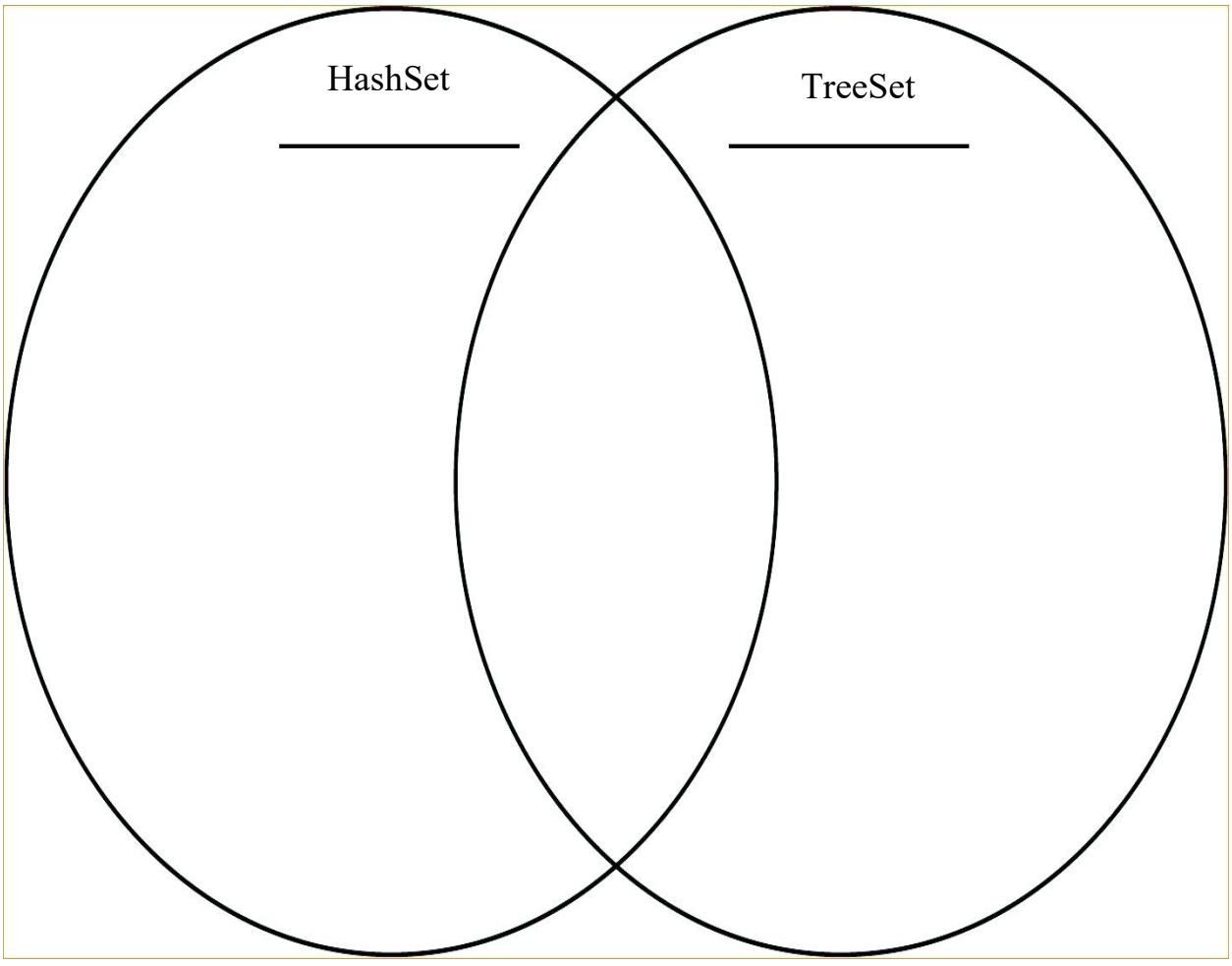
27. What is the runtime for removing from a HashMap?

