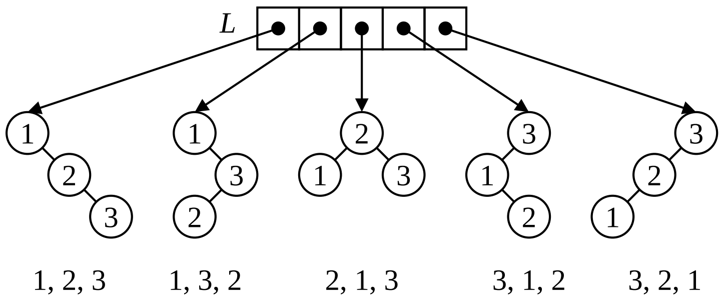
CS69001 Computing Laboratory – I

EVEN PC **Lab Test 1** **Date: 09–September–2019**

In this exercise, you work with binary search trees (BSTs). Each of the trees stores the keys $1, 2, 3, \ldots, n$. Your task is to generate and work with all possible structurally different BSTs storing these keys. The structure of the BST can be completely specified by the preorder listing of its keys. Therefore, if you can generate all valid preorder listings of $1, 2, 3, \ldots, n$, you can construct all the desired trees from these listings. The following figure describes all structurally different BSTs storing the keys $1, 2, 3$. The root pointers are stored in an array L. The array should be sorted with respect to the preorder listings of the trees. Notice that only the trees are to be stored, but not their preorder listings (shown below the trees for illustration only).



Part 1: The data types (4)

Define a data type to store a node in a BST. Each node should consist of an integer key, two child pointers (left and right), and nothing else. You need to store a set of trees. Use an array L (not vector) to store the pointers to the root nodes of the trees.

Part 2: Count of structurally different trees (4)

Let C_n denote the count of structurally different BSTs with n nodes. It can be proved that C_n is the n-th Catalan number, and that the Catalan numbers can be defined as follows.

$$\begin{aligned} C_0 &= 1, \\ C_n &= C_0C_{n-1} + C_1C_{n-2} + C_2C_{n-3} + \cdots + C_{n-2}C_1 + C_{n-1}C_0 \text{ for } n \geqslant 1. \end{aligned}$$

Write an <u>efficient</u> function *treecount*(n) to return the value of C_n for an integer input of $n \geqslant 0$. This count will be used later on multiple occasions.

Part 3: Check for valid preorder listings (4)

Write a function *ispreorder* that takes as input an array A storing a permutation of $1, 2, 3, \ldots, n$ (no need to check whether it is a permutation). The function should return true/false depending upon whether the permutation stored in A is a valid preorder listing of the keys in a BST. Implement the following algorithm.

Since A purportedly stores the preorder listing of a BST, the first element (that is, $A[0]$) must be the key stored at the root node. Call this key r. The array A can be decomposed as rA_LA_R, where A_L and A_R are the preorder listings of the left and right subtrees of the root. Notice that one or both of A_L and A_R may be empty. Compute the maximum L_{max} of A_L, and the minimum R_{min} of A_R. If A_L (resp. A_R) is empty, take $L_{max} = -\infty$ (resp. $R_{min} = +\infty$). For A to store a valid preorder listing, we must have $L_{max} < r < R_{min}$. Finally, check recursively whether A_L and A_R are valid preorder listings of the two subtrees.

Part 4: Construct the tree from a valid preorder listing (8)

Write a function *pre2BST* that takes an input array A storing a permutation of $1, 2, 3, \ldots, n$, which is a <u>valid</u> preorder listing of the keys of a BST. The function should create the BST, of which the input permutation is the preorder listing, and return a pointer to the root node of the tree created. Follow an algorithm similar to that described in Part 3. Since *pre2BST* assumes that A stores a valid listing of the keys, you do not need to check for the validity again. You should instead create the different nodes, and link them appropriately. For example, the key r is to be stored in a new node allocated for the root. Its left and right child pointers are set to the roots of the two subtrees constructed recursively on A_L and A_R.

Part 5: Generate all structurally different BSTs **(8)**

Write a function *allBST* to generate all permutations of $1,2,\ldots,n$ in the increasing order. For example, if $n = 3$, the permutations should be generated in the order $[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]$. After each permutation is generated, check whether this is a valid preorder listing (using *ispreorder* of Part 3). If so, create the BST with this listing (use *pre2BST* of Part 4), and append the pointer to the root of the tree to the array L. Otherwise, discard this permutation. This function should print how many trees it creates. This count should match C_n as computed in Part 2.

Part 6: Printing the trees **(2)**

Write a function *preorder* to print the preorder listing of the keys of a BST T.

