2017 Block 4 – Data Structures – Practice Final

The actual final should represent your individual understanding of the course material. As such, the final should be done independently and will be closed book, closed notes, closed Internet, closed phone, closed friends, etc.

The practice final is shorter than the actual final is likely to be (instructor was pressed for time), but demonstrates the basic shape and structure questions on the final are likely to take.

**Question 1:**

8 pts - Look at the following code and consider how the memory changes as the main method executes. Draw the memory diagram for the state of the memory just before the main method
finishes execution.

```
public class Point {
   int x;
   int y;

   public Point(int a, int b) { x=a; y=b; }
}

public class LineSegment {
   Point a;
   Point b;

   public LineSegment(Point x, Point y) { a=x; b=y; }
}

public class Test {
   public static void main(String[] args) {
      Point s = new Point(0,0);
      Point e = new Point(5,10);
      LineSegment l = new LineSegment(s,e);
      // What does the memory look like here, just before
      // main returns?
   }
}
```

| Address | Type | Name | Value | |
|---|---|---|---|---|
| 0x001 | Point | s | 0x00A | |
| 0x002 | Point | e | 0x00C | |
| 0x003 | LineSegment | l | 0x00F | |
| | | | | |
| 0x00A | int | x | | 0 |

| | | | |
|---|---|---|---|
| 0x00B | int | y | 0 |
| 0x00C | int | x | 5 |
| 0x00D | int | y | 10 |
| 0x00E | Point | a | 0x00A |
| 0x00F | Point | b | 0x00C |

**Question 2:**
3 pts - As computer scientists, we frequently work with problems, algorithms, and programs. How are these ideas related to one another?

A problem is a task to be performed, and an algorithm is the method and or process that allows a person to complete the task designated in the problem. Algorithms can be instantiated as programs in computer programs.

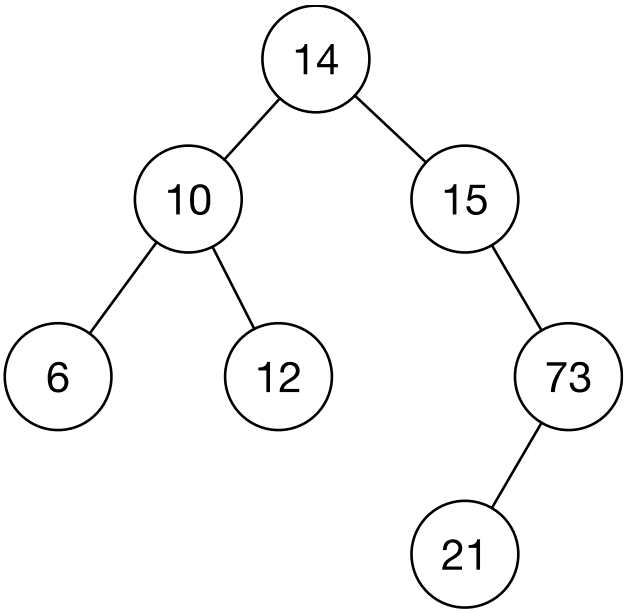
**Question 3:**
2 pts - If two different sorting algorithms have O(n^2) time complexity does that imply that implementations of these algorithms will take the same amount of time to sort lists of the same size? Explain your answer.

Since n is the size of the list, if both implementations of these algorithms have the same size list, then the time complexity will be the same. However, for these two implementations, there is the potential that the time complexities may have different equations, that when represented as time complexity, are changed. For example, the time complexity for one implementation could have been n^2 multiplied by a constant, which would still be represented as O(n^2) for the time complexity. The variations possible in the time complexity equations may lead to a potential difference in running time. Even though two algorithms may have the same time complexity, it is likely the run times are not identical.

**Question 4:**
2 pts - JAVA supports two ways to check for equality between objects, '==' and 'equals()'. Do both of these ways to check for equality produce the same result? Please explain.

