Final Review Questions

1: JCF & Big-Oh

1.1 Big-Oh Notation

1. What is O(n)? What kind of object is it?

   It is a set of all functions..

   O(f(n)) denotes the set of functions g(n) where c•f(n) starts to dominate at large enough n.

   It describes the limiting behaviour of a function. In terms of algorithms, it describes the worst case runtime. It is a set of all functions that are larger than the runtime of an algorithm, at sufficiently large n.

   It describes the growth rate of an algorithm's runtime as the number of elements increases.

2. What does the statement 2n + 3 = O(n^2) mean?

   It means that the function 2n+3 is a member of the Set O(n^2)

   It means that for a given value k, 2n + 3 ≤ k•n^2

3. Recall the basic big-O hierarchy: for any constants a,b > 1:

   ```
   O(a) < O(log n) < O(n) < O(n•log n) < O(n^b) < O(c^n)
   ```

   Where do the functions 2n^2 = O(n^b), 100nlogn = O(n•log n), and log(n^3) = O(n•log n) fit into this hierarchy?

4. What is the running time (in big-o notation) for the following function, assuming the method foo() runs in constant time?

```
for(int i=1; i<n; i++)          // n times =>O(n)
    for(int j=n-i; j<n; j++)    // n/2 times =>
O(n)
        foo();                  // O(1)
```

   Thus, this function has a runtime that is O(n^2).

1.2 Java Collections Framework - Interfaces

All of these questions should be considered in the context of the interfaces in the JCF.

1. Explain the differences and similarities between a Set and a List.

   - A Set has no duplicates, whereas a List may have duplicates.
   - A List has an order to its elements, wheras a set is unordered.
     - List elements can be accessed with an index [quick get]
     - Set elements can be added / removed easily
   - Both contain elements.
   - Sorted Sets have successor search

2. Explain the difference between a Collection and a Map. Could it also make sense to have Map be a subclass of Collection?

   A Collection contains values but no key

   - Ordered or Unorded

   A Map organizes values based of a key [key-value pairs]

   - It is possible to find a value knowing only a key
   - No duplicate entries
   - Unordered

3. Which of the JCF interfaces would be the most useful if we want to store a collection of students enrolled in COMP2402 so that we can quickly check if a student is enrolled in COMP2402?

   USet: if we are only storing name or student number [or both together as one value] HashTable: has quick contains() if we are storing name and student number seperately

4. How does your answer to the previous question change if we also want to store some auxiliary information (e.g., a mark) with each student.

   HashTable: unordered Map, quick for contains()

5. A Bag is like a set except that elements can be stored more than once. Explain how you could implement a Bag using a Map.

   You can implement Bag interface by using key as the identifier for the element, and value as a counter for how many instances of this element are in the bag.

1.3 Java Collections Framework - Implementations

1. Explain why it is important that elements that are stored in a Set or Map aren't modified in a way that affects the outcome of the equals() method.

   This is important because, otherwise they cannot be sorted using a comparator, since this uses the equals() method of the object.

2. Describe the running time of the methods get(i) and set(i,x) for an ArrayList versus a LinkedList.

   ArrayList is quicker since no need to traverse.

   - O(1)

   LinkedList traversal is minimized if doubly linked list

- O(min(n-i, i) + 1)

3. Describe the running time of the method add(i,x) for an ArrayList versus a LinkedList.

   ArrayList is good for adding or removing at the front / back because no need to shift elements.

   - ArrayStack: O(n-i+1)
   - ArrayDeque: O(min(n-i,i) + 1) LinkedList is good for adding anywhere, as long as there is a pointer to the location being added to, otherwise traversal through each element is needed.
   - O(1) [given pointer to location]

