Cairo University
Faculty of Engineering
Credit Hour System

SBEN429 Biomedical Data Analytics
Assignment 5

Name: **Sandra Adel Aziz Gebraiel**

ID: **1180059**

Date: **29/12/2021**

Assignment 5 Data Structures
Binary Trees and Binary Search Trees

Question (1):

1) **Preorder Traversal:** 1, 2, 4, 5, 3
2) **Inorder Traversal:** 4, 2, 5, 1, 3
3) **Postorder Traversal:** 4, 5, 2, 3, 1

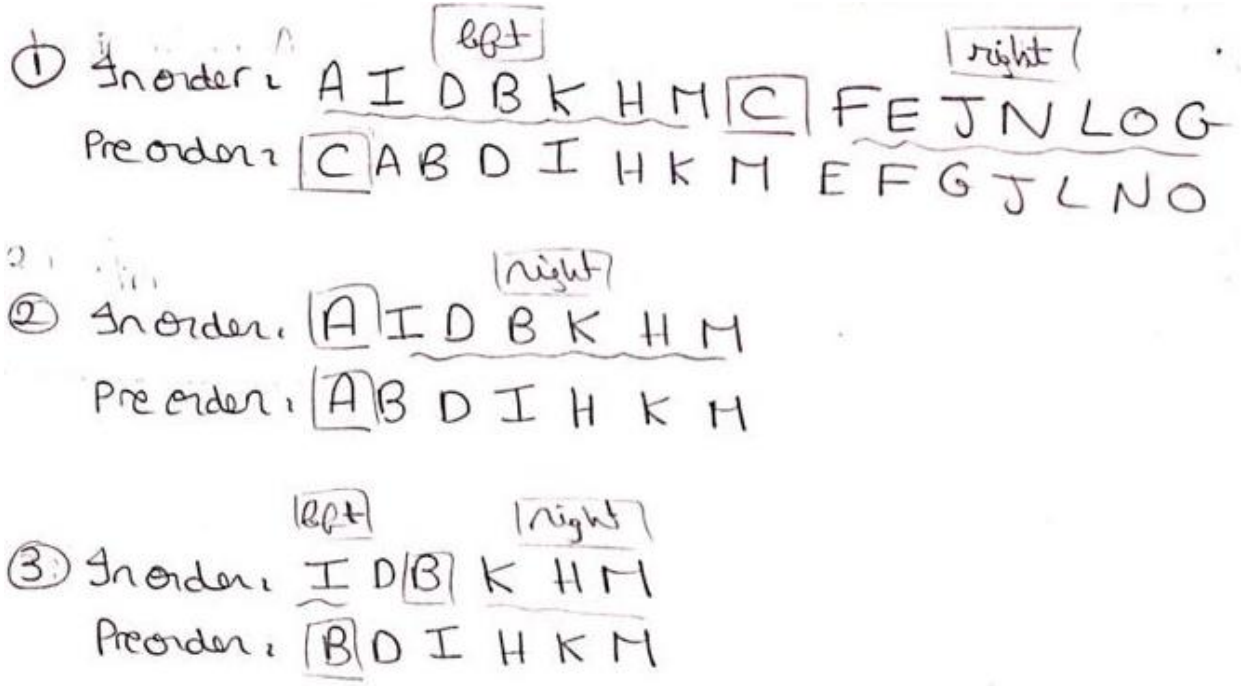
Question (2):

1) Steps:

Since the preorder traversal begins with the root, the first element in the traversal is always the root. With this piece of information, we can locate this element in the inorder traversal, then, all the elements on its left will be in its left subtree and all the elements on its right will be in its right subtree.

We can recursively perform the previous steps until we build the binary tree and find its postorder traversal.

2) Solution:



