RUTGERS

CS112 Data Structures

Recitation 06

Yu Yang
yy388@cs.rutgers.edu
Office: CoRE 331
Office Hour: 3-5pm, Wed.

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

RUTGERS

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 */

 }
```

RUTGERS

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 */
```

RUTGERS

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);
}
```