CS29002 Algorithms Laboratory

Assignment No: 3

Last date of submission: 10–August–2016

Let T be a binary search tree (BST) storing integer-valued keys. For any node v, the notation $|v|$ stands for the size of (that is, the number of nodes including v in) the subtree rooted at v. Let $\mathit{left}(v)$ and $\mathit{right}(v)$ denote the left and right children of v, so $|\mathit{left}(v)|$ and $|\mathit{right}(v)|$ are the sizes of the two subtrees of v. We have $|\mathit{left}(v)| = 0$ (or $|\mathit{right}(v)| = 0$) if v does not have a left (or right) child. If r is the root of T, we also denote $|T| = |r|$.

A BST T is called *perfectly balanced* if for every node v in T, we have $|\mathit{left}(v)| - |\mathit{right}(v)| \in \{0,1\}$. Let us now relax this very strict requirement of perfect balance, and attempt to maintain the property that no subtree of a node v should contain less than one-third and more than two-thirds of the nodes of the tree rooted at v. More precisely, for every node v, we would like to maintain

$$|\mathit{left}(v)| \leqslant \frac{2}{3}|v|, \quad \text{and} \quad |\mathit{right}(v)| \leqslant \frac{2}{3}|v|. \tag{1}$$

Height-balanced trees introduced in the class store extra information at every node (color in red-black trees, and balance factor in AVL trees). Now, we will manage without storing the subtree size at every node. Each node would store only an integer key, and three pointers (left, right, and parent). Parent pointers can be avoided by using the recursion stack. For simplicity, you are allowed to use parent pointers. We externally maintain two counts: $n = |T|$, and m (a maximum value of n, needed during deletion). In short, the tree is specified by T (a pointer to the root node), and the two counts n and m.

The tree uses standard BST insertion and deletion procedures with the exception that from time to time (but not during every insert/delete), it perfectly balances certain subtrees for restoring the conditions (1).

Part 1: Write functions for performing the following standard operations on BSTs. A node is passed to a function by passing a pointer to it. The running time of each of these functions should be linear in the size of the input tree or subtree.

- $\mathit{size}(u)$ to compute and return the size $|u|$ of the subtree rooted at the node u of T.
- $\mathit{height}(T)$ to compute and return the height of T.
- $\mathit{inorder}(T)$ to print the inorder listing of the keys stored in T.
- $\mathit{destroy}(v)$ to free the memory allocated to the nodes of the subtree rooted at v.

Part 2: Write a function $\mathit{rebuild}(v,s)$ to perfectly balance the subtree rooted at v. The size $|v|$ of the subtree is s, and is assumed to be known beforehand. Use a local array $A[\,]$ to store the sorted sequence of the keys stored in the subtree. The subtree is then destroyed. A perfectly balanced BST is then prepared from the sorted array $A[\,]$. For this, the median is placed at the root of the tree, and the left and right subtrees are built recursively. The function should return a pointer to the root of the new balanced tree. You may write one or more additional functions that are called by $\mathit{rebuild}()$.

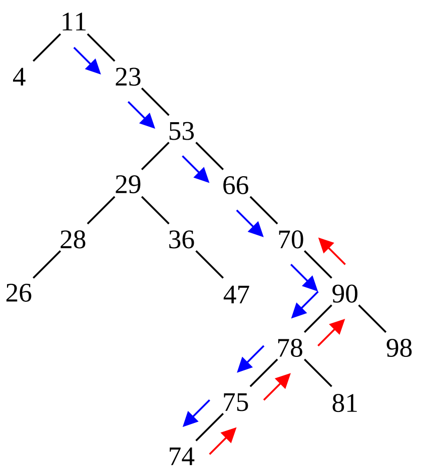
Part 3: Write a function $\mathit{insertKey}(T,x,n,m)$ in order to insert a key x to a tree T. If insertion is successful, the count $n = |T|$ increases by one. The maximum count m may also change during insertion. So you should pass n and m (pointers to them actually) to the function. A pointer to the root of the tree after insertion is to be returned. You may pass an additional flag as a directive whether or not to print a diagnostic message about rebuilding (see $\mathit{main}()$).

