Session 1: You have 40 Minutes

1. [20 pts.] Graphs, Trees and Queue/Stack. Remember that each tree is a graph as well. Write a method that takes a binary tree and traverses the tree in breadth-first-search order. During the traversal, the method should print out the values in tree with their levels. Analyze the running time of your method.

```
public void breadthFirstSearch(BinaryTree<T> tree) {
    Queue<Pair<BinaryTree<T>,int>> theQueue =
          new ArrayDeque< Pair<BinaryTree<T>,int>>();
    theQueue.offer(new Pair<>(tree, 1));
    StringBuilder sb = new StringBuilder();
    while (!theQueue.isEmpty()) {
          Pair<BinaryTree<T>,int>  currPair = theQueue.remove();
          BinaryTree<T> currTree = currPair.getKey();
          int currLevel = currPair.getValue();
          if (!currTree.isEmpty()){
                sb.append(currTree.getData());
                sb.append(currLevel+"\n");
                theQueue.offer(new Pair<>(current.getLeftSubtree(),
                                          currLevel+1);
                theQueue.offer(new Pair<>(current.getRightSubtree(),
                                          currLevel+1);
          }
    }
    System.out.print(sb.toString());
}
```

A pair is pushed into the queue once for each element in the tree. At each iteration of the while loop one element in the queue is popped out and processed. So, the while loop iterates N times for N element tree. Each loop of the while loop takes constant time since all the operations in the loop is constant time operations. As a result the running time is θ(N).

2. [15 pts.] Run-time analysis and Sorting. Answer each of the following questions briefly. You do not need to give explanation.
   a. What are the worst-case, best-case and average-case running times of quick sort?
      $\theta(n^2)$, θ(n log n), θ(n log n)
   b. What is the running time of heap sort if all elements to be sorted are same?
      θ(n)
   c. Specify which of the following sorting algorithms
      - insertion sort
      - merge sort
      - quick sort
      - Shell sort

      i. gives the best running time if the array is almost sorted.
         Insertion sort
      ii. gives the best worst-case running time.
         Merge sort
      iii. uses more than constant size extra memory space.
         Merge sort

Session 2: You have 40 Minutes

3. [20 pts.] Hashing and List. Consider **`Class HashTableChain`** with data fields given below. Write a private method **`find`** that takes an element **`key`** as parameter and searches the **`key`** value in the Map. The method should return a **`ListIterator`** such that when the **`next()`** method of the **`ListIterator`** is called, it should return the **`Entry`** that has the **`key`** value. It should not have a next node if the **`key`** is not in the corresponding list. Remember **`LinkedList`** has method **`listIterator()`** that returns a **`ListIterator`** and **`ListIterator`** traverses the list in both directions (forward and backward).

Data Field	Attribute
`private LinkedList<Entry<K, V>>[] table`	A table of references to linked lists of `Entry<K, V>` objects.
`private int numKeys`	The number of keys (entries) in the table.
`private static final int CAPACITY`	The size of the table.
`private static final int LOAD_THRESHOLD`	The maximum load factor.

```
private ListIterator<Entry<K, V>> find(Object key) {
    int index = key.hashCode() % table.length;
    if (index < 0) {
        index += table.length;
    }
    if (table[index] == null) {
        return null;
    }
    ListIterator<Entry<K, V>> iter = table[index].listIterator();
    while (iter.hasNext()) {
        Entry<K, V> nextItem = iter.next();
        if (nextItem.key.equals(key)) {
            iter.prev();
            return iter;
        }
    }
    return iter;
}
```

4. [15 pts.] You have learnt several data structures in this course. Considering the following data structures, explain **briefly** when do prefer each data structure over the others and when we should not use each data structure.
   a. Linked List

      Prefer: Elements are accessed sequentially. Elements have predecessor and successor. Elements are inserted or deleted from both ends.

      Don't use: Elements are accessed in sorted order or using indices. Fast search is required. Fast access of minimum is required.

   b. Red Black Tree

      Prefer: Elements are accessed in sorted order. Insertion, deletion and search performance is critical, linear worst-case is not acceptable.

