Problem Set 7, Part I

Problem 1: Choosing an appropriate representation
1-1) *ArrayList or LLList?* **ArrayList**
explanation: You would need to use an ArrayList because using an ArrayList would be better for time efficiency when retrieving the last order placed due to random access. For space efficiency, the ArrayList resizes itself, so whenever there is an order that surpasses the amount of space in the ArrayList, it will automatically resize. The ArrayListIterator should start at the end of the list to display the most recent order to the first order.

1-2) *ArrayList or LLList?* **LLList**
explanation: You would need to use an LLList because it is more space-efficient than an ArrayList and in this case, the workshops would be displayed in chronological order so it would be better to use an LLListIterator. Time efficiency matters less since the number of workshops is consistent.

1-3) *ArrayList or LLList?* **ArrayList**
explanation: You would need to use an ArrayList because the course ID that is numbered would need to be accessed often which would benefit since ArrayList has random access. Since there are very few additions and removals, the space efficiency matters less. The ArrayListIterator would be able to efficiently iterate through the list to see if it is full or if there are still spots.

Problem 2: Switching from one list implementation to another
2-1) The overall time complexity for this method would be O(n^2) because the loop runs at n times, or the length of the list, in backward order and the addItem() operation would also run at a time complexity of O(n) because it would need to traverse the list to add it to the front of the linked list every time. The getItem() operation would run at a time complexity of O(1) since looking through an ArrayList has random access, so it would not affect the time complexity as a whole.

2-2) This method would also have a time complexity of O(n^2). The loop would run n times, except this time it would run through in the order of the ArrayList instead of backward. Like the first one, getItem() would run in constant time O(1), and lastly, the addItem() operation would run at O(n) time complexity because it would need to traverse through the linked list until it reaches "i."

2-3)

```
public static ArrayList convert_LtoA(LLList vals) {
        ArrayList converted = new ArrayList(vals.length());

        Node trav = vals.head.next;
        int counter = 0;
        while (trav != null) {
           Object item = vals.getItem(trav);
           converted.addItem(item, counter);
           trav = trav.next;
           counter += 1;
        }
        return converted;
   }
```

2-4) The overall time complexity for this method would be O(n) because this method uses a traversal through a linked list for every node. The addItem() operation would run at a constant time because of random access for an ArrayList.

Problem 3: Working with stacks and queues

3-1)

```
public static void doubleAllStack(Stack<Object> stack, Object item) {
      Stack<Object> newStack = new Stack<Object>();

      while (!stack.isEmpty()) {
            Object current = stack.pop();

            if (current.equals(item)) {
                 newStack.push(item);
                 newStack.push(item);
            } else {
                 newStack.push(current);
            }
      }

      while (!newStack.isEmpty()) {
             stack.push(newStack.pop());
      }

}
```

3-2)

```
public static void doubleAllQueue(Queue<Object> queue, Object item) {
      Stack<Object> newQueue = new Stack<Object>();

      while (!queue.isEmpty()) {
             newQueue.push(queue.remove());
         }


      while (!newQueue.isEmpty()) {
            Object current = newQueue.pop();

            if (current == item) {
                  queue.insert(current);
                  queue.insert(current);
            } else {
                  queue.insert(current);
            }
      }

}
```

Problem 4: Binary tree basics

4-1) The height is 3.

4-2) 4 leaves and 5 interior nodes.

