CS29003 ALGORITHMS LABORATORY
Assignment No: 2
Last Date of Submission: 07-Aug-2014

In this assignment, you work with a type of balanced binary search trees. The balancing condition being very demanding (compared to what happens in height-balanced trees), the insert and delete functions can be inefficient. However, the height of the tree will always have the least possible value.

A node in the tree contains an integer value v, a counter c, and three pointers L, R and P. The value v is the key stored at the node. The count c is the size (number of nodes) of the subtree rooted at the node (including the node itself). Finally, the pointers point respectively to the left child, the right child, and the parent. For the root node, the parent pointer is assumed to be NULL.

Part I

Write a function *tryinsert*() in order to make an insertion attempt of a key x in a tree T. Use the standard BST insertion procedure for this insertion attempt. The function returns a pointer to the root of the modified tree T after the insertion. If x was already present in T, the returned tree is the same as the tree before insertion. The function also returns (via a proper argument) a pointer q to the parent of the inserted node. If x was already present in T, then no node is inserted, and q receives the value NULL. A new key is always inserted at a new leaf node, so its initial count is always 1.

After a successful insertion, we need to update the counts of subtrees. Only the nodes on the insertion path are affected. If q is not NULL (that is, a real insertion took place), then traverse from q to the root via the parent pointers, and increment the count at every node on the path by one. Write a function *incrcount*() which takes the pointer q as the only argument, and performs this count updating. If q is NULL, no insertion took place (or the root is inserted to a NULL tree), so no node counts are affected, and this function need not be called.

Part II

Write a function *trydelete*() in order to make a deletion attempt of a key x in a tree T. If x is not present in T, then a pointer to the root of the original tree T is returned, and a pointer argument q receives the value NULL. If x is present, then the standard BST deletion procedure is invoked, a pointer to the root of the new tree is returned, and q is assigned a pointer to the parent of the deleted node.

The purpose of q is to readjust (decrement by one) the count at every node on the path from q to the root. Write a function *decrcount*() that implements this readjustment. If q is NULL, no deletion took place (or the root is deleted), so no node counts are affected, and this function need not be called.

Part III

We now introduce the balancing condition. We call a BST T a *perfectly balanced binary search tree* (or *PB-BST*) if the absolute difference of the counts of the nodes of the two subtrees at every node is never more than 1. If we define the imbalance at a node v as count(right(v)) – count(left(v)), then the only allowed values of imbalance(v) are 0, +1, and –1.

A successful insertion/deletion may throw a PB-BST T out of balance. Since T was perfectly balanced before the insertion/deletion, some nodes can now have an imbalance of +2 or –2. All such nodes lie on the insertion/deletion path (from the pointer q returned in Part I or II to the root). This is detected after the count adjustments are made. Write a function *rebalance*() that repairs all imbalance cases, and restores perfect balance back to T.

The re-balancing algorithm is explained now. Suppose that at some node v, the imbalance is +2. This means that for some $k \geq 0$, the left subtree of v contains k nodes, and the right subtree $k + 2$ nodes. We insert the value at v in the left subtree of v. Next, we locate in the right subtree of v the immediate successor of the value stored at v. Since the size of the right subtree is $k + 2$, it is in particular non-empty, and the immediate successor can always be found. Copy this immediate successor value to the node v. Finally, delete this immediate successor value from the right subtree. After this re-balancing, both the subtrees of v contain exactly $k + 1$ nodes, and we proceed to the parent of v for possible re-balancing.

The case of imbalance(v) = –2 is symmetrically handled.

Append a call of this re-balancing function after the count-adjustment function during each insertion/deletion. Notice that each insertion or deletion in the re-balancing stage may additionally throw certain subtrees out of balance. These must be recursively handled.

Sample Output

You should first read the number of integers to insert, call it *nins*. For each of the *nins* integers read, call the functions *tryinsert*(), *incrcount*() and *rebalance*() (the last two if needed), and print the tree *before* you read and insert the next integer. The pre-order listing of the tree should be printed with each line specifying the following items at a node *v*:

value(*v*) (count(*v*)) → value(left(*v*)) (count(left(*v*))), value(right(*v*)) (count(right(*v*)))

For example, the line

```
53 (14) -> 21 (6), 82 (7)
```

means that a node contains the key 53 and count 14. Its left and right children store the keys 21 and 82 and have respective counts 6 and 7.

