Assignment 12 - Search Trees and Hashing

Q1.

```
static class Node {
       private int data;
       private Node left;
       private Node right;

       Node(int data) {
           this.data = data;
           this.left = this.right = null;
       }
   }

   static Node findLeftMost(Node root){
       // helper function to traverse

       if (root == null){
           return null;
       }

       while (root.left!= null){
           root = root.left;
       }
       return root;
   }

   static Node deleteNode(Node root){
       // node with only one child or none
       if (root.left == null){
           Node child = root.right;
           root = null;
           return child;
       } else if (root.right == null){
           Node child = root.left;
           root = null;
           return child;
       }
```

```java
        // Node with two children, get inorder successor in the right sub-tree
        Node next = findLeftMost(root.right);
        root.data = next.data;
        root.right = deleteNode(root.right);

        // return root pointer
        return root;
    }

    static Node delete(Node root, int key1, int key2){
        // base
        if (root == null){
            return null;
        }

        // traverse BST
        root.left = delete(root.left, key1, key2);
        root.right = delete(root.right, key1,key2);

        // once root's data is in the range, call deletenode
        if (root.data >= key1 && root.data<= key2){
            return deleteNode(root);
        }

        return root;
    }
```

Q2:

```
public class Q2 {
    static class BNode{
        private int cnt;
        private int[] keys;
        private BNode[] children;
        private boolean leaf;

        BNode(int n, BNode parent){
            this.cnt = 0; // count of keys
            this.keys = new int[n-1]; // array list of keys
            this.children = new BNode[n]; // array list of children (BNode)
            this.leaf = true; // boolean for leaf node or not
            this.parent = parent; // a pointer to parent node
        }

    }

    static class BTree{

        private int order; // order of BTree
        private BNode head; // a pointer for BNode head

        BTree(int order){
            this.order = order;
            this.head = new BNode(order, null);
        }
    }

    public BNode search(BNode root, int key){
        int i = 0;
        while (i < root.cnt && root.keys[i] < key ){
            i++;
        }
        if (i<= root.cnt && root.keys[i] == key){
            return root;
        }

        if (root.leaf){
```

```
            return null;
        } else {
            return search(root.children[i], key);
        }
    }

    public void deleteKey(BTree root, int key){
        BNode tmp = search(root.head, key);

        // deletion occcurs at leaf node
        if (tmp.leaf) {
            int i = 0;

            // searching through keys
            while (tmp.keys[i] < key){
                i++;
            }

            while (i < tmp.cnt){
                tmp.keys[i] = tmp.keys[i+1];
                i++;
            }
            tmp.cnt--;
        } else {
            // deletion at non-leaf BNode
        }
    }
}
```

Q3:
For a hash table having tablesize positions and n records already occupied, the lf = n/tablesize. Since new keys are uniformly distributed to the hash table, the possibility that n+1th element's insertion causes collision would be n/table size = lf, like a table below:

Key ID	P(Collision)
1	0/tablesize
2	1/tablesize
3	2/tablesize
...	
n	(n-1)/tablesize

Therefore, an expected number of collisions would be the sum of each independent possibility of collision for n keys:

$$E(\#\ of\ \ collisions)\ =\sum_{1}^{n-1}\frac{n}{tablesize}=\frac{1}{tablesize}+\frac{2}{tablesize}+\ \dots\ +\frac{n-1}{tablesize}$$

$$=\frac{1}{tablesize}\cdot\frac{(n-1+1)\,(n-1)}{2}=\frac{n}{tablesize}\cdot\frac{n-1}{2}=\frac{lf\cdot(n-1)}{2}$$

Q4:
The number of occupied slots in the tablesize-sized hash table is n.
If start by counting the number of insertions for each item, the numbers of comparisons would be:

For the first element, n = 0 is occupied, # of comparison is exactly one = (tablesize+1)/(tablesize-0+1).
For the second element, n = 1, so # of comparisons is: (tablesize+1)/tablesize = 1 + 1/tablesize = 1 + 1/ P(insertion collides for n =1)

If the hash table is full,so n = tablesize, # of comparisons = (tablesize+1)/1 = tablesize+1.

So that the average number of comparisons needed to insert a new element = (tablesize+1)/(tablesize-n+1).

On the other hand, linear proving will not satisfy this condition since it always requires more sequential comparisons, and the number will be over the average number.