Part 7: Binary search in the array of trees **(8)**

Write a function *binsearch* that, given the array L (of size C_n) and an array A (of size n) storing a permutation of $1,2,3,\ldots,n$, finds out whether A is the preorder listing of a tree in L. Since L is sorted with respect to the preorder listing of the trees, a binary search suffices. For comparing A with the preorder listing of a tree $T = L[i]$, store the preorder listing of T in an array B, and compare A and B element by element.

Part 8: The *main*() function **(4)**

The user enters an integer $n \in \{3,4,5,\ldots,10\}$. Call *treecount* (Part 1) to get the value of C_n. Print the count. Call *allBST* (Part 5) to create C_n BSTs and store their root pointers in L. For each tree $T = L[i]$, $i = 0,1,2,\ldots,C_n - 1$, print the preorder listing of the keys stored in T. Keep on generating random permutations of $1,2,3,\ldots,n$ until one is found which is a valid preorder listing. Likewise, keep on generating random permutations of $1,2,3,\ldots,n$ until one is found which is not a valid preorder listing. Check the validity (or otherwise) of each permutation by calling *binsearch*. Print the two permutations obtained as above. Notice that the user does not enter these permutations as inputs.

Output **(8)**

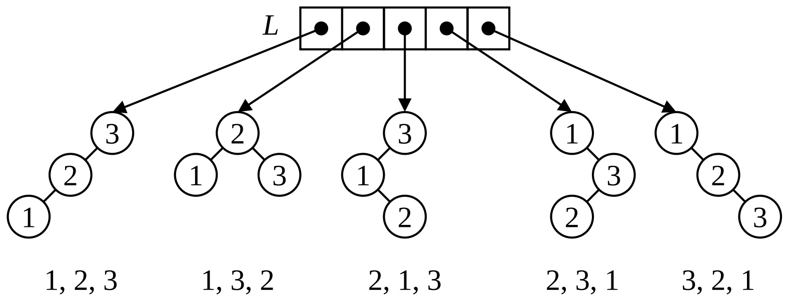
```
n = 4

+++ Count of BSTs = 14                                      [1 mark for correct count]

+++ 14 trees created                                        [1 mark for correct count]

+++ The preorder listings of the BSTs                       [4 marks]
    BST #    1 : 1 2 3 4
    BST #    2 : 1 2 4 3
    BST #    3 : 1 3 2 4
    BST #    4 : 1 4 2 3
    BST #    5 : 1 4 3 2
    BST #    6 : 2 1 3 4
    BST #    7 : 2 1 4 3
    BST #    8 : 3 1 2 4
    BST #    9 : 3 2 1 4
    BST #   10 : 4 1 2 3
    BST #   11 : 4 1 3 2
    BST #   12 : 4 2 1 3
    BST #   13 : 4 3 1 2
    BST #   14 : 4 3 2 1

+++ Search : 4 1 2 3 : Tree # 10 matches                    [1 mark]
+++ Search : 1 3 4 2 : Tree not found                       [1 mark]
```

Submit one C/C++ file. Do not use STL data structures. Write your name, roll no, and PC no as a comment.

CS69001 Computing Laboratory – I

ODD PC **Lab Test 1** **Date: 09–September–2019**

In this exercise, you work with binary search trees (BSTs). Each of the trees stores the keys $1,2,3,\ldots,n$. Your task is to generate and work with all possible structurally different BSTs storing these keys. The structure of the BST can be completely specified by the postorder listing of its keys. Therefore, if you can generate all valid postorder listings of $1,2,3,\ldots,n$, you can construct all the desired trees from these listings. The following figure describes all structurally different BSTs storing the keys $1,2,3$. The root pointers are stored in an array L. The array should be sorted with respect to the postorder listings of the trees. Notice that only the trees are to be stored, but not their postorder listings (shown below the trees for illustration only).



Part 1: The data types (4)

Define a data type to store a node in a BST. Each node should consist of an integer key, two child pointers (left and right), and nothing else. You need to store a set of trees. Use an array L (not vector) to store the pointers to the root nodes of the trees.

Part 2: Count of structurally different trees (4)

Let C_n denote the count of structurally different BSTs with n nodes. It can be proved that C_n is the n-th Catalan number, and that the Catalan numbers can be defined as follows.

$$\begin{aligned} C_0 &= 1, \\ C_n &= C_0C_{n-1} + C_1C_{n-2} + C_2C_{n-3} + \cdots + C_{n-2}C_1 + C_{n-1}C_0 \text{ for } n \geqslant 1. \end{aligned}$$

Write an efficient function $treecount(n)$ to return the value of C_n for an integer input of $n \geqslant 0$. This count will be used later on multiple occasions.

