CS218 Data Structures

Assignment #03 (CS-4E)

Instructor	**Dr. Shahnawaz Qureshi**
Session	**Spring 2021**

Instructions:

1. You are required to submit a **Microsoft Word** file, containing all your codes along with screenshots (complete console) of every question next to it. Any question without the screenshot will not be accepted. PDF or other format files will not be accepted.

2. Late submissions will **not be accepted.**

3. Word file format should be "**Roll-Number_Section_AsNo**", for example *19F-0101_B_As #01*. Marks will be deducted for not following the correct format.

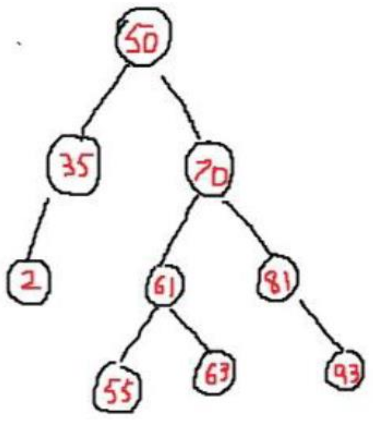
4. Plagiarism will not be tolerated, either done from internet or from some fellow classmate of same/other section. Plagiarized questions will result in **straight zero** or **negative marking.**

Using Binary Search tree, your task is to implement the following functionalities to your class. Please note that you should add necessary utility functions (private methods in the class) that you need for the working of the following methods written in the public part of the class. In the following, n means the number of nodes in the tree and h means the height of the tree.

1. The **height** method. Add a recursive method called height that computes the height of the bst. Note that the height of a node n is simply one more than the height of its taller child. This should take O(n).

2. The **depth** method. This method should return the depth of the key passed as parameter. Recall that the depth of a node (or key) is the number of edges on the path from the root to that node. This should take O(h).

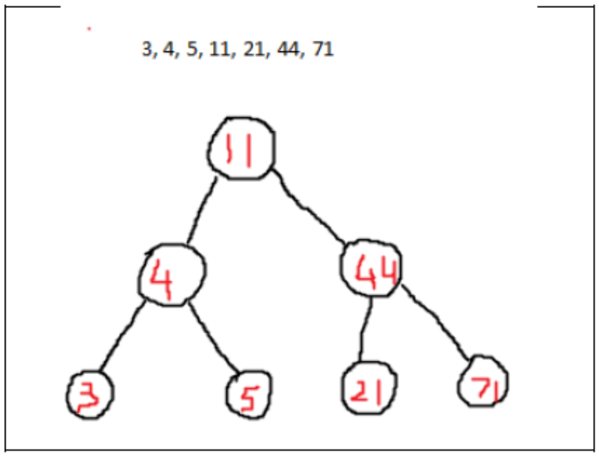
3. Add a method called **isBalanced**, which returns true if the tree is balanced and false otherwise. We define a balanced tree as: a tree in which the difference in the heights of the two sub-trees of every node is no more than 1. For example, the tree below is balanced. The difference in the heights of the sub-trees at any node is called the 'balance factor' (or bf) of that node. Precisely, bf(x) = height(x.right) – height(x.left). Note that we define the height of a nullptr to be -1. So the balance factor of a leaf is always -1 – (-1) = 0. In the following tree, bf(2)=0, bf(55) =0, bf(63) = 0, bf(93) = 0, etc. Furthermore, bf(35) = -1, bf(81) = 1, bf(70) = 0 and bf(50) = 1, etc. In other words, a tree is balanced if the balance factor of every node in it is either 0, 1 or -1. **Your function should work in O(n).**



4. The **destructor ~bst().** The destructor should de-allocate every node in the bst while making sure that there are no memory leaks. You should do this with a post-order traversal.

5. The **copy constructor** and **= operator**. These methods should create a deep copy of the tree, i.e. another tree with its own nodes allocated with the same data as the original tree. In the copy constructor, use the level order traversal. In the assignment operator, do the same thing using the pre-order traversal.

6. Write a parameterized constructor: **bst(T sortedData[], int n)** (*where bst is the name of your class of Binary Search Tree*). This constructor should convert the sorted data given in the array sortedData intro a perfect (or complete) bst. The running time of the function should be no more than O(n). Note that repeatedly calling the insert method will not do the job. Following is an example of a sorted array and the corresponding tree.
   Hint: use recursion.



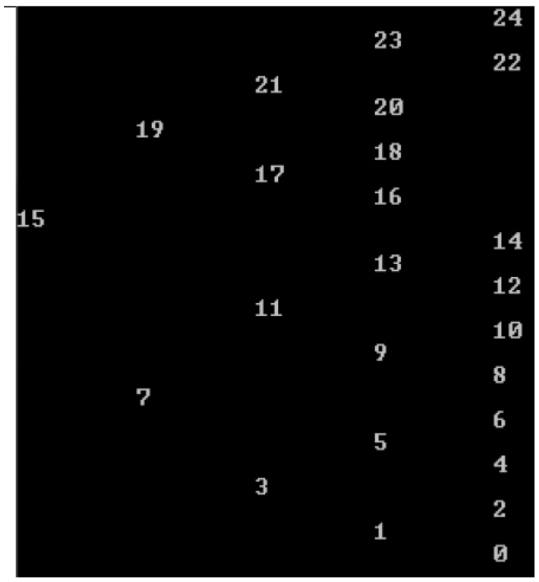
7. Add a method called **isSubtree**. When used like bst1.isSubtree(bst2), it should return true if bst2 is a sub-tree of bst1, i.e. bst2 occurs exactly (in terms of both its data and structure) inside bst1. This should take O(n).

8. Add a method called **breadth** to the class bst. It should return the breadth of the tree, where breadth is defined as the number of nodes in the 'widest level' of the tree, i.e. the level containing most nodes in it.

9. Write two methods called **successor** and **predecessor** to the class BST. The successor of a key in a bst is simply the next key in order of values whereas predecessor is the previous key in order of values. For example, if BST has 1, 2, 3, 6, 10, 13 values in it and you are required to find successor and predecessor of 6 then its successor will be 10, and predecessor will be 3. Also note that 1 has no predecessor and 13 has no successor (your function must handle this exception).

10. Add a method called **trimBelowK** to class bst. It should accept a number k>=0. The method should delete all nodes in the levels below level k, i.e. all nodes in levels k+1, k+2, and so on. Make sure that the new leaf nodes now have NULL children. No new nodes may be allocated at any stage.

11. By using the pre-order, inorder, post-order traversals or even the level traversal, we can only print the data in a straight flat line. Type a method by name of **printTree** which prints your BST in a Tree like shape, so that your BST is more visual and clearer to see. For example, following output shows a BST of 25 nodes.



```
                                24
                        23
                                22
                21
                        20
        19
                        18
                17
                        16
15
                                14
                        13
                                12
                11
                                10
                        9
                                8
        7
                                6
                        5
                                4
                3
                                2
                        1
                                0
```

Now we can clearly see the root 15, and every node and its children; such was for example, 9 and its children 8 and 10, etc.

Your printing method should work in O(n).
Hint: a recursive method with a few lines of code will do the job.