After the insertion of *nins* integers, read the number of integers to delete, call it *ndel*. For each of the *ndel* integers read, call the functions *trydelete*(), *decrcount*() and *rebalance*() (the last two if needed), and print the tree *before* you read and delete the next integer.

Here, we demonstrate insertion and deletion in a small PB-BST. Let, at some point of time, the tree contain eight nodes, and the preorder listing of these nodes is:

```
26 (8) -> 20 (3), 55 (4)
   20 (3) -> 4 (1), 24 (1)
      4 (1) -> -1 (0), -1 (0)
      24 (1) -> -1 (0), -1 (0)
   55 (4) -> 49 (1), 74 (2)
      49 (1) -> -1 (0), -1 (0)
      74 (2) -> 70 (1), -1 (0)
         70 (1) -> -1 (0), -1 (0)
```

Let us insert 37 in this tree. This makes one violation of the PB-BST property (at Node 26 where the imbalance becomes +2). So two recursive calls are made. First, 26 is inserted to the left subtree (rooted at Node 20). The immediate successor of 26 is 37. So 37 replaces 26 at the root, and 37 is deleted from the right subtree (rooted at Node 55).

The subtrees are printed after each call.

```
+++ Insertion of 26 under 20

20 (4) -> 4 (1), 24 (2)
   4 (1) -> -1 (0), -1 (0)
   24 (2) -> -1 (0), 26 (1)
      26 (1) -> -1 (0), -1 (0)

+++ Deletion of 37 under 55

55 (4) -> 49 (1), 74 (2)
   49 (1) -> -1 (0), -1 (0)
   74 (2) -> 70 (1), -1 (0)
      70 (1) -> -1 (0), -1 (0)
```

After these adjustments, the tree is perfectly balanced, and looks like the following:

```
37 (9) -> 20 (4), 55 (4)
   20 (4) -> 4 (1), 24 (2)
      4 (1) -> -1 (0), -1 (0)
      24 (2) -> -1 (0), 26 (1)
         26 (1) -> -1 (0), -1 (0)
   55 (4) -> 49 (1), 74 (2)
      49 (1) -> -1 (0), -1 (0)
      74 (2) -> 70 (1), -1 (0)
         70 (1) -> -1 (0), -1 (0)
```

Now, the BST deletion of 37 brings 49 to the root, and deletes the leaf node 49. At Node 55, the PB-BST property is violated. This requires the adjustments: insertion of 55 in the left subtree (here it is NULL), replacement of 55 by its immediate successor 70, and deletion of 70 from the right subtree (rooted at Node 74).

```
+++ Insertion of 55 under NULL

55 (1) -> -1 (0), -1 (0)

+++ Deletion of 70 under 74

74 (1) -> -1 (0), -1 (0)
```

One node up the tree, we encounter Node 37, where re-balancing is not necessary. Since Node 37 is the root, the deletion procedure stops, and the final balanced tree is:

```
49 (8) -> 20 (4), 70 (3)
   20 (4) -> 4 (1), 24 (2)
      4 (1) -> -1 (0), -1 (0)
      24 (2) -> -1 (0), 26 (1)
         26 (1) -> -1 (0), -1 (0)
   70 (3) -> 55 (1), 74 (1)
      55 (1) -> -1 (0), -1 (0)
      74 (1) -> -1 (0), -1 (0)
```

Look at sample output files linked from the laboratory page. The files demonstrate that recursion levels can be quite high (near the height of the tree).

Submit two separate files.

1. A program that implements Parts I and II alone. Here, no re-balancing attempts are made, so the imbalance at a node is allowed to be more than +2 or less than –2.

2. A program that additionally implements Part III. Every tree printed in this program should be a PB-BST. You may use the same Parts I and II as in your first file.