4. For each of the following methods, decide if it is fast or slow when:

   (a) when l is an ArrayList
   (b) when l is a LinkedList.

```
// Good for ArrayList
// Good for LinkedList
public static void frontGets(List<Integer> l, int
n) {
    for (int i = 0; i < n; i++) {
        l.get(0);
    }
}
// Good for ArrayList
// Good for LinkedList
public static void backGets(List<Integer> l, int
n) {
    for (int i = 0; i < n; i++) {
        l.get(l.size()-1);
    }
}
// Good for ArrayList
// Bad for LinkedList
public static void randomGets(List<Integer> l, int
```

```
n) { Random gen = new Random();
    for (int i = 0; i < n; i++) {
        l.get(gen.nextInt(l.size()));
    }
}
// Good for ArrayList
// Good for LinkedList
public static void insertAtBack(List<Integer> l,
int n) {
    for (int i = 0; i < n; i++) {
        l.add(new Integer(i));
    }
}
// Bad for ArrayList, unless ArrayDeque
// Good for LinkedList
public static void insertAtFront(List<Integer> l,
int n) {
    for (int i = 0; i < n; i++) {
        l.add(0, new Integer(i));
    }
}
// Bad for ArrayList, Unless DualArrayDeque
// Good for LinkedList
public static void insertInMiddle(List<Integer> l,
int n) {
    for (int i = 0; i < n; i++) {
        l.add(new Integer(i));
    }
    for (int i = 0; i < n; i++) {
        l.add(n/2+i, new Integer(i));
    }
}
```

2: Lists as Arrays

These questions are all about implementing the List interface using arrays.

2.1 ArrayStacks

Recall that an ArrayStack stores n elements in a backing array a at locatiosn a[0],...,a[n-1]:

```
public class ArrayStack<T> extends AbstractList<T> {
    T[] a;
    int n;
    ...
}
```

1. Describe the implementation and running times of the operations get(i) and set(i,x) in an ArrayStack.

   An ArrayStack is just an array with access only to back. Superceded by an ArrayDeque.

   - get(), set() in O(1)

2. Recall that the length of the backing array a in an ArrayStack doubles when we try to add an element and n + 1 > a.length. Explain, in general terms why we choose to double rather than just add 1 or a constant.

   Better to double size of the backing array because otherwise we would have to resize after each element that is added. A constant isn't used because if you have a small list you don't want 100 unused spaces, but when working with large sets with highly variable size you don't want to be resizing every 100 if you are adding or removing 1000 each second.

   A dynamic number that is relative to the size of the stack is best.

3. Recall that, immediately after an ArrayStack is resized it has a.length = 2*n.

   a. If we are currently about to grow the backing array a, what can you say about the number of add() and remove() operations since the last

time the ArrayStack was resized?

At least n/2 add or remove operations.

- If we have n items:
  - Growing: last time it was resized it had n/2 items before [n/2 calls]

b. Recall that we shrink the back array a when 3*n < a.length. If we are currently about to shrink the backing array a, what can you say about the number of add() and remove() operations since the last time the ArrayStack was resized?

We had 3n items before

- so at least 2n calls

4. From the previous question, what can you conclude about the total number of elements copied by resize() if we start with an empty ArrayStack and perform m add() and remove() operations?

   At most 2m items will be copied for ArrayStack/ArrayQueue/ArrayDeque after m calls.

5. What is the amortized (or average) running time of the add(i,x) and remove(i) operations, as a function of i and size().

   add / remove: O(n-i+1) + 1

   - Quick to add to back

   get / set: O(1)

6. Why is the name ArrayStack a suitable name for this data structure?

   It is suitable because it implements the Stack Interface, FILO, and is implemented using arrays.

2.2 ArrayDeques

Recall that an ArrayDeque stores n elements at locations a[j], a[(j+1)%a.length], ..., a[(j+n-1)%a.length]:

```
public class ArrayDeque<T> extends AbstractQueue<T> {
    T[] a;
    int j;
    int n;
    ...
}
```

1. Describe, in words, how to perform an add(i,x) operation **(a)** if $i < n/2$ **(b)** if $i \geq n/2$.

   ArrayDeque appears like a circular array, no need to worry about the actual front or back of the array.

   - If the index is less than n/2, then shift values left since fewer values to shift.
   - Else, shift values right
   - There is a private index j which stores where the first value is:
   - $i < n/2$, j--
   - $i \geq n/2$, j = j

2. What is the running time of add(i,x) and remove(i) operations as a function of i and size()?

   ArrayDeque is faster than ArrayStack because it has front access

   - O(min(n-i,i) + 1)

3. Describe, in words, why using System.arraycopy() to perform shifting of elements in the add(i,x) and remove(i) operations is so much more complicated for an ArrayDeque than an ArrayStack.

Because the first index in the backing array does not necessarily correspond to the first index of the ArrayDeque.

4. Explain why, using an example, if a.length is a power of 2, then x % a.length = x & (a.length-1). Why is this relevant when discussing ArrayDeques?

