Data Structures and Algorithms
LAB #7
Binary Search Trees II

Fall 2018

Objectives

By the end of this lab, the student should be able to:

- Write code for alternative linked implementation of binary trees.
- Write some simple application code using trees, stacks, and queues.

Binary Search tree - Alternative implementation

In the previous lab, you have learned how to implement a BST with Node class and BST class. This implementation considers a tree as a node with left and right nodes (or null).

In this lab an alternative implementation is give. The fact that a binary tree can be defined as: An empty tree or a tree with left and right sub-binary trees is adopted for the new implementation.

Binary Search Tree as tree with subtrees not subnodes:

Each node in the above tree is represent by class ***TreeNode*** below

```
template <typename T>
class BinarySearchTree
{
private:
   T Data;
   bool Empty; //true of tree is empty
   BinarySearchTree<T> *left;   //left subtree
   BinarySearchTree<T> *right;  //right subtree
public:
  // BinarySearchTree Member functions

};
```

Code 1 : BinarySearchTree class

The advantage of this implementation is that it does not need a utility fuction for each member function as in implementation of the previous lab.

See Examples➔BinarySearchTrees.sln➔ BinarySearchTrees

Practice Exercises

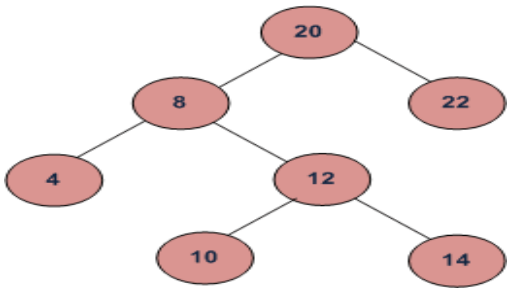
Part I:

Re-write all the functions from Lab6 - practice exercises section again as member functions of the BinarySearchTree class given in this lab.

Part II: Applications using trees, stack, and queues data structures.

In the following problems, you can use the STL classes of "stack" and "queue" instead of writing you own stack/queue

1. A breadth-first traversal (BFT) traverses the tree level by level. e.g. For the tree in the following figure:



   The BFT is: 20, 8, 22, 4, 12, 10, 14
   Write a member function that traverses the tree on breadth-first basis
   Hint: you should use an auxiliary queue.

2. Write a member function **Iterative_post_order** that traverses the tree in post-order manner with help of a **stack** instead of recursion.

3. You have studied an algorithm that converts a postfix expression to an expression tree with help of a stack. Write a ***global*** function that takes a string containing a valid postfix expression and constructs the corresponding expression tree and returns it.