Data Structures Lab
Session 9

Course: Data Structures (CS2001) **Semester:** Fall 2022
Instructor: Eman Shahid **T.A:** Insha

Note:

- Lab manual cover following topics
  {Tree, BST, Design and implement classes for binary tree nodes and nodes for general tree, Traverse the tree with the three common orders, Operation such as searches, insertions, and removals on a binary search tree and its applications}
- Maintain discipline during the lab.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

BST

KeyPoint: A Binary Search Tree (BST) is a binary tree with the following properties:

- The left subtree of a particular node will always contain nodes whose keys are less than that node's key.
- The right subtree of a particular node will always contain nodes with keys greater than that node's key. The left and right subtree of a particular node will also, in turn, be binary search trees

BST Insertion

Task-1:

Build functionality named autoGrader which will assist DS teacher to check students BST assignments such that if given tree is BST assign 10 points if not then assign 0.

Task-2: Implement main.cpp for the code provided such that a given array is passed to form a BST{15, 10, 20, 8, 12, 16, 25 }

Tree Traversals: Inorder, PreOrder, PostOrder

Pseudo code For Inorder Traversal (iteration)

```
1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current is
NULL
4) If current is NULL and stack is not empty then
     a) Pop the top item from stack.
     b) Print the popped item, set current = popped_item->right
     c) Go to step 3.
5) If current is NULL and stack is empty then we are done.
```

Task-3:

A. **Write recursive algorithms that perform preorder and inorder ,postorder tree walks.**

   Preorder Traversal approach.

   1. Visit Node.
   2. Traverse Node's left sub-tree.
   3. Traverse Node's right sub-tree

B. **Write the iterative code for inorder traversal.**

BST Deletion

BST Deletion

1) ***Node to be deleted is the leaf:*** Simply remove from the tree.

```
          50                            50
       /     \         delete(20)     /    \
      30      70       --------->    30     70
     /  \    /  \                     \    /  \
   20   40  60   80                    40  60   80
```

2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child

```
          50                            50
       /     \         delete(30)     /    \
      30      70       --------->    40     70
        \    /  \                          /  \
        40  60   80                       60   80
```

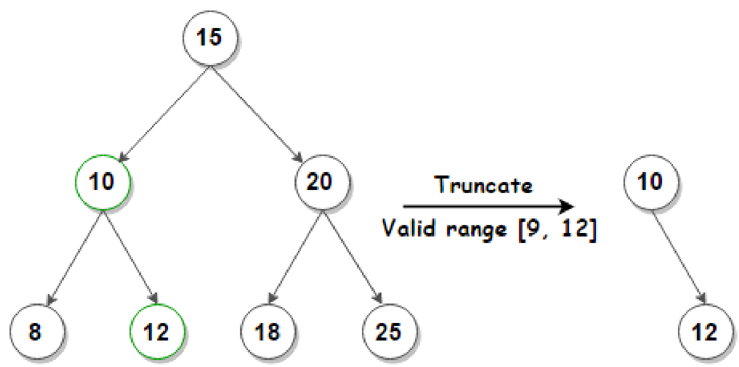
3) ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

```
          50                            60
       /     \         delete(50)     /    \
      40      70       --------->    40    70
             /  \                            \
            60   80                           80
```

The important thing to note is, inorder successor is needed only when the right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in the right child of the node.

Task:4

Given a BST and a range of keys(values), remove nodes from BST that have keys outside the given range



Task: 5:

Assume that a given binary tree stores integer values in its nodes. Write a recursive function that traverses a binary tree and prints the value of every node who's parent has a value that is a multiple of five. The function should be implemented in the main program.

Task: 6:

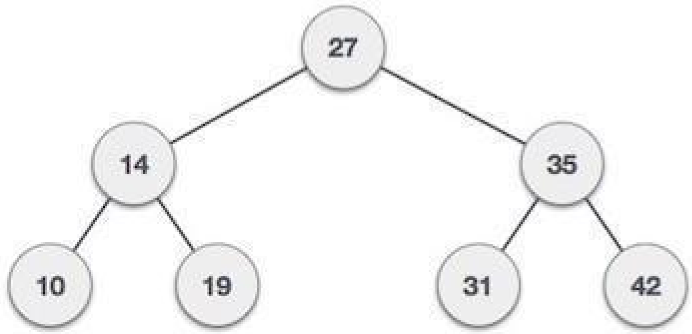
Write a recursive function named smallcount that, given the pointer to the root of a BST and a key K, returns the number of nodes having key values less than or equal to K. Function should be implemented in the main program.

Summary Discussion

Binary search tree (BST)

Binary search tree (BST) or a lexicographic tree is a binary tree data structure which has the following binary search tree properties:

- Each node has a value.
- The key value of the left child of a node is less than to the parent's key value.
- The key value of the right child of a node is greater than (or equal) to the parent's key value.
- And these properties hold true for every node in the tree.



- **Subtree**: any node in a tree and its descendants.
- **Depth of a node**: the number of steps to hop from the current node to the root node of the tree.
- **Depth of a tree**: the maximum depth of any of its leaves.
- **Height of a node**: the length of the longest downward path to a leaf from that node.
- **Full binary tree**: every leaf has the same depth and every nonleaf has two children.
- **Complete binary tree**: every level except for the deepest level must contain as many nodes as possible; and at the deepest level, all the nodes are as far left as possible.
- **Traversal**: an organized way to visit every member in the structure.

Traversals

The binary search tree property allows us to obtain all the keys in a binary search tree in a sorted order by a simple traversing algorithm, called an in order tree walk, that traverses the left sub tree of the root in in order traverse, then accessing the root node itself, then traversing in in-order the right sub tree of the root node.

The tree may also be traversed in preorder or post order traversals. By first accessing the root, and then the left and the right sub-tree or the right and then the left sub-tree to be traversed in preorder. And the opposite for the post order.
The algorithms are described below, with Node initialized to the tree's root.

• **Preorder Traversal**

1. Visit Node.
2. Traverse Node's left sub-tree.
3. Traverse Node's right sub-tree.

• **In-order Traversal**

1. Traverse Node's left sub-tree.
2. Visit Node.
3. Traverse Node's right sub-tree

• Post-order Traversal

1. Traverse Node's left sub-tree.
2. Traverse Node's right sub-tree.
3. Visit Node

Searching

We use the following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key *k,* `TREE-SEARCH` returns a pointer to a node with key *k* if one exists; otherwise, it returns `NIL`.

```
TREE-SEARCH (x, k)
if x = NIL or k = key[x]
    then return x
 if k < key[x]
    then return TREE-SEARCH (left[x], k)
    else return TREE-SEARCH (right[x], k)
```

The procedure begins its search at the root and traces a path downward in the tree, as shown in Figure 13.2. For each node *x* it encounters, it compares the key *k* with *key*[*x*]. If the two keys are equal, the search terminates. If *k* is smaller than *key*[*x*], the search continues in the left subtree of *x,* since the binary-search-tree property implies that *k* could not be stored in the right subtree. Symmetrically, if *k* is larger than *key*[*k*], the search continues in the right subtree. The nodes encountered during the recursion form a path downward from the root of the tree, and thus the running time of `TREE-SEARCH` is $O(h)$, where h is the height of the tree.

Reference:

http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap13.htm

Binary Search Tree		
Std Name:		**Std_ID:**
Lab1-Tasks	**Completed**	**Checked**
Task #1		
Task #2		
Task #3		
Task# 4		
Task# 5		
Task# 6		