Part 3: Check for valid postorder listings (4)

Write a function *ispostorder* that takes as input an array A storing a permutation of $1,2,3,\ldots,n$ (no need to check whether it is a permutation). The function should return true/false depending upon whether the permutation stored in A is a valid postorder listing of the keys in a BST. Implement the following algorithm.

Since A purportedly stores the postorder listing of a BST, the last element (that is, $A[n-1]$) must be the key stored at the root node. Call this key r. The array A can be decomposed as A_LA_Rr, where A_L and A_R are the postorder listings of the left and right subtrees of the root. Notice that one or both of A_L and A_R may be empty. Compute the maximum L_{max} of A_L, and the minimum R_{min} of A_R. If A_L (resp. A_R) is empty, take $L_{max} = -\infty$ (resp. $R_{min} = +\infty$). For A to store a valid postorder listing, we must have $L_{max} < r < R_{min}$. Finally, check recursively whether A_L and A_R are valid postorder listings of the two subtrees.

Part 4: Construct the tree from a valid postorder listing (8)

Write a function *post2BST* that takes an input array A storing a permutation of $1,2,3,\ldots,n$, which is a valid postorder listing of the keys of a BST. The function should create the BST, of which the input permutation is the postorder listing, and return a pointer to the root node of the tree created. Follow an algorithm similar to that described in Part 3. Since *post2BST* assumes that A stores a valid listing of the keys, you do not need to check for the validity again. You should instead create the different nodes, and link them appropriately. For example, the key r is to be stored in a new node allocated for the root. Its left and right child pointers are set to the roots of the two subtrees constructed recursively on A_L and A_R.

Part 5: Generate all structurally different BSTs **(8)**

Write a function *allBST* to generate all permutations of $1,2,\ldots,n$ in the increasing order. For example, if $n=3$, the permutations should be generated in the order $[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]$. After each permutation is generated, check whether this is a valid postorder listing (using *ispostorder* of Part 3). If so, create the BST with this listing (use *post2BST* of Part 4), and append the pointer to the root of the tree to the array L. Otherwise, discard this permutation. This function should print how many trees it creates. This count should match C_n as computed in Part 2.

Part 6: Printing the trees **(2)**

Write a function *postorder* to print the postorder listing of the keys of a BST T.

Part 7: Binary search in the array of trees **(8)**

Write a function *binsearch* that, given the array L (of size C_n) and an array A (of size n) storing a permutation of $1,2,3,\ldots,n$, finds out whether A is the postorder listing of a tree in L. Since L is sorted with respect to the postorder listing of the trees, a binary search suffices. For comparing A with the postorder listing of a tree $T=L[i]$, store the postorder listing of T in an array B, and compare A and B element by element.

Part 8: The *main*() function **(4)**

The user enters an integer $n\in\{3,4,5,\ldots,10\}$. Call *treecount* (Part 1) to get the value of C_n. Print the count. Call *allBST* (Part 5) to create C_n BSTs and store their root pointers in L. For each tree $T=L[i]$, $i=0,1,2,\ldots,C_n-1$, print the postorder listing of the keys stored in T. Keep on generating random permutations of $1,2,3,\ldots,n$ until one is found which is a valid postorder listing. Likewise, keep on generating random permutations of $1,2,3,\ldots,n$ until one is found which is not a valid postorder listing. Check the validity (or otherwise) of each permutation by calling *binsearch*. Print the two permutations obtained as above. Notice that the user does <u>not</u> enter these permutations as inputs.

Output **(8)**

```
n = 4

+++ Count of BSTs = 14                                      [1 mark for correct count]

+++ 14 trees created                                        [1 mark for correct count]

+++ The postorder listings of the BSTs                      [4 marks]
    BST #    1 : 1 2 3 4
    BST #    2 : 1 2 4 3
    BST #    3 : 1 3 2 4
    BST #    4 : 1 3 4 2
    BST #    5 : 1 4 3 2
    BST #    6 : 2 1 3 4
    BST #    7 : 2 1 4 3
    BST #    8 : 2 3 1 4
    BST #    9 : 2 3 4 1
    BST #   10 : 2 4 3 1
    BST #   11 : 3 2 1 4
    BST #   12 : 3 2 4 1
    BST #   13 : 3 4 2 1
    BST #   14 : 4 3 2 1

+++ Search : 2 3 1 4 : Tree # 8 matches                     [1 mark]
+++ Search : 3 1 2 4 : Tree not found                       [1 mark]
```

Submit one C/C++ file. Do <u>not</u> use STL data structures. Write your name, roll no, and PC no as a comment.