left

④ Inorder: I D
Preorder: D I

left right

⑤ Inorder: K H M
Preorder: H K M

left right

⑥ Inorder: E E J N L O G
preorder: E F G J L N O

left

⑦ Inorder: J N L O G
Preorder: G J L N O

right

⑧ Inorder: J N L O

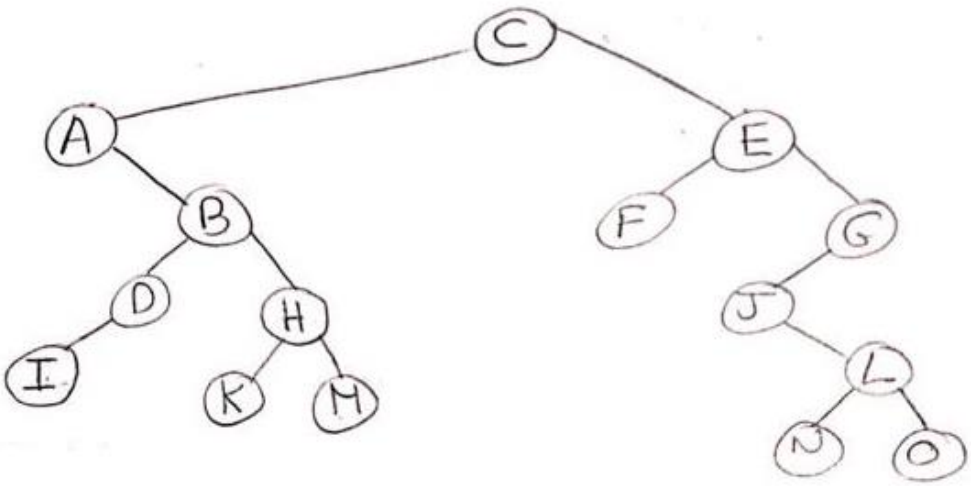
preorder: J L N O

left right

⑧ Inorder: N L O
Preorder: L N O

Tree: Postorder Traversal:

I D K M H B A F N O L J G E C



Questions (3) and (4):

1) Explanation of Code:

A) Classes Definition:

To build the binary search tree data structure, two classes are defined: BinaryTreeNode, specifying the properties of a binary tree, and BinarySearchTree, specifying the properties of a binary search tree in particular. An object of class BinaryTreeNode is a node consisting of a key, a left child and right child (simulating pointers since Python does not have pointers) and has basic OOP methods allowed to act on it outside the class, namely setters and getters for node elements.

As for the BinarySearchTree class, an object of it is the tree itself, consisting of a root (again simulating pointers) which is initially null and is assigned to an object of class BinaryTreeNode which by turn is connected to other objects/nodes of the class as the tree is filled up, and the root has a getter and setter as basic OOP methods. To be able to find the minimum key of a BST, an insertion method must be defined. To be able to insert a new node, its appropriate place (or parent in specific) must be found.

B) FindParentOfNewNode Method:

A method FindParentOfNewNode is defined for this cause. This function takes the value of the key to be inserted to the new node, and current_node which is the root of the tree or subtree to find parent of new node in. The function compares the new key to the key of the current node, if it is less, it further checks if this node has a left child. If it does not, then the new node can be its left child, then this node is returned as an expected parent of the new node. Otherwise, if it does have a child, then we further check down the tree by recursively calling the function but with the left child of the current node as the new current node of the next recursive step.

The same happens if the new key is bigger than the key of the current node. If it does not have a right child, then the current node is returned as the expected parent of the new node. Otherwise, if it does, the function is recursively called with the right child of the current node as the new current node of the next recursive step, in order to further look down the tree for an appropriate parent of the new node, maintaining the properties of the BST.

COMPLEXITY: O(h) where h is the height of the tree and ranges between O(log(n)), if the tree is balanced, and O(n), if the tree is extremely unbalanced.

C) InsertOneNode Method:

As for the insertion process itself, a method InsertOneNode is defined which takes the new key to be inserted. Firstly, an object of class BinaryTreeNode is instantiated as the new node. Then, the tree is checked if it is empty, if it is, this node is inserted as the root of the BST, otherwise, the method FindParentOfNewNode is called with the key and tree root as parameters to find the appropriate parent of the new node, then the new key is checked against the key of the parent, to find out whether to put the new node as its left or right child.

COMPLEXITY: Same as complexity of FindParentOfNewNode method as the complexity of the rest of operations in the method is O(constant)

D) InsertManyNodes Method:

To extend the generalization of the method to insert many nodes at once, not one by one, a method InsertManyNodes is defined, which takes a list of new keys to be added, and iteratively calls the InsertOneNode method on each key.

COMPLEXITY: O(n x h), where n reperesnts the for loop of the number of new keys to be inserted, and in each iteration, the method InsertOneNode is called which is of complexity O(h), where O(log(n)) <= O(h) <= O(n)

E) FindMinimumIteratively and FindMinimumRecursively Methods:

Finding minimum key in BST is implemented iteratively and recursively in FindMinimumIteratively and FindMinimumRecursively respectively. The minimum key in general in a BST is key of the leftmost node of the tree. As long as the tree is not empty, the FindMinimumIteratively method sets the current_node, which is the node to begin checking if it is the leftmost node from, as the root of the tree, and continues assigning the current node with its left child as long as it has one, until it reaches a node without a left child. Therefore, this is the leftmost node, and its key is returned as the minimum of the BST. FindMinimumRecursively takes current node as a parameter and keeps recursively calling the function, assigning it with its left child as long as it has one, until, a left-childless node is reached, and its key is returned as the minimum of the BST as it is the leftmost node.

COMPLEXITY of both: O(h), where O(log(n)) <= O(h) <= O(n). Although both are of same complexity, each method has its pros and cons. Recursion is written in just a few lines of code but has a high space complexity due to stack consumption, while iteration is sometimes less understandable and is written in relatively more lines of code but has less space complexity than recursion.