   This is relavant because you can check if the front of the ArrayDeque is pointing to the last index in the backing array, so it can be reset to the 0th index if an element is added.

2.3 DualArrayDeques

Recall that a DualArrayDeque implements the List interface using two ArrayStacks:

```
public class DualArrayDeque<T> extends AbstractList<T>
{
    ArrayStack<T> front;
    ArrayStack<T> back;
    ...
}
```

1. If the elements of the list are x(0, ... x_(n-1), describe how these are distributed among front and back and in what order they appear.

2. Recall that we rebalance the elements among front and back when 3•front.size() < back.size() or vice versa. After we rebalance, we have front.size() == back.size() ± 1. What does this tell us about the number of add() and remove() operations between two consecutive rebalancing operations?

   This means that we had at least n/2 calls to add() / remove().

   - assume initially balanced arrays
   - each array has n/2

- before rebalancing, front array had n/4, and back array has 3n/4
  - difference of n/2
- this means that there was n/2 add() / remove(), (where n is the final size)

2.4 RootishArrayStacks

Recall that a RootishArrayStack stores a list in a sequence of arrays (blocks) of sizes 1, 2, 3, 4, ...

```
public class RootishArrayStack<T> extends
AbstractList<T> {
    List<T[]> blocks;
    int n;
    ...
}
```

1. If a RootishArrayStack has r blocks, then how many elements can it store?

   Each block stores r elements, so r+(r−1)+(r−2)..1 elements

   Geometric Series: r(r+1) / 2

2. Explain how this leads to the equation:

   ```
   b(b+1)/2 < i+1 < (b+1)(b+2)/2
   ```

   - The number of indices less than or equal to i are i+1 [inclusive of element i]
   - The block at which i is contained must be at least b(b+1)/2
   - But cannot be within the next block given by b+1(b+2)/2

3. In a RootishArrayStack that contains n elements, what is the

maximum amount of space used that is not dedicated to storing data?

The amount of wasted space in a RootishArrayStack is sqrt(n)

- Makes it memory-efficient

3: Linked Lists

3.1 Singly-Linked Lists

Recall our implementation of a singly-linked list (SLList):

```
protected class Node{
    T x;
    Node next;
}
public class SLList<T> extends AbstractQueue<T> {
    Node head;
    Node tail;
    int n;
    ...
}
```

1. Draw a picture of an SLList containing the values a, b, c, and d. Be sure to show the head and tail pointers.

   ...in markdown? [nah]

2. Consider how to implement a Queue as an SLList. When we enqueue (add(x)) an element, where does it go? When we dequeue (remove()) an element, where does it come from?

   A queue can be created with two pointers: head and tail

   - The tail should be here new nodes are added to, so they can point to the existing queue

- Removals should be made at the head, so that head.next becomes the new head

3. Consider how to implement a Stack as an SLList. When we push an element where does it go? When we pop an element where does it come from?

   Only one pointer is needed: head

   - Insertion and removals should be done at the head

4. How quickly can we find the ith node in an SLList?

   Quick to find if near the front [head]

   - O(1+i)

5. Explain why we can't have an efficient implementation of a Deque using an SLList.

   You don't have quick access to the predecessor to the tail

   - Tail is only good for insertion
   - Can be fixed with doubly linked list

3.2 Doubly-Linked Lists

Recall our implementation of a doubly-linked list (DLList):

```
protected class Node {
    Node next, prev;
    T x;
}
public class DLList<T> extends
AbstractSequentialList<T> {
    protected Node dummy;
    protected int n;
    ...
}
```

1. Explain the role of the dummy node. In particular, what are dummy.next and dummy.prev?

   The dummy node stores the pointers to the front and back nodes

   - dummy.next is the head
   - dummy.prev is the tail

2. One of the following two functions correctly adds a node u before the node p in the DLList, the other one is incorrect. Which one is correct?

```
// correct implementation
protected Node add(Node u, Node p){
    u.next = p;
    u.prev = p.prev;
    u.next.prev = u;
    u.prev.next = u;
    n++;
    return u;
}
// incorrect implementation
protected Node add(Node u, Node p){
    u.next = p;
    u.next.prev = u;    // this overwrites p.prev
    u.prev = p.prev;
    u.prev.next = u;
    n++;
    return u;
}
```

3. What is the running-time of add(i,x) and remove(i) in a DLList?

   Quick if no traversal is needed O(1) given a pointer.

