# Problem Set 6 - Solution

## Binary Search Tree (BST)

1. Given the following sequence of integers:

   ```
   5, 15, 35, 10, 70, 25, 30, 75
   ```

   1. Starting with an empty binary search tree, insert this sequence of integers one at a time into this tree. Show the tree after every insertion. Assume that the tree will not keep any duplicates. This means when a new item is attempted to be inserted, it will not be inserted if it already exists in the tree.
   2. How many item-to-item comparisons in all did it take to build this tree? (Assume one comparison for equality check, and another to branch left or right.)
   3. What is the worst case number of comparisons for a successful search in this tree? For an unsuccessful (failed) search? (Assume one comparison for equality check, and another to branch left or right.)
   4. What is the average case number of comparisons for a successful search in this tree?
   5. From this tree, delete 35: find the node (y) the largest value in its left subtree, write y's value over 35, and delete y. How much work in all (locating 35, then locating y , then deleting y) did it take to complete the deletion? Assume 2 units of work to perform a target-to-item comparison, and 1 unit of work for a pointer assignment.

   **SOLUTION**

   1. Following is the final tree.

      ```
           5 (0)
            \
            15 (2)
           /   \
      (4) 10   35 (4)
              /   \
         (6) 25   70 (6)
              \    \
           (8) 30   75 (8)
      ```

      Numbers in parentheses next to an item node indicate the number of comparisons required to insert that item.

   2. Total number of comparisons = 38.

   3. Worst case number of comparisons:

      ```
           5
          / \
         F  15
           /   \
          10   35
         / \  /   \
        F  F 25   70
            /  \  / \
            F  30 F 75
              /  \  / \
              F  F  F F
      ```

      Note: The 'F' nodes are not actual tree nodes - they are failure positions.
      - **Successful search:** 9 comparisons. (search for 30 or 75)
      - **Failed search**: 10 comparisons (search for 26 thru 29, or 31 thru 34, or 70 thru 75, or values > 75 - these will end up in one of the lowest level leaf nodes marked 'F' in the tree above.

   4. Average case number of comparisons for successful search:

      ```
           5 (1)
            \
            15 (3)
           /   \
      (5) 10   35 (5)
              /   \
         (7) 25   70 (7)
              \    \
           (9) 30   75 (9)
      ```

      Numbers in parentheses in the tree above indicate the number of comparisons to successfully search for the associated value in the tree. Total number of comparisons = 46. Total number of successful search positions = 8. Assuming equal probabilities of search for all successful search positions, average number of comparisons = 46/8.

   5. To delete 35, here is the work done:
      - **Locating 35**: Number of comparisons is 5. Number of pointer assignments is 6: to move two pointers (prev and ptr) down the tree, with 2 initial assignments, then 2 assignments to move to 15, then 2 assignments to move to 35. And one assignment to hold on to 35.
      - **Locating y**: The largest value in the left subtree of 35 is 30. Locating this requires 4 more pointer assignments. (Move prev and ptr from 35 to 25, then to 30.)
      - **Overwriting 35 with 30**: Not counted since there is no comparison or pointer assignment.
      - **Deleting y (30)**: Since this is a leaf node, one pointer assignment to set 25's right child to null.

      So in all 5 comparisons and 6+1+4+1 = 5*2 + 12*1 = 22 units of work total.

2. Given the following BST node class:

   ```
   public class BSTNode<T extends Comparable<T>> {
       T data;
       BSTNode<T> left, right;
       public BSTNode(T data) {
           this.data = data;
           this.left = null;
           this.right = null;
       }
        public String toString() {
           return data.toString();
   ```

```
                              -
      }
   }
```

Consider the following method to insert an item into a BST that does not allow duplicate keys:

```
   public class BSTN<T extends Comparable<T>> {
        BSTNode<T> root;
        int size;
        ...
        public void insert(T target)
        throws IllegalArgumentException {

                BSTNode ptr=root, prev=null;
                int c=0;
                while (ptr != null) {
                        c = target.compareTo(ptr.data);
                        if (c == 0) {
                                throw new IllegalArgumentException("Duplicate key");
                        }
                        prev = ptr;
                        ptr = c < 0 ? ptr.left : ptr.right;
                }
                BSTNode tmp = new BSTNode(target);
                size++;
                if (root == null) {
                        root = tmp;
                        return;
                }
                if (c < 0) {
                        prev.left = tmp;
                } else {
                        prev.right = tmp;
                }
        }
```

Write a recursive version of this method, using a helper method if necessary.