28. Create a Node class for use in a LinkedList.

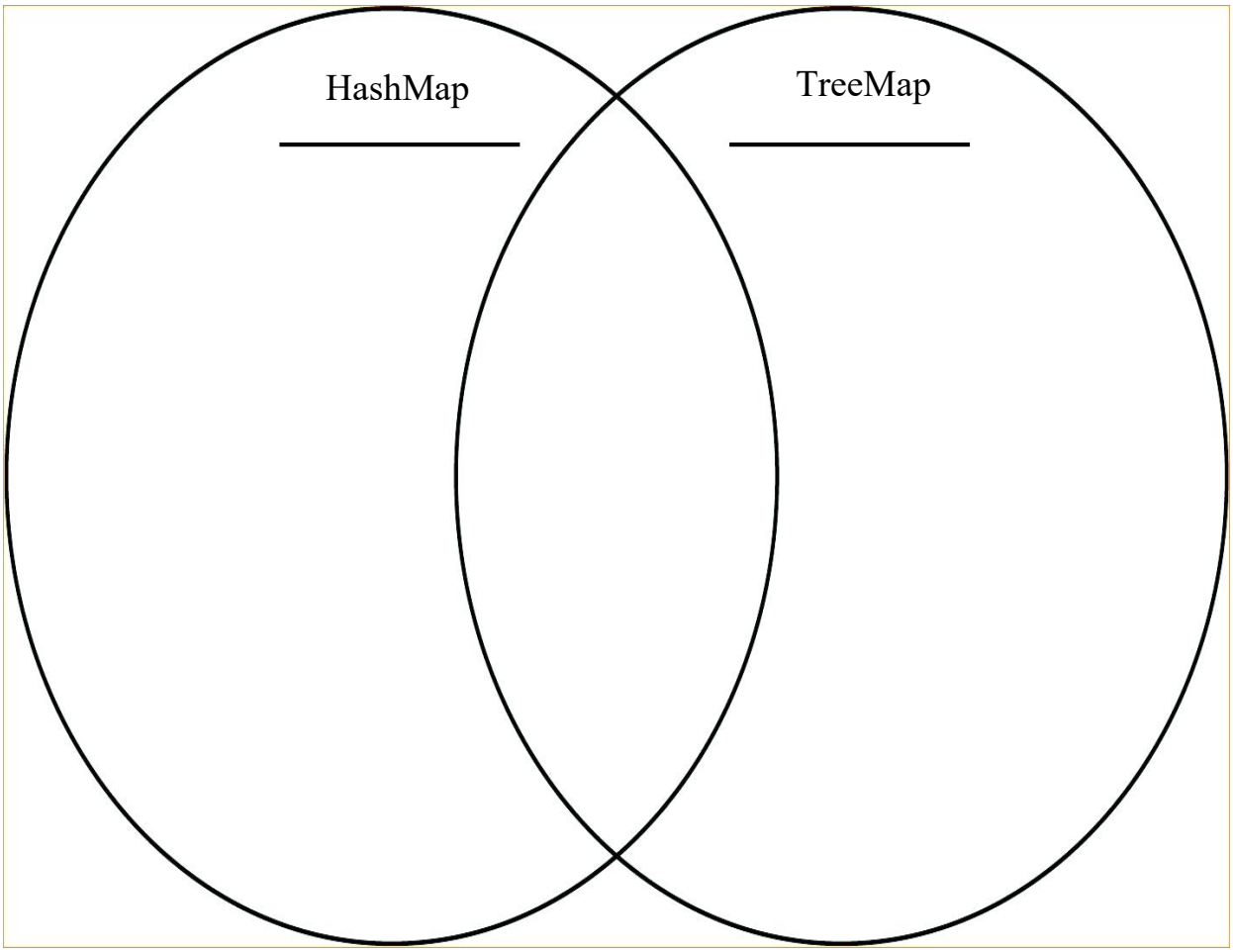
29. Write both Add and all three Remove methods of a LinkedList using the node class above.

30. Instantiate a TreeMap of TreeMaps of Queues of Arraylists of Integers and LinkedLists of Stacks of Bytes and HashMaps of TreeSets of Strings and Priority Queues of Doubles.