The insertion algorithm first inserts x in T using the standard BST insertion procedure. That is, x is searched in T. If it is already present, the old tree is returned. Otherwise, x is inserted at a new leaf node; call it l. Increment n, and if n exceeds m, set m to n. The insertion may violate (one of) the conditions (1). Since we have not maintained the size information at the nodes, we check for the validity of an alternative condition:

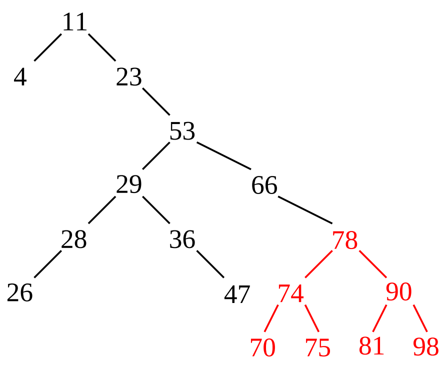
$$\mathit{depth}(l) \leqslant 1 + \left\lfloor \log_{3/2} |T| \right\rfloor. \tag{2}$$

Here, *depth*(l) is the length of the path from the root (call it r) to l. This must be the new height of the tree, since the tree was height-balanced before insertion. This is to be computed by maintaining a counter during the BST insertion, not by calling *height*(T). The new condition (2) is loosely related (but not equivalent) to the original conditions (1). If the depth check fails, we follow the parent pointers from l to r. Let v be the first (that is, deepest) node we encounter on the upward path, at which one of the conditions (1) is violated (we must have $v \neq l$, since a leaf is always balanced). Let $s = |v|$. The subtree rooted at v is replaced by a perfectly balanced tree using the function *rebuild*() of Part 2.

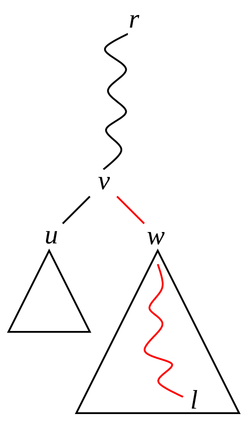
The following figure demonstrates the working of the insertion procedure. This corresponds to the example given in the sample output. The insertion path is shown by blue arrows. 74 is inserted at depth 8, whereas $1 + \lfloor \log_{3/2} 17 \rfloor = 1 + \lfloor 6.987563\ldots \rfloor = 7$. We move up the tree (red arrows), and at node 70 detect a violation of (1) (see Part (a) of the figure). Rebuilding the subtree rooted at 70 gives the tree of Part (b). Notice that the tree is unbalanced under the strict conditions (1), both before and after the rebuilding. But this is not a problem, since the eventual goal is to produce a height-balanced tree, and the condition (2) ensures a height of at most $1 + \lfloor 1.71 \log_2 |T| \rfloor$.



(a) Before rebuilding



(b) After rebuilding



(c) Size calculations

Insertion including rebuilding must run in $O(\mathit{height}(T) + s)$ time. When the upward movement from l to r is at a node v, the size of one of its subtrees (rooted at w, the previous node on the l-to-r path) is known. We get the size of the other subtree by calling *size*(u) at the sibling node u of w. We have $|v| = 1 + |u| + |w|$. Part (c) of the above figure demonstrates this incremental size calculation during the upward walk.

Part 4: Write a function *deleteKey*(T, x, n, m) to delete a key x from T. The counts n and m are as during insertion. You may pass an additional flag to optionally print whether rebuilding is done or not (see *main*()). First, delete x from T following the standard BST deletion procedure. If deletion is successful (x was present in T), decrement n, and check for the condition

$$n \geqslant \frac{2}{3} m. \tag{3}$$

It this condition is not satisfied, the entire tree T is rebuilt, that is, converted to a perfectly balanced tree by using the function *rebuild*() (with the root of T is passed as v). After this rebuilding, m is set to n.

The *main*() function:

- Create an initially empty BST T. Read a small (positive) integer *nsml* from the user.
- Read *nsml* keys from the user, and insert the keys one by one to T. After each insertion, print the height and the inorder listing of T. Indicate whether a subtree is rebuilt during an insertion. If so, print the size of the subtree rebuilt.
- Read *nsml* keys from the user, and delete the keys one by one from T. After each deletion, print the height and the inorder listing of T. Indicate whether T is rebuilt during a deletion.
- Destroy T to an empty tree.

- Read a large integer *nins* (like a few thousands) from the user. Insert the integers $1,2,3,\ldots,nins$ one by one to T. Do not print T or the rebuilding message after each insertion. Instead print only the height of T after every hundred insertions.
- Read a large integer *ndel* (meaningfully, $\leqslant nins$) from the user. Delete the integers $1,2,3,\ldots,ndel$ one by one from T. Do not print T or the rebuilding message after each deletion. Instead print only the height of T after every hundred deletions.