**SOLUTION**

```
   public class BSTN<T extends Comparable<T>> {
      ...
        public void insert(T target)
        throws IllegalArgumentException {
                root = insert(target, root);
                size++;
        }

        private BSTNode<T> insert(T target, BST<T> root)
        throws IllegalArgumentException {

                if (root == null) {
                        return new BSTNode(target);
                }

                int c = target.compareTo(root.data);
                if (c == 0) {
                        throw new IllegalArgumentException("Duplicate key");
                }
                if (c < 0) {
                        root.left = insert(target, root.left);
                } else {
                        root.right = insert(target, root.right);
                }
                return root;
        }
```

3. * With the same **BSTNode** class as in the previous problem, write a method to count all entries in the tree whose keys are in a given range of values. Your implementation should make as few data comparisons as possible.

```
    // Accumulates, in a given array list, all entries in a BST whose keys are in a given range,
    // including both ends of the range - i.e. all entries x such that min <= x <= max.
    // The accumulation array list, result, will be filled with node data entries that make the cut.
    // The array list is already created (initially empty) when this method is first called.
    public static <T extends Comparable<T>>
    void keysInRange(BSTNode<T> root, T min, T max, ArrayList<T> result) {
        /* COMPLETE THIS METHOD */

     }
```

**SOLUTION**

```
    public static <T extends Comparable<T>>
    void keysInRange(BSTNode<T> root, T min, T max, ArrayList<T> result) {
        if (root == null) {
           return;
        }
        int c1 = min.compareTo(root.data);
        int c2 = root.data.compareTo(max);
        if (c1 <= 0 && c2 <= 0) {  // min <= root <= max)
           result.add(root.data);
        }
        if (c1 < 0) {
           keysInRange(root.left, min, max, result);
        }
        if (c2 < 0) {
           keysInRange(root.right, min, max, result);
        }
    }
```

4. With the same **BSTNode** class as in the previous problem, write a method that would take a BST with keys arranged in ascending order, and "reverse" it so all the keys are in descending order. For example:

```
            25                         25
          /    \                     /    \
        10     40      --->        40      10
       /  \   /  \                /  \    /  \
      2   20 30   45             45  30  20   2
         /    \                     /     \
        15    35                   35     15
```

The modification is done in the input tree itself, NO new tree is created.

```
public static <T extends Comparable<T>>
void reverseKeys(BSTNode<T> root) {
    /* COMPLETE THIS METHOD */
```

**SOLUTION**

```
public static <T extends Comparable<T>>
void reverseKeys(BSTNode<T> root) {
   if (root == null) {
      return;
   }
   reverseKeys(root.left);
   reverseKeys(root.right);
   BSTNode<T> ptr = root.left;
   root.left = root.right;
   root.right = ptr;
}
```

5. * A binary search tree may be modified as follows: in every node, store the number of nodes in its *right subtree*. This modification is useful to answer the question: what is the **k-th largest element** in the binary search tree? (k=1 refers to the largest element, k=2 refers to the second largest element, etc.)

You are given the following enhanced binary search tree node implementation:

```
public class BSTNode<T extends Comparable<T>> {
   T data;
   BSTNode<T> left, right;
   int rightSize;  // number of entries in right subtree
   ...
}
```

Implement the following *recursive* method to find the **k-th largest** entry in a BST:

```
public static <T extends Comparable<T>> T kthLargest(BSTNode<T> root, int k) {
   /* COMPLETE THIS METHOD */
}
```

**SOLUTION** Assume root is not null, and *1* <= *k* < *n*

```
public static <T extends Comparable<T>>
T kthLargest(BSTNode<T> root, int k) {
   if (root.rightSize == (k-1)) {
       return root.data;
   }
   if (root.rightSize >= k) {
       return kthLargest(root.right, k);
   }
   return kthLargest(root.left, k-root.rightSize-1);
}
```