In JAVA, .equals() is a method, and == is an operator. When checking for equality, == references the address in memory and checks to see if the two addresses are the same, where as .equals() compares the content or value of two objects. They are both Boolean type statements, however they will not always produce the same results because == can be used for primitive types and object comparison, however, .equals() is used to compare objects. With ==, if the two object being compared are not referenced to the same object, the result will be false, even if the values are the same, but .equals() in the same scenario would be true.

**Question 5:**
Use the following tree for the sub-questions below.




1 pts - What would the output be for a pre-order traversal?
14, 10, 6, 12, 15, 73, 21

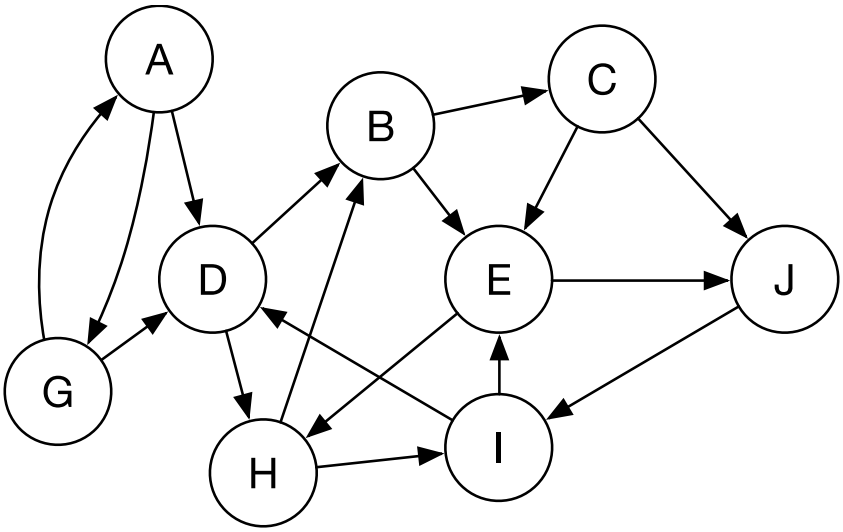
1 pts - What would the output be for an in-order traversal?
6, 10, 12, 14, 15, 21, 73

1 pts - What would the output be for a post-order traversal?
6, 12, 10, 21, 73, 15, 14

1 pts - Is the tree full or complete? Explain your answer.
The tree is not complete because every level must be full, except the last level, which can be partially full, with the nodes being filled from left to right. The tree is not full because every node except the leaves does not have two children.

1 pts - Does this tree look like it represents a binary search tree? Explain your answer.
This tree appears to represent a binary search tree because every left child is of lesser value than the parent and every right child is of greater value than the parent. Every parent also has no more than two children.

**Question 6:**
Use the following graph for the sub-questions below.




1 pts - Perform a depth first search from node A to E, write down the path you found.

Go to right most node first
A → D
D → B
B → C
C → J
J → I
I → E

1 pts - Perform a breadth first search from node G to E, write down the path you found.

G → D
G → A
D → B
D → H
A → D X already visited
A → G X already visited
B → C
B → E Found
Highlighted in green is the path that is used to go from G to E.

1 pts – Does this graph more than one cycle? If so, write out paths for 2 cycles.

