CS29003 Algorithms Laboratory

Assignment No: 6

Last date of submission: 06–March–2018

This assignment deals with binary search trees. Each node of the tree stores an integer key value and two child pointers. Do not store parent pointers or any additional field in a node. Let T be a BST with n nodes in this representation. By $h(T)$, we denote the height of T.

Part 1: Write a function **`buildBST(A,n)`** that takes as input an array A with n integers. Assume that A stores the valid preorder listing of the keys in T. The function should reconstruct the tree T from A. Implement the following $O(n \log n)$-time algorithm. $A[0]$ is the key stored at the root of the tree. Apply a binary-search-like procedure to identify the two indices l, r such that $r = l + 1$, $A[l] < A[0]$, and $A[r] > A[0]$. Build the left subtree recursively from the subarray $A[1 \ldots, l]$, and the right subtree recursively from the subarray $A[l+1, \ldots, n-1]$. If no such indices l, r exist, one subtree or both is/are NULL.

Part 2: Write a function **`preorder(T)`** for the preorder printing of the keys in T. This function is the first step to verify the correctness of **`buildBST`**.

Part 3: Since different binary trees may have the same preorder listing, the test of Part 2 is not sufficient. You need to verify that you have constructed a binary search tree. Write a function **`isBST(T,l,r)`** to verify whether a binary tree T is a BST. We pass two additional bounds l, r to the function. If x is the key stored in the root of T, we must have $l < x < r$. When you make a recursive call on the left subtree, these bounds change to (l, x). For the recursive call on the right subtree, the bounds change to (x, r). For the outermost call (in **`main()`**), pass $l = -\infty$ and $r = +\infty$.

Part 4: Write a function **`inorderSuccessor(T,x)`** to return the node (a pointer to the node, to be more precise) in T, that stores the immediate successor of the key x. If T does not store x, or if x is the largest key stored in T, this successor does not exist (return NULL in these cases). Otherwise, the node stores the smallest key larger than x. Use the following $O(h(T))$-time algorithm.

Let p be the node storing x. If the right child of p exists, then the node q storing the immediate successor is the smallest key in the right subtree of p. If the right child of p does not exist, locate the node q, of which p is the immediate predecessor. This is an ancestor of p. You do not have parent pointers. So you should modify the search procedure for x so that the last left turn is always remembered. If the traversal never makes a left turn, x is the largest key in T, and the immediate successor does not exist. Otherwise, return q.

Part 5: The preorder successor of a key x stored in T is the key printed immediately after x in the preorder listing of T. Implement an $O(h(T))$-time function **`preorderSuccessor(T,x)`** to return (a pointer to) the node storing the preorder successor of x. If T does not store x, or if x is the last key printed in the preorder listing, then this successor does not exist, and your function should return the NULL pointer. No credit will be given in output/code (even if correct) if you use the array A of Part 1. Use only the constructed tree T. Appropriately handle the absence of parent pointers.

The *main*() function

- Read the number n of nodes in the BST T.
- Read and store in an array A the preorder listing of the keys in T. Reconstruct T by calling **`buildBST`**. Verify the correctness of your reconstruction by first calling **`preorder`** and then calling **`isBST`**.
- Read three keys from the user. Call **`inorderSuccessor`** for each of these keys, and print the keys pointed to by the returned pointers.
- Read three keys from the user. Call **`preorderSuccessor`** for each of these keys, and print the keys pointed to by the returned pointers.

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

Sample output

```
    n = 100

+++ Array storing preorder listing of keys
    490 303 255 205 127 193 132 177 147 142 144 170 173 198 202 246 220 215 219
    245 236 242 286 290 289 287 350 312 307 339 335 323 321 315 326 404 353 362
    360 356 394 367 409 487 484 423 411 412 481 489 846 504 497 493 529 510 508
    509 520 523 835 582 581 560 562 563 804 712 583 687 608 607 631 617 653 795
    716 732 764 747 744 785 827 820 819 830 831 970 910 885 863 903 904 908 962
    921 913 948 976 973

+++ Preorder listing of the keys in the constructed tree
    490 303 255 205 127 193 132 177 147 142 144 170 173 198 202 246 220 215 219
    245 236 242 286 290 289 287 350 312 307 339 335 323 321 315 326 404 353 362
    360 356 394 367 409 487 484 423 411 412 481 489 846 504 497 493 529 510 508
    509 520 523 835 582 581 560 562 563 804 712 583 687 608 607 631 617 653 795
    716 732 764 747 744 785 827 820 819 830 831 970 910 885 863 903 904 908 962
    921 913 948 976 973

+++ The constructed tree is a BST...

    Inorder successor of 353 is 356
    Inorder successor of 177 is 193
    Inorder successor of 976 does not exist

    Preorder successor of 147 is 142
    Preorder successor of 563 is 804
    Preorder successor of 973 does not exist
```