   - O(min(n-i) + 1) [minimized traversal]

3.3 Space-Efficient Doubly-Linked-Lists

Recall that a space efficient doubly-linked list implements the List interface by storing a sequence of blocks (arrays) each containing b ± 1 elements.

The wasted space is divided by b, but adds b blocks

- n + O(n/b + b)

1. What is the running-time of get(i) and set(i) in a space-efficient doubly-linked list?

   Traversals are reduced by b blocks

   - O(min(n-i,i)/b + 1)

2. What is the amortized running time of the add(i) operation in a space-efficient doubly-linked list?

   Same as regular SLL, but divided by b blocks

   - O(min(n-i)/b + 1)

3. In a space-efficient doubly-linked list containing n elements, what is the maximum amount of space that is not devoted to storing data?

   Wasted space: n pointers + O(n/b + b)

4: SkipLists

Recall that a skiplist stores elements in a sequence of smaller and smaller lists L0, ..., Lk. Li is obtained from Li-1 by tossing a coin for each element in Li-1 and including the element in Li if that coin comes up heads.

1. Draw an example of a skiplist select a few elements and show the search paths for these elements

   Like a regualar SLL with extra randomized nodes for skipping forward

2. Explain how the reverse search path is related to the following experiment: Toss a coin repeatedly until the first time the coin comes up heads.

   The expected search path can be described as the expected coin flip, since each additional height in a node due to a probability.

3. If there are n elements in L0, what is the expected number of elements in L1? What about in Li?

   L1 = L0/2 = n/2 because there is a P(height++) = 0.5

   - L0 contains all node levels with 0 skips to next node [also known as all elements]

   Li = i/2

4. If there are n elements in L0, give an upper bound on the expected length of the search path for any particular element.

   O(log n)

   - specifically 2•log(n) + O(1)

5. Explain, briefly, how a skiplist can be used to implement the SortedSet interface. What are the running times of operations add(x), remove(x), and contains(x)?

   Define the length of a skip [edge].

   Contains:

   - Go right if the next node is less than value
   - Go down if next node is greater than
   - This is a successor search

   add(), remove(), contains() runs in O(log n)

6. Explain, briefly, how a skiplist can be used to implement the List interface. What are the running times of the operations add(i,x),

remove(i), get(i,x), and set(i,x).

add(), remove(), get(), set() runs in O(log n) [expected]

5: Hash tables

1. If we place n distinct elements into a hash table of size m using a good hash function, how many elements do we expected to find in each table position?

2. Recall the multiplicative hash function hash(x) = (x.hashCode() * z) >>> w-d.

   a. In 32-bit Java, what is the value of w?

   - 32 bits...

   b. How large is the table that is used with this hash function? (In other words, what is the range of this hash function?)

   - The range is m - 1, where m = 2^d
     - 0...(2^d -1)

   c. Write this function in more standard mathematical notation using the mod and div (integer division) operators. hash(x) = (z•x) mod 2^w) div 2^(w-d)

3. Explain the relationship between a class's hashCode() method and its equals(o) method.

   Two objects with equal values will return the same hash codes

   - a.equals(b) => a.hashCode() == b.hasCode()

4. Explain, in words, what is wrong with the following hashCode() method:

```
public class Point2D{
```

```
    Double x,y;
    ...
    public int hashCode(){
        return x.hashCode() ^ y.hashCode();
    }
}
```

Give an example of many points that all have the same hashCode().

- If '^' is exponent The point 1,2 will return the same hash code as 2,1, and any two point with the same values will return the same hash codes even though they are not the same points
- If '^' is XOR, then it will return 0 if x == y

5. Explain, in words, what is wrong with the following hashCode() method:

```
public class Point2D{
    Double x,y;
    ...
    public int hashCode(){
        return x.hashCode() + y.hashCode();
    }
}
```

Give an example of 2 different points that have the same hashCode().

- This will give an issue with points of swapped values, e.g. 1,2 and 2,1

6: Binary Trees

6.1 Binary Trees

1. Draw a good sized binary tree.

...ha

2. Label the nodes of your drawing with their depth.

   ...

3. Label the nodes by the order they are processed in a preorder traversal.

   Preorder: Root, then left always, and then right **Inorder:** Left to right **Postorder:** Bottom to top, with left priority

4. Label the nodes by the order they are processed in an inorder traversal.

   ...

