Data Structures BCS-4F
FAST-NU, Lahore, Spring 2021

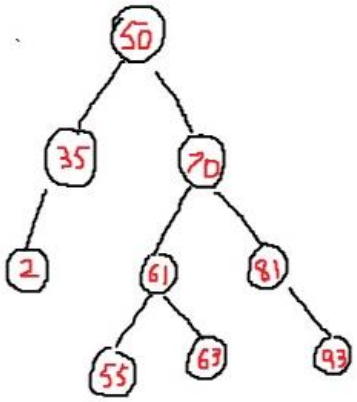
Homework 4
Augmenting the class bst

Due date: Friday May 28, 1159 PM **160 pts**

In this homework, you will add a number of additional methods/functionalities to the class bst that was developed during the lectures. Your code must be added to the code in bst.cpp that's been provided with this statement.

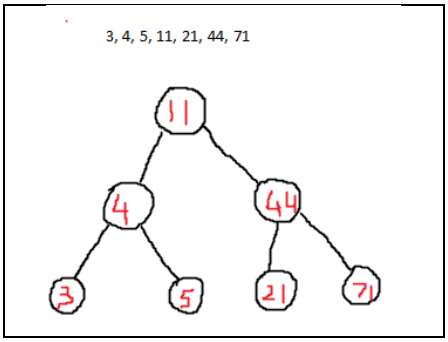
Following is a list of the functionalities you need to add to the class bst. Please note that you should add necessary utility functions (private methods in the class) that you need for the working of the following methods written in the public part of the class. In the following, n means the number of nodes in the tree and h means the height of the tree.

1. Complete the code of the function **erase**, as described in **lecture 16**.
2. Complete the codes of the functions **successor** and **predecessor** as described in **lecture 17**.
3. 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).
4. 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).
5. 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).

6. Write a parameterized constructor: **bst(T sortedData[], int n)**. 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.

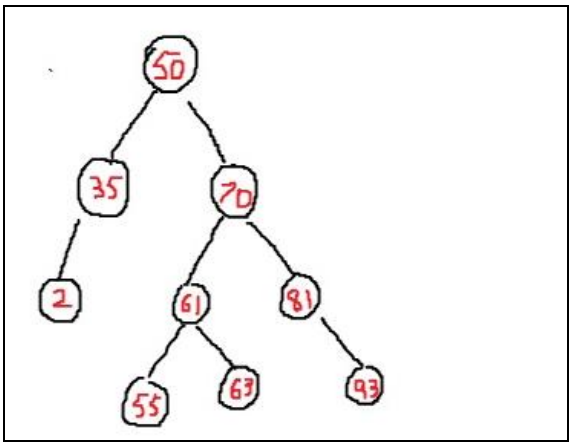


7. Add a method called **pathSums** to class bst. It should return a vector containing the sums of the keys along each path of the tree, starting from the root to each leaf. For example, for the tree given in problem 4, the method should return a vector containing [18, 20, 76, 126].
8. Add the **operator ==** to class bst. When it is used like bst1==bst2, it should return true if bst1 and bst2 have the exact **same data**, even though their structures may be different. This should take O(n).
9. 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).
10. Add a method called **isSubset**. When used like bst1.isSubset(bst2), it should return true if the data of bst2 occurs in bst1, even if bst2 is not an exact subtree of bst1. This should take O(n).
11. [This function needs knowledge from the lecture on Tuesday May 18] Update the method **searchAndPromote** so that after a single search the key being searched should be promoted to the root of the tree. This should take O(h). Moreover: since you need to move all the way up to root, you will need the pointers to all the ancestors of the node with the key. To achieve this, you should update the getAccess method so that it now provides the pointers to all ancestors,

not just the parent and grandparent. getAccess should return these pointers in a stack passed to it as a reference parameter.

12. 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.
13. Add a recursive 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. This method should take no more than O(n) time.

14. Write a method called **lowestCommonAncestor** in the bst class. This method should return the key that is the lowest (nearest) common ancestor of two keys passed in parameters. For example, in the following bst, the lowest common ancestor of 81 and 55 is 70; the lowest common ancestor of 55 and 63 is 61; while the lowest common ancestor of 2 and 63 is 50, etc. If one of the keys is an ancestor of the other than that key should be reported. For example, the lowest common ancestor of 70 and 91 is 70, etc.



*** END ***