Sample output: In the sample output below, $[R\quad s]$ indicates that a rebuilding of a subtree of size s was done. Absence of this text implies that no rebuilding was necessary.

```
nsml = 20
+++ insert(11):         Height =  0:  11
+++ insert(23):         Height =  1:  11 23
+++ insert(53):         Height =  2:  11 23 53
+++ insert(29):         Height =  3:  11 23 29 53
+++ insert(66):         Height =  3:  11 23 29 53 66
+++ insert( 4):         Height =  3:   4 11 23 29 53 66
+++ insert(70):         Height =  4:   4 11 23 29 53 66 70
+++ insert(36):         Height =  4:   4 11 23 29 36 53 66 70
+++ insert(47):         Height =  5:   4 11 23 29 36 47 53 66 70
+++ insert(90):         Height =  5:   4 11 23 29 36 47 53 66 70 90
+++ insert(78):         Height =  6:   4 11 23 29 36 47 53 66 70 78 90
+++ insert(28):         Height =  6:   4 11 23 28 29 36 47 53 66 70 78 90
+++ insert(26):         Height =  6:   4 11 23 26 28 29 36 47 53 66 70 78 90
+++ insert(75):         Height =  7:   4 11 23 26 28 29 36 47 53 66 70 75 78 90
+++ insert(90):         Height =  7:   4 11 23 26 28 29 36 47 53 66 70 75 78 90
+++ insert(98):         Height =  7:   4 11 23 26 28 29 36 47 53 66 70 75 78 90 98
+++ insert(81):         Height =  7:   4 11 23 26 28 29 36 47 53 66 70 75 78 81 90 98
+++ insert(74): [R  7]  Height =  6:   4 11 23 26 28 29 36 47 53 66 70 74 75 78 81 90 98
+++ insert( 4):         Height =  6:   4 11 23 26 28 29 36 47 53 66 70 74 75 78 81 90 98
+++ insert(89):         Height =  7:   4 11 23 26 28 29 36 47 53 66 70 74 75 78 81 89 90 98

+++ delete(23):         Height =  6:   4 11 26 28 29 36 47 53 66 70 74 75 78 81 89 90 98
+++ delete( 4):         Height =  6:  11 26 28 29 36 47 53 66 70 74 75 78 81 89 90 98
+++ delete(89):         Height =  5:  11 26 28 29 36 47 53 66 70 74 75 78 81 90 98
+++ delete(90):         Height =  5:  11 26 28 29 36 47 53 66 70 74 75 78 81 98
+++ delete(26):         Height =  5:  11 28 29 36 47 53 66 70 74 75 78 81 98
+++ delete(74):         Height =  5:  11 28 29 36 47 53 66 70 75 78 81 98
+++ delete( 4):         Height =  5:  11 28 29 36 47 53 66 70 75 78 81 98
+++ delete(78): [R 11]  Height =  3:  11 28 29 36 47 53 66 70 75 81 98
+++ delete(66):         Height =  3:  11 28 29 36 47 53 70 75 81 98
+++ delete(90):         Height =  3:  11 28 29 36 47 53 70 75 81 98
+++ delete(75):         Height =  3:  11 28 29 36 47 53 70 81 98
+++ delete(47):         Height =  3:  11 28 29 36 53 70 81 98
+++ delete(53): [R  7]  Height =  2:  11 28 29 36 70 81 98
+++ delete(36):         Height =  2:  11 28 29 70 81 98
+++ delete(98):         Height =  2:  11 28 29 70 81
+++ delete(11): [R  4]  Height =  2:  28 29 70 81
+++ delete(29):         Height =  1:  28 70 81
+++ delete(70): [R  2]  Height =  1:  28 81
+++ delete(81): [R  1]  Height =  0:  28
+++ delete(28):         Height = -1:

nins = 600
+++ Inserting increasing sequence of 600 keys
    Height after  100 iterations: 12
    Height after  200 iterations: 13
    Height after  300 iterations: 14
    Height after  400 iterations: 15
    Height after  500 iterations: 16
    Height after  600 iterations: 16

ndel = 500
+++ Deleting increasing sequence of 500 keys
    Height after  100 deletions: 12
    Height after  200 deletions: 11
    Height after  300 deletions:  8
    Height after  400 deletions:  8
    Height after  500 deletions:  6
```

Submit a single C/C++ source file. Do not use global/static variables.