5. Label the nodes by the order they are processed in a postorder traversal.

   ...

6. Label the nodes of your tree with the sizes of their subtrees

   ...

7. What kind of traversal would you do to compute the sizes of all subtrees in a tree?

   You would use inorder traversal

   - Preorder is good for copying the tree [root first]
   - Postorder is good for deleting the tree

6.2 Binary Search Trees

1. Consider an array containing the integers 0...15 in sorted order. Illustrate the operation of binary search on a few (a) integer values and a few (b) non-integer values.

- Check if current value is equal to expected value
- if current value is smaller: go to left child, else right child

2. Draw a binary search tree containing (at least) 0...15

   ...

3. Show the search path for a value x in the tree and a value x' not in the tree

   ...

4. Insert some value x' into the tree

   Like finding but simply add at point search would fail.

5. Delete some value x from your tree (try this for external nodes, internal nodes with 1 child, and internal nodes with two children).

   External node:

   Easy, delete node...

   Internal node:

   One child: move child up one, done...

   Two children: Find closest value greater than deleted node value. Replace deleted node with this one.

7: Treaps

1. Choose a permutation of the values 0...15. Draw the binary search tree that results from inserting the elements of your permutation in order. Are there other permutations that could have given the same search tree?

   ...

   And yes, there are many [probablistically]

2. Explain the relationship between quicksort and random binary search trees.

   By building a RBST with the pivot value at the node, quick sort is implemented, since every value is compared to this pivot.

   Given quicksort input of size n, the recursion tree is a RBST.

3. Define the heap property for how priorities are used to make a binary tree into a Treap.

   The heap property states that a child's key will be more extreme than it's parent's key. This is used to organize a binary tree into a heap by adding a 'priority' key to each node. This is done to bound the height of the tree, as the keys are randomly generated.

4. Pick some random numbers p_0...p_15, and draw the treap that contains 0...15 where p_i gives the priority for number i.

   ...

5. Explain the relationship between random binary search trees and Treaps.

   Since the keys are unique and randomly generated, a Treap can be thought of as following both the heap property and binary search tree properties.

   Or, a Treap can be thought of as a RBST with nodes insterted by increasing priority.

6. Explain the relationship between insertion and deletion in a Treap.

   When values are added or removed, the tree may have to move nodes around to maintain heap property [heapify].

8: Scapegoat Trees

Recall that a scapegoat node is where 3•size(u) > 2•size(u.parent).

```
They have a deterministic height [this is what triggers
rebuild, when height is exceeded]
```

1. In a scapegoat tree, is size(u) < size(u.parent) for every non-root node u?

   Yes, since any subtree of u, must be a subtree of the parent node.

2. Draw an example of a tree that looks surprisingly unbalanced but is still a valid scapegoat tree.

   ...

   - Assume a value for limiting parameter q (q = n = 10)
   - Then the height can be up to log_{3/2}(q)
     - log_{3/2}(10) is about 5.6, so can draw tree with up to 5 height

3. Explain why, in a scapegoat tree, we keep two separate counters n and q that both -sort of- keep track of the number of nodes?

   n is the number of elements in the list

   q is a counter, used to maintain the upper-bound on the number of nodes

   They are used to bound the height, by checking if by adding / removing a node does not maintain a height ≤ log_{3/2}(q).

   q is initially the value of n, and incremented with each added node, and decremented with each removed node.

4. If v is a scapegoat node (for example, because 3•size(v.left) > 2•size(v)) then explain how many operations (insertion/deletions) have affected v's subtree since the last time the subtree containing v was rebuilt.

at least 1/3•size(v) calls to add() / remove().

9: 2-4 and Red-Black Trees

Advantages of 2-4: log n height **always**; deterministic. Treap have randomness. Scapegoat has amortized cost.

1. Draw an interesting example of a 2-4 tree.
    - Tree with all every leaf at equal depth, all internal nodes have 2-4 children
2. Explain what happens in a 2-4 tree when an insertion causes a node to have more than 4 children.

    If this causes an internal node to have more than 4 children, then we split the node into two with 2 and 3 children. If the parent node now has too many children, then it too will be split.
3. Explain what happens in a 2-4 tree when a deletion causes a node to have only one child.

    The node is added to the sibling of the internal node, since it is guaranteed to have at least one sibling. If this causes the parent of the internal node to have less than 2 children, then it's merged with its sibling; if the parent is the root, then the sole child of the root becomes the new root.
4. Draw a red-black tree that corresponds to your 2-4 tree.

