Data Structures and Algorithms
LAB #6 – Binary Search Trees

Fall 2018

Objectives

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

- Write code for linked implementation of binary trees.
- Implement different binary trees operations
- Write function for binary trees applications.

Binary Search tree

Binary search tree (BST) is a data structure used for both searching and sorting. Any node contains at most two children and for each node all the values in the left sub-tree are smaller than the root while all the values in the left sub-tree are larger than the root.

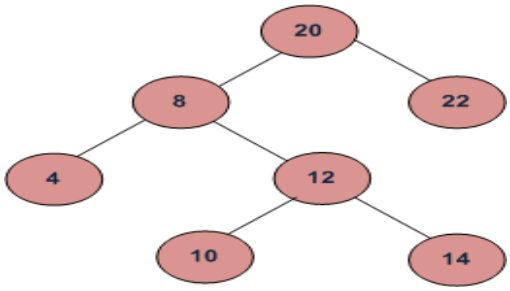


Figure 1 - A binary search tree of integer values

Binary Node Class

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

```
class TreeNode
{
      int data;
      TreeNode * left;    //left child
      TreeNode * right;   //Right child
      public:
        // setters and getters
};
```

Code 1 : TreeNode class

But this definition works for binary tress that stores integer values only.
To generalize, we will redefine the above node as class template

```
template <typename T>
class TreeNode
{
      T data;
      TreeNode<T> * left; //left child
      TreeNode<T> * right;//Right child
      public:
        // setters and getters
};
```

Code 2 : TreeNode class template

Binary Tree ADT Class

The binary tree can be defined as follows:

```
template <typename T>
class BSTree
{     TreeNode<T> * root;
        // some utility functions
      public:
        // some public functions
};
```

Code 3 : Binary Tree class template

The tree is a **recursive** data structure by nature which means any node in the tree can be treated as another tree so most functions that deal with trees are written in a recursive manner.

Each **public function** in BST will need a **utility (private) recursive function** that takes a passed sub-root as a parameter. The utility recursive function will operate on the tree that starts from the sub-root passed to the function.
See Examples➔BSTree.sln➔BSTree

Read the functions carefully and in each function notice the following:

- In each public function, **is it a constant function or not? Why?**
- In each utility function, **does it need the "subRoot" to be passed by value or by reference? Why?**

BST insertion

Read the code of `insertBST`public function and `rec_insertBST` recursive function.

- What will the tree look like if the input data inserted is sorted? How can we solve this problem?
- What is the complexity for this insertion algorithm? And What is the searching complexity for BST?

Tree Traversal

Traversing a tree involves iterating (looping) over all nodes in some manner. Three common traversal algorithms are known for the trees. **Preorder, Inorder, and Postorder** tree traversal

Read the code of `inorder_traverse`, `preorder_traverse` and `postorder_traverse` public function and their utility recursive function.

Destroying the Tree

Trees are made of pointers so you should free the memory used by the tree safely before exiting your program. Read **destroy_tree** function and its recursive utility function.

Note: we called destroy_tree in the BSTree destructor.

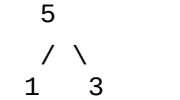
Searching

Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node).

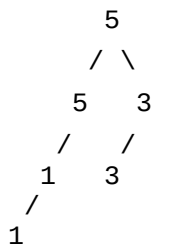
Practice Exercises

Add the following member functions to BSTree class.
For each function you may need to write to functions: (a utility non-public function and a public function)

1. A function that counts the number of nodes in the tree.
2. A function that returns the sum of all the nodes in the tree.
3. A function that counts the number of leaves in the tree.
4. A function that counts the number of leaves that contain even values in the tree.
5. A function ***Search*** that returns a pointer to the node with a given value, *V*.
6. A function that returns the height of the tree.
7. A function that returns the maximum value of all the nodes in the tree.
8. A function that returns the minimum value of all the nodes in the tree.
9. A function that prints all the keys that are less than a given value, *V,* in the tree.
10. A function **has_path_sum** that returns true if the tree contains any path from the root to one of the leaves with the given sum. Try to achieve this without calling function sum from part 2 above.
11. A function ***autumn*** that deletes all the leaves from the tree, leaving the root and intermediate nodes in place. (**Hint**: Use a preorder traversal.)
12. A function **same_structure** that returns true if a passed tree has the same structure as the current tree regardless to the data inside them.
13. A function **is_mirror** that returns true if a passed tree is the mirror for the current tree; every left child in the first tree is a right second tree and vice versa.
14. A function that creates a mirror image of the tree. All left children become right children and vice versa. The input tree must remain the same.
15. A function **double_tree** that, for each node in the tree, it creates a new duplicate node and insert it as the left child.
    So the tree...



Is changed to...



16. A **balanced binary tree** is a tree in which all the following conditions are satisfied:
    a. the left and right sub-trees heights differ by at most one level
    b. the left sub-tree is balanced
    c. the right sub-tree is balanced

    Write a function that checks if a given tree is balanced or not.

17. A **full** binary tree is a tree in which every node other than the leaves has two children.
    Write a function that checks whether a given tree is full or not.

18. Rewrite the binary tree **pre-order, in-order and post-order** traversal member functions using a **stack** instead of recursion.