      Don't use: Fast access of minimum is required. Elements have predecessor and successor. Fast search is required. Elements are not ordered.

c. Hashing

    Prefer: Fast insertion, deletion and search is required. Elements are not ordered.
    Don’t use: Elements are accessed in sorted order. Fast access of minimum is required. Elements have predecessor and successor.

d. Binary Heap

    Prefer: Fast access of minimum is required.
    Don’t use: Elements are accessed in sorted order. Elements have predecessor and successor. Elements are not ordered. Fast search is required.

Session 3: You have 40 Minutes

5. [20 pts.] BBST. Write a constructor in **`Class AVLTree`** that takes a sorted **`ArrayList`** of N elements and builds an AVL tree for these N elements. Your method should run in O(N) time. Note that inserting each element one by one into an empty AVL tree would take O(N log N) time. Analyze the running time of your method and show that it is really O(N) time. You may use helper methods.

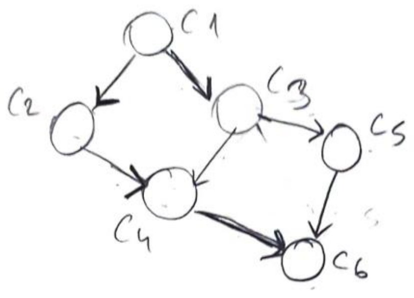
```
public AVLTree(ArrayList<E> items) {
      super();
      root = buildtree (items, 0, items.size());
      setBalance(root);
}

private AVLNode<E> buildTree (ArrayList<E> items, int s, int e) {
      if (e<s) return null;
      if (e==s) return AVLNode(items.get(e));
      int mid = (e+s)/2;
      AVLNode<E> curr = AVLNode(items.get(mid));
      curr.left = buildTree (items, s, mid-1);
      curr.right = buildTree (items, mid+1, e);
      return curr;
}

private int setBalance(AVLNode<E> curr){
      // sets the balance of node curr
      // returns height of the tree rooted at curr
      if (curr==null) return 0;
      int left = setBalance(curr.left);
      int right = setBalance(curr.right);
      curr.balance = right - left;
      return (left<right) ? right+1 : left+1;
}
```

Both helper methods, buildTree and setBalance take linear time since a recursive method call is executed for each element and each call takes constant time. So, overall running time is Theta(N).

6. [15 pts.] Graphs and List. There is no programming in the question. Consider the following list of course prerequisites where C1 -> C2 denotes that the course C1 is a prerequisite for the course C2.

C1 -> C2	C3 -> C4	C2 -> C4	C4 -> C6
C1 -> C3	C3 -> C5	C5 -> C6	

   a. Draw the graph representing this prerequisite list, neatly.



b. Write down the adjacency list representation of the graph.

$$\begin{tikzcd}
\boxed{C_1} \arrow[r] & \boxed{C_2} \arrow[r] & \boxed{C_3} \arrow[r] & \emptyset \\
\boxed{C_2} \arrow[r] & \boxed{C_4} \arrow[r] & \emptyset & \\
\boxed{C_3} \arrow[r] & \boxed{C_4} \arrow[r] & \boxed{C_5} \arrow[r] & \emptyset \\
\boxed{C_4} \arrow[r] & \boxed{C_6} \arrow[r] & \emptyset & \\
\boxed{C_5} \arrow[r] & \boxed{C_6} \arrow[r] & \emptyset & \\
\boxed{C_6} \arrow[r] & \emptyset & &
\end{tikzcd}$$

c. Keeping the adjacency lists in alphabetical order, perform a depth-first-search beginning at vertex C1. Write down the discovery order and finish order or the nodes.

)	Disc Order	Finish Order
Disc C1	C1	
Disc C2	C1, C2	
Disc C4	C1, C2, C4	
Disc C6	C1, C2, C4, C6	
Finish C6		C6
Finish C4		C6, C4
Finish C2		C6, C4, C2
Disc C3	C1, C2, C4, C6, C3	
Disc C5	C1, C2, C4, C6, C3, C5	
Finish C5		C6, C4, C2, C5
Finish C3		C6, C4, C2, C5, C3
Finish C1	C1, C2, C4, C6, C3, C5	C6, C4, C2, C5, C3, C1

d. Find the topological ordering for the graph using the algorithm specified in our textbook.

When we reverse the dfs finish order of a graph we get the topological ordering of that graph. So:

T.O = $C_1, C_3, C_5, C_2, C_4, C_6$