4-3) 21 18 7 25 19 27 30 26 35

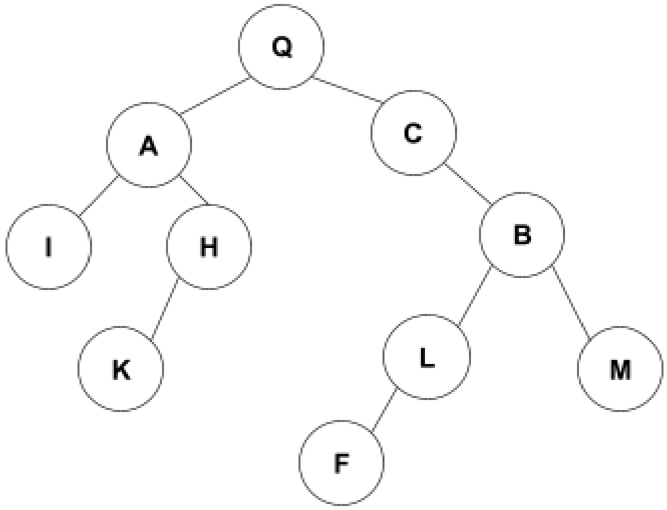
4-4) 7 19 25 18 26 35 30 27 21

4-5) 21 18 27 7 25 30 19 26 35

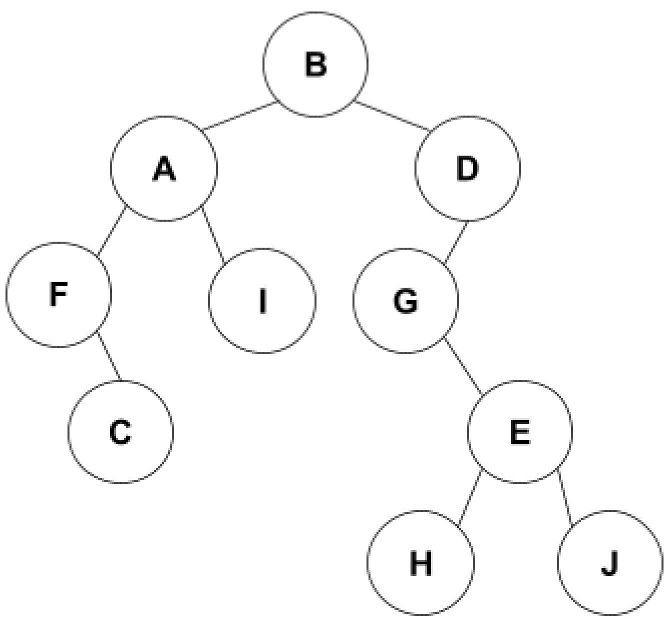
4-6) This is not a search tree because there is a value on the left side of the tree that is greater than the root.

4-7) It is balanced because the height difference is 0 and if the height difference is 1 or less, then the tree is balanced

Problem 5: Tree traversal puzzles
5-1)

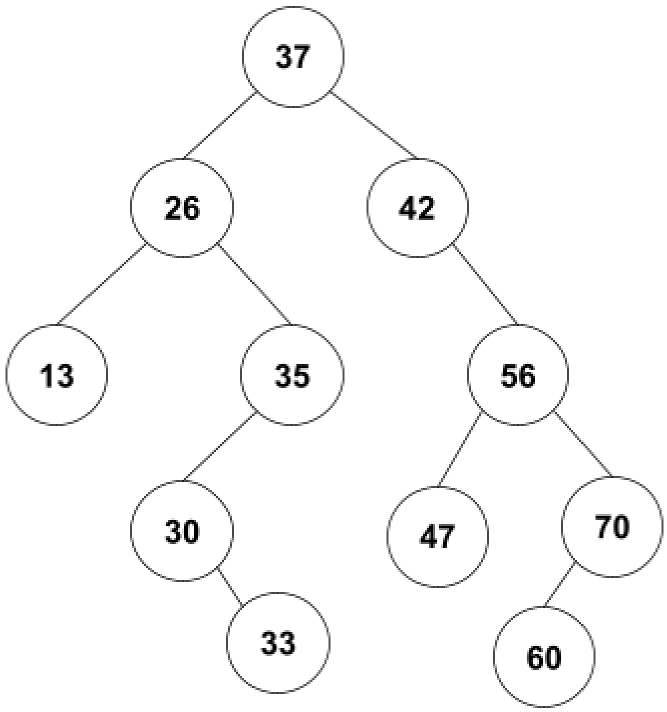


5-2)



Problem 6: Binary search trees

6-1)



6-2)