Problem Set 6 - Solution

Binary Search Tree (BST)

---

1. Given the following sequence of integers:

   ```
   10, 17, 3, 90, 22, 7, 40, 15
   ```

   1. Starting with an empty binary search tree, insert this sequence of integers one at a time into this tree. Show the final tree. 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.)

   SOLUTION

   Following is the final tree.

   ```
       10            // 0 comparisons for 1st item
    /       \
   3         17      // 2 comparisons each for 3 and 17
    \      /   \
     7    15    90   // 4 comparisons each for 7, 15, and 90
                /
              22     // 6 comparisons
                \
                 40  // 8 comparisons
   ```

   Total number of comparisons = 30

---

2. For the tree built in the above problem:
   1. 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.)

      ANSWER

      ```
            10
         /       \
        3         17
       / \      /   \
      F    7    15   90
          / \  / \   / \
         F  F F  F 22   F
                  /  \
                 F    40
                     /  \
                    F    F
      ```

      Note: The 'F' nodes are not actual tree nodes - they are failure positions.
      - **Successful search**: 9 comparisons. (search for 40)
      - **Failed search**: 10 comparisons (search for 23 thru 39, or 41 thru 89 - these will end up in one of the lowest level leaf nodes marked 'F' in the tree above.

   2. What is the average case number of comparisons for a successful search in this tree?

      ANSWER

      Average case number of comparisons for successful search:

      ```
          10            // 1 comparison for match
       /       \
      3         17      // 3 comparisons each for 3 and 17
       \      /   \
        7    15    90   // 5 comparisons each for 7, 15, and 90
                   /
                 22     // 7 comparisons
                   \
                    40  // 9 comparisons
      ```

Total number of comparisons = 1+2*3+3*5+7+9 = 38. Total number of successful search positions = 8. Assuming equal probabilities of search for all successful search positions, average number of comparisons = 38/8.

3. From this tree, delete 17: find the node (y) that has the smallest value in its right subtree, write y's value over 17, and delete y. How much work in all (locating 17, then locating y , then deleting y) did it take to complete the deletion? Assume the following (a) you are using two pointers to navigate down the tree, a tracking pointer, and a lagging pointer, (b) 1 unit of work for an equality comparison between target and tree item, one for an inequality check to branch left or right, and 1 unit of work for a pointer assignment.

ANSWER

To delete 17, here is the work done:

- **Locating 17**: Number of comparisons is 3. Number of pointer assignments is 4: to move two pointers (prev and ptr) down the tree, with 2 initial assignments (prev=null, ptr=@10), then 2 assignments to move to 17 (prev=@10, ptr=@17)
- **Locating y**: The smallest value in the right subtree of 17 is 22. Locating this requires 4 more pointer assignments. (Move prev and ptr from 17 to 90, then 90 and 22.)
- **Overwriting 17 with 22**: Not counted since there is no comparison or pointer assignment.
- **Deleting y (22)**: One pointer assignment to set 90's left child to @40.

So in all, comparisons=3, pointer assignments=4+4+1=9, for a total of 3+9=12 units of work.

---

3. 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 BST<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;
     }
```

---

4. * 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);
    }
}
```

---

5. 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;
}
```

---

6. * 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);
}
```