31. Complete the Venn Diagram below with two points in every circle.



32. Make a chart showing QuickSort and MergeSorts runtimes for best case, worst case, and average case.
33. Complete the Venn Diagram below with two points in every circle



34. Write a paragraph explaining the process of Fractional Knapsack.

35. Write the code for BFS.

36. Using a LinkedList, write the Queue class's code.

37. Using an ArrayList, write the Stack class's code.

38. Instantiate a TreeMap, and explain why each part is important.

39. Write a method to convert an integer from base 10 to an integer in base 16 and return the converted integer.

40. You are organizing housing for a group of four hundred university students and only one hundred of the students will receive places. The Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. The pairs will be given in a TreeMap, with a student's name, and an ArrayList of all incompatible students. Write code that could provide a list of students or print "No list possible" if there is no possible combination of students.

Multiple Choice Test:

You may write notes here once time has begun.

True or False:
Write true or false for each statement below

1. A LinkedList can be neither singly nor doubly linked.

2. A Priority Queue sorts from highest to lowest.

3. Adding to a TreeMap is an O(n) process.

4. Bitwise Shifts have a higher priority than Binary And.

5. MergeSort is more efficient than QuickSort when sorting LinkedLists.

6. Another name for base 6 is hexadecimal.

7. 15 in base 12 is greater than 15 in base 10.

8. Divide and Conquer formulas rely on splitting large data chunks.

9. Stacks use push and pop instead of add so remove.

10. Queue works on a first-in-first-out system.

11. Stack works on a first-in-last-out system.

12. Linked List can go forwards and backwards when singly linked.

13. Items in a Linked List are added to the end.

14. The best runtime for Quicksort is the same as the best runtime for Merge Sort.

15. The worst runtime for Quicksort is the same as the worst runtime for Merge Sort.

16. The first place in a number system is to the first power.

17. You can instantiate a Map or Set without putting anything in the chevrons.

18. LinkedList<String> q = new Queue<String>(); is a valid instantiation.

19. Fractional Knapsack works through ratio calculations and value analysis.

20. Dijkstra's algorithm works by picking the maximum distance value and not already included.

Matching:
Answers may be used multiple times or not at all

1. ____ Level by level searching
2. ____ Leftmost from root then jump to right
3. ____ Can have two pointers in a node
4. ____ Must have 2 pointers in a node
5. ____ Last in first out
6. ____ First in first out
7. ____ First variable removed, but data structure is sorted
8. ____ Time to add to middle of a doubly Linked List
9. ____ Worst Case sort of Quick Sort
10. ____ Best Case sort of Merge Sort
11. ____ Preorder
12. ____ Postorder
13. ____ Doubly LinkedList
14. ____ Ideal for solving post-fix expressions
15. ____ Ideal for solving pre-fix expressions
16. ____ TreeSet and TreeMap use
17. ____ HashSet and HashMap use
18. ____ Ranges from 0-f
19. ____ 1001 0100 1010 1011
20. ____ Used in the code for a Queue

A. Binary
B. Dijkstra
C. Hashtables
D. Bitwise Shifts
E. Priority Queue
F. O(n^2)
G. O(1)
H. Fractional Knapsack
I. Queue
J. Print, Left, Right
K. O(nlog(n))
L. Binary Trees
M. Base 16
N. DFS
O. Left, Right, Print
P. Stack
Q. Nodes
R. LinkedList
S. BFS
T. O(1) speed adding to middle

Author Info

Anish Goel

Born November 6, 2001

Follow me on Instagram at _anish.goel_

Going to University of Texas at Austin

Follow me on Github at Anish-Goel

My other interests include natural sciences and business

If you have any questions, email me at goelanish12@gmail.com

Support me at: paypal.me/GoelAnish

Qualifications:

4 years of High School Computer Science Education

Over 1583 Practice it Problems done

5 On the Computer Science A AP Test

100% CodingBat problems complete (April 2019)

3 Years of Computer Science Club

Vice President of the Computer Science Club

Author of this book

Personal Statement

Please hire me.

Final Words

““Break out the rubber duck”

– Bryce Hulett”

– Anish Goel

Works Cited

9615-09, cis.temple.edu/~pwang/9615-AA/Lecture/09-Graph-1.htm.

"A Computer Science Portal for Geeks." GeeksforGeeks, www.geeksforgeeks.org/.

"Breadth First Search Tutorials & Notes | Algorithms." HackerEarth, www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/.

"It." Practice, practiceit.cs.washington.edu/.

Shankar, Shrivu. "sshh12 - Overview." GitHub, 25 June 2017, github.com/sshh12?tab=repositories.