CS29003 Algorithms Laboratory, Autumn 2013–14

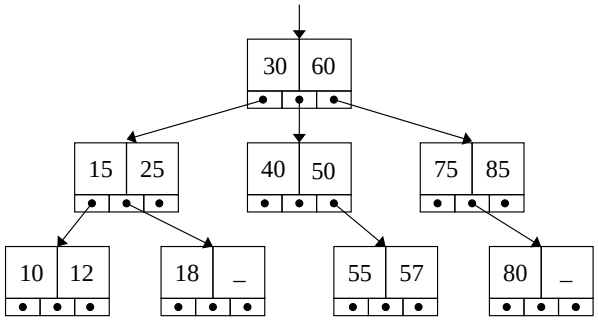
Lab Test

Maximum marks: 25 Date: 23-Oct-2013 Duration: 2 hours

In a *ternary search tree*, each node stores two key values (k_1 and k_2) and three pointers (L, M and R). Let x be any key stored in the left subtree of a node, y any key stored in the middle subtree, and z any key stored in the right subtree. We must have:

$$x < k_1 < y < k_2 < z\,.$$

The tree does not store duplicate keys. Leaf nodes are allowed to store single keys as k_1. The other keys k_2 are left undefined. For example, when the tree consists of an odd number of keys, at least one node must contain only one key, and any such node must be a leaf. The following figure shows a ternary search tree with undefined keys shown as _.



Insertion in a ternary search tree proceeds as follows. Let x be the key to be inserted. Starting from the root, we attempt to locate x in the tree by making a three-way branching at each visited node. If x is found in the tree, no changes are made, and the original tree is returned.

Now, suppose that x does not exist in the tree. There are two ways the search can fail. First, the search reaches a leaf node storing only one key (different from x). Since that leaf can accommodate the new key x, we populate its two keys k_1 and k_2. Depending upon the value of x in comparison with the earlier key, this adjustment is made. Think about the insertion of 17 or 19 in the tree of the above figure.

The second way the search fails is when a NULL pointer is followed from a node storing both k_1 and k_2. For example, think about the insertion of 35, 45 or 58 in the above tree. In this case, a new leaf is created, its first key k_1 stores x, its second key k_2 is left undefined, and the leaf is connected to the last node in the search path by replacing the NULL pointer mentioned above by a pointer to the new leaf node.

A recursive function similar to the in-order listing of a binary search tree prints the keys stored in a ternary search tree in the sorted order.

Write a C/C++ program containing the following functions.

- A function implementing the ternary-search-tree insertion procedure described above.
- A function for the sorted printing of the keys stored in a ternary search tree. Do not print the undefined keys. Since you are going to insert only positive integers, you can take 0 or -1 as the undefined key.
- A main function that first reads the total number n of insertions in the tree. An empty ternary search tree is created. Subsequently, n randomly generated positive integers are inserted in the tree. After each insertion, the keys in the tree are printed in the sorted order. An insertion attempt fails to change the tree if an existing key value is attempted to be inserted. You do not need to worry about that. Just make n insertion attempts.

Sample output

```
n = 50
+++ Insert 986 done
 986
+++ Insert 130 done
 130 986
+++ Insert 945 done
 130 945 986
+++ Insert 90 done
  90 130 945 986
+++ Insert 665 done
  90 130 665 945 986
...
```

A challenging sequel

If you are done with the above program well before the end of the test, make a copy of your program to another file, and change the copy. This part is not meant for submission. You will not get any extra credit for solving this part, completely or partially. In the submission server, submit the file solving only the parts given on Page 1. If you can make this part work, e-mail me (abhij@cse.iitkgp.ernet.in and SadTijihba@gmail.com) a complete and error-free code by 4:30pm today, and earn a chocolate bar. So, here you go.

Implement deletion in a ternary search tree, that is, write a function that, upon the input of a ternary search tree T and a key value x, returns a ternary search tree with x deleted from T (if it existed in T at all). Don't forget that *only* leaf nodes are allowed to have undefined second keys k_2. All internal nodes must contain two valid keys. And, of course, your function must run in $O(h(T))$ time, where $h(T)$ is the height of T. The algorithm is yours.

To demonstrate the working of your deletion routine, start with the tree created by n insertion attempts (see Page 1), and keep on deleting one of the keys stored at the root. You may alternately or randomly select the deletion key from the two possibilities at the root node. After every deletion, print the tree. Repeat until the tree becomes empty.

If you have no segmentation faults until the end, claim your chocolate from me.