A → G → A
B → C → J → I → E → H → B

**Question 7:**

Alex, the bear, had a really hard time with lists but feels good about the queue implementation. On a new project, Alex needs list operations that work by indexes and proposes the following list implementation.

```
public class QueueList<T> {
   Queue<T> q; // a queue to hold the list
   int size;   // number of elements in the list

   public QueueList() {
      q = new Queue<T>();
      size = 0;
   }

   public T fetch(int idx) {
      T r = null;
      Queue<T> tmp = new Queue<T>();
      int i=0;
      while(i<idx) { tmp.enqueue(q.dequeue()); i++; }
      r = q.dequeue();
      tmp.enqueue(r);
      i++;
      while(i<size) { tmp.enqueue(q.dequeue()); i++; }
      q = tmp;
      return r;
   }

   public void remove(int idx) {
      Queue<T> tmp = new Queue<T>();
      int i=0;
      while(i<idx) { tmp.enqueue(q.dequeue()); i++; }
      q.dequeue();
      i++;
      while(i<size) { tmp.enqueue(q.dequeue()); i++; }
      q = tmp;
      size--;
   }

   public void append(T v) {
      q.enqueue(v);
   }

   public void insert(int idx, T v) {
      Queue<T> tmp = new Queue<T>();
      int i=0;
      while(i<idx) { tmp.enqueue(q.dequeue()); i++; }
      q.enqueue(v);
      while(i<size) { tmp.enqueue(q.dequeue()); i++; }
      q = tmp;
      size++;
   }
}
```

2 pts - Do you think Alex's list implementation looks like it will work? Argue for or against Alex's general solution.

I think that Alex's implementation will work as a general solution. Both the remove and insert at a specific index will correctly grab the value at the index by removing all the value before the index and then grabbing the index value. The append will work if the enqueue for the queue implementation works. This way of creating a list resembles a list backed by an array rather than a linked or double linked list.

4 pts - Are there defects in Alex's code? Edge cases that need to be handled or potential off by one problems? If so, high light where the problem might be and explain what might go wrong.

In the append method, the size is not updated. This will cause the size of the queue list to be incorrect and affect the methods that require the size to fetch values for the updated queues. This will mean the remove, insert, and fetch methods will have the incorrect list size and will be unable to perform the function on the correct value. Also, because the code resembles the algorithms used for an array list, in the constructor the queue should be instantiated with a size. If the queue is not given an initial size that can be updated and changed throughout the code when elements or added or removed, there will be an error in the array index. Alex's code does not account for if the queue is empty, this would lead to null pointers and errors in the code. However, Alex does recreate the array by setting the new one at the ends of methods that change the size equal to the original array. So if Alex accounts for the size of the list being zero, initializes the queue with a size, and updates the size, these defects in the code should be resolved.

1 pts - What is the time complexity for fetch? Is it a tight bound? Justify your answer.

Lower: $\Omega(n)$ Upper: $O(n)$ finds value and then remakes the list again into a new list by en-queueing and de-queueing values. This means that the time it takes to execute this method depends on the number of elements in the list. Because the lower and the upper are the same, the tight bound is $O(n)$.

1 pts - What is the time complexity for remove? Is it a tight bound? Justify your answer.

Lower: $\Omega(n)$ Upper: $O(n)$ finds value and then remakes the list again into a new list by en-queueing and de-queueing values. The remove method is imilar to the fetch method, as it has to remake the list each time, so the time complexity depends on the number of elements. Because the lower and the upper are the same, the tight bound is $O(n)$.

1 pts - What is the time complexity for insert? Is it a tight bound? Justify your answer.

Lower: $\Omega(n)$ Upper: $O(n)$ finds value and then remakes the list again into a new list by en-queueing and de-queueing values. For the insert method, the time complexity if the same as the remove and fetch methods due to the remaking of the list each time. Because the lower and the upper are the same, the tight bound is $O(n)$. It would be $n + 1$, but for time complexity, that translates to n.

1 pts – Is there any setting in which Alex's list implementation would be preferable to a linked list or array list? Please explain.

Alex's list implementation would not be preferable to a linked list or an array list in many cases because the methods in Alex's implementation require linear time, but for a linked list and an array list there are situations where constant time is possible. This is because Alex's implementation required recreating the queue list for the remove, insert, and fetch methods. However, a queue would be beneficial if you are adding to the end of the queue and removing from the end due to the first in last out nature of queue. The enqueuer and dequeuer methods in a queue implementation add or remove at the last index in the array, which allows for faster retrieval since the entire list does not need to be searched. In these cases, it is easier and faster to access these elements of the queue than a linked list.