2) Input and Output:

A) Formats:

Input: One line of BST keys to be inserted, space separated, ending with CTRL+D

Output: First Line: The minimum in BST using iteration

Second Line: The minimum in BST using recursion

B) Samples:

Sample 1:

Input:

9 5 12 13 7 6 14

Ouput:

5

5

Sample 2:

Input:

12 10 15 8 5 7 3

Output:

3

3

3) Pseudocode:

FindParentOfNewNode (key, CurrentNode):

If key of current node >= key:

    If current node has a left child:
        Return FindParentOfNewNode(key, left child of CurrentNode)

    Otherwise:
        Return CurrentNode

If key of current node < key:

    If current node has a right child:
        Return FindParentOfNewNode(key, right child of CurrentNode)

    Otherwise:
        Return CurrentNode

InsertOneNode (key):

NewNode ← Instantiate new object of class BinaryTreeNode with key

If tree is empty:
    Assign NewNode to tree root

Otherwise:
    Parent ← FindParentOfNewNode(key, tree root)

If key <= key of Parent
    Assign NewNode as left child of Parent

Otherwise:
    Assign NewNode as right child of Parent

InsertManyNode (new_keys_list):

For i = 1 → length of new_keys_list:
    InsertOneNode(new_keys_list[i])

FindMinimumIteratively ( ):

If tree is empty:
    Return None

CurrentNode ← tree root

While CurrentNode has a left child:
    CurrentNode ← left child of CurrentNode

Return key of CurrentNode

FindMinimumRecursively (CurrentNode):

If tree is empty:
    Return None

If CurrentNode does not have a left child:
    Return key of CurrentNode

Otherwise:
    Return FindMinimumRecursively (left child of CurrentNode)

4) Stress Testing Code:

```
if __name__ == '__main__':

    '''
    input = sys.stdin.read()
    data = list(map(int, input.split()))

    tree = BinarySearchTree()
    tree.InsertManyNodes(data)

    print(tree.FindMinimumIteratively())
    print(tree.FindMinimumRecursively(tree.GetRoot()))
    '''

    while 1:
            data = []
            n = r.randint(0, 100000)
            print(n)
            for i in range(n):
                data.append(r.randint(0, 100000))
            print(data)

            tree = BinarySearchTree()
            tree.InsertManyNodes(data)

            if len(data) == 0:
                res1 = None
            else:
                res1 = min(data)
            #res2 = tree.FindMinimumRecursively(tree.GetRoot())
            res2 = tree.FindMinimumIteratively()

            if res1 != res2:
                print('wrong answer'+ ' ' + str(res1) + ' ' + str(res2))
                break
            else:
                print('OK')
```

Question (5):

Explanation:

The basic operations performed on BST, whether search, insertion or deletion, requires traversing on the tree until we reach node to be found, appropriate place for node to be inserted or node to be deleted [and finding node whose key is to be replaced with that of node to be deleted (largest in left sub-tree or smallest in right sub-tree) in case of deleting node with 2 children].

Nevertheless of what we will do when we reach node to be found or how to insert the new node, or how to carry out the deletion operation itself, as the complexity of those steps in each of those algorithms is O(constant) [Redirection of pointers, deleting nodes themselves, swapping keys] but the whole weight of their complexity lies in the process of reaching the appropriate node itself within the tree to perform the operation on it.

In worst cases, we go through the number of edges on the longest path from root to node, which is the height of the tree, but we can always end up going through a shorter path. So, complexity of any basic operation on BST is O(h).

However, the balance of the tree has a significant effect on the height, and the balance of the tree is by turn affected by the insertion and deletion operations, not as in algorithm, but as in the order of keys/nodes to be inserted or deleted.

For ex: if we insert in a BST the values of a sorted array, we will end up with an extremely unbalanced tree, with one node at each level, skewed to the right, and its height will actually be the total number of nodes in the tree (n).

But in case of a balanced tree (all levels are full maybe except the last one). When traversing through this tree, we ask ourselves one question at each node: "Is the key to be found smaller than key of this node or bigger?" and going left or right, dividing our search space into 2, until we possibly, in the worst case, reach the deepest leaf in the tree at depth equals to the height of the tree, which will be log(n) in this case.

Since the shape of the tree is unknown when we perform the basic operations in it, the height is estimated to be between n in the worst case (unbalanced) and log(n) (balanced)

SO, Complexity of each of these operations will range between:

$$O(n) <= O(h) <= O(\log(n))$$

Resources:

MOOC: Data Structures Slides - Weeks 1 and 5

Tutorial 9 Slides

https://www.geeksforgeeks.org/construct-a-binary-tree-from-postorder-and-inorder/