    Same thing, except all paths to external nodes have same number of black nodes [analagous to same depth of external nodes in 2-4 tree].
    - Anywhere there are more than 2 children to a node, split into groups of one or two with red node parents.
5. Explain the relationship between the red-black tree properties and the 2-4 tree properties.

A RedBlack tree simulates a 2-4 tree with a binary tree implementation.

10: Heaps

10.1 Binary Heaps

These questions are about complete binary heaps represented using the Eytzinger Method.

1. Draw an example of a complete binary heap with key values.

   ...

   Binary tree that maintains the heap property, like a Treap without the priority key, implemented with an array.

   e.g. root is always minimum value

2. Illustrate how your example's values map into an array using the Eytzinger Method.

   ...

   Go through tree with breadth-first traversal, with left child priority.

   - Parent of node at index i are found at (i-1)/2
   - Left child of node at index i are found at 2i + 1
   - Right child of node at index i are found at 2i + 2

3. Consider a binary heap represented using the Eytzinger Method. Give the formulas for the parent, left child, and right child of the value stored at position i.

   - Parent of node at index i are found at (i-1)/2
   - Left child of node at index i are found at 2i + 1
   - Right child of node at index i are found at 2i + 2

4. When we perform a DeleteMin (remove()) operation on a heap, where

do we get the value to replace the root, and what do we do with it?

The easiest way is to replace the root with the last value in the array a[n-1].

- This is probably not the minimum value, thus it must be moved down [heapify]
  - compare value to its children
  - if it is larger than either one, it is swapped with the smallest child
  - repeat

5. Explain or illustrate the insertion algorithm for a heap

   - First check the size of the array, if it is full resize()
   - Next place new value in a[n] [first empty space in array]
   - Heapify
     - swap new value with its parent if it is smaller than the parent

6. Explain the operation of HeapSort

   Heap Sort: Turn everything into Binary Heap, and remove root value everytime.

   Remove root node [this will store it on array at a[n] after deletion]

   - Afterwards array will have sorted list

10.2 Randomized Meldable Heaps

```
Binary Heaps with no limit to size and shape.
```

1. Draw two examples of heaps (not necessarily complete) and show how to meld (merge) them.

   ...

Given two binary trees x, y:

- Random coin flip if y is merged with x.left or x.right
- This is done recursively

2. Explain how the add(x), and remove() operations can be done using the meld() operation.

   Merging only one node is the same as adding one node.

   To remove a node, merge its children

11: Sorting Algorithms

1. Explain the relative advantages of MergeSort, HeapSort, and Quicksort. Things to keep in mind are (a) the number of comparisons performed, (b) the amount of extra memory required by the algorithms, and (c) whether their running time is guaranteed or only probabilistic.

   Stable: Means order of equal values is preserved In-place: Means that original list is modified by sort

   Merge Sort:

   - Only one that is stable
   - Only one that isn't in-place
     - thus extra memory is needed
   - The most comparisons
   - No randomness, this runtime is guaranteed

   Heap Sort:

   - Not stable
   - In-place
   - Quickest in terms of least comparisons
   - No randomness, thus runtime is guaranteed

   Quick Sort:

- Not stable
- In-place
- The second best in terms of comparisons
- Randomness, thus runtime is probabilistic

2. Create a random permutation of numbers 0...15. Walkthrough the sorting of this permutation using the HeapSort, MergeSort, QuickSort, and CountingSort algorithms.

   ...

3. Explain how the Radix sort algorithm works. What key feature of the counting sort algorithm makes it work correctly?

   Radix sort: Best when you have a sparse array, or large range with many zeros.

   It uses counting sorts in sections of size *d*; Sort least significant bits first using Counting Sort, so that next significant bit duplicates are already sorted...

12: Graphs

1. Draw an interesting directed graph (with at least 6 nodes).

   ...

2. How many entries are in the Adjacency List representation of this graph? (Can you know without listing them?)

   ...

3. How many zeroes are in the Adjacency Matrix representation of this graph?

   ...

   Adjacency list has all outbound edges from a node at it's index

4. Walkthrough the Breadth-first and Depth-first traversals of this graph. Of the two trees that these traversals create, which has a greater height?

   Breadth-first search probably has the greatest height since it doesn't really have dead ends easily.