CISC-235 Data Structures
Assignment 2

February 5, 2019

1 Problems

In this assignment, you will implement a binary search tree structure and associated functions. The goal of this assignment is to:

1. get familiar with the implementation of a binary (search) tree.
2. use stacks (you need to implement the stack ADT as well).
3. get familiar with iterative/recursive operations on a binary (search) tree.

1.1 Implementation of Stack Using Array-based List: 10 points

Create a class named **Stack** using array-based list provided by a particular programming language (e.g., List in Python). Your class should contain an initialization function, as well as the following functions (we just mention the name of each function, you need determine what is the input):

- **isEmpty**, this function checks whether the current Stack instance is empty. It should return true if the Stack is empty.
- **push**, this function should takes a data item as input and add it into the Stack.
- **pop**, this function should return the data item on the top of the current Stack and remove it from the Stack.
- **top**, this function should return the data item on the top of the current Stack without modifying the Stack.
- **size**, this function should return the number of data items in the Stack.

1.2 Implementation a Binary Search Tree: 80 points

Binary search tree (BST) is a special type of binary tree which satisfies the binary search property, i.e., the key in each node must be greater than any key stored in the left sub-tree, and less than any key stored in the right sub-tree. See a sample binary search tree below:

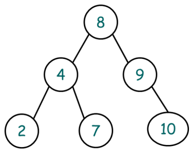

Figure 1: Binary Search Tree 1

Your task is to create a class named **BinarySearchTree** satisfying the following requirements (you can create more instance variables if you want):

1) Each node inside the BST has at least these attributes: value(data stored in the node), leftChild (reference to another node), and rightChild (reference to another node)

2) Must have an **insert** function (i.e., insert a new node). You may assume that the values to be stored in the tree are integers.

3) Must have a **searchPath** function that returns a list of all the values visited on the search path. For example, on a particular tree T, T.searchPath(10) might return 8, 20, 15, 14, 10.

4) Must have a **getTotalDepth** function that computes the sum of the depths of all nodes in the tree. Your getTotalDepth function should run in O(n) time. Hint: if a node knows its own depth, it can tell its children what their depth is. Each node can add its own depth to the total depth reported by its children, and return that sum. The value returned by the root will be the total depth of the tree.

5) Mush have a **getWeightBalanceFactor** function. The weight balance factor of a binary tree is a measure of how well-balanced it is, i.e., how evenly its nodes are distributed between the left and right subtrees of each node. More specifically, weight balance factor of a binary tree is the maximum value of the absolute value of (number of nodes in left subtree-number of nodes in right subtree) for all nodes in the tree. For example, given the binary search tree shown in Figure 2, your function should return 2 although if you just look at the root node, the difference in the number of nodes in the left subtree and right subtree is 1.

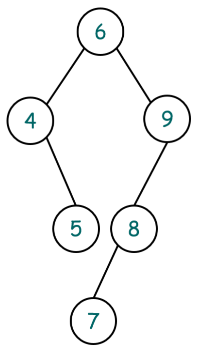

Figure 2: Binary Search Tree 2

1.3 Reconstruct a Binary Search Tree From a File Using a Stack: 30 points

Create a **loadTreeFromFile** function in the above BinarySearchTree class. This function takes a String filename as input, and return a BinarySearchTree object. The file list the nodes of the tree in post-order sequence, one node per line. Each line contains a string which is the data value stored in the node, and two other 0/1 tags. The first tag indicates whether or not this node has a left child, the second tag indicates whether the node has a right child. For example, to reconstruct the sample binary search tree in Figure 1, the input file contains the following lines:
2 0 0
7 0 0
4 1 1
10 0 0
9 0 1
8 1 1

The above information is sufficient to uniquely determine any binary tree. The tree can be reconstructed from the file using the following algorithm:

```
create empty stack of BinarySearchTree
 while there is another line of input in the file:
   read a line (containing data and 2 tags)
   if the second tag is 1, pop an element from the stack and call it
       right_tree
   if the first tag is 1, pop an element from the stack and call it
       left_tree
   crate a new binary search tree with data as its root and left_tree
       and right_tree as its subtrees (if they exist)
   push the new tree onto your stack
```

When you exit the while loop, the stack should contains one element and that is the tree you want to return. Note that you should check the second tag (indicating if the node has right child or not) first. And if your right_tree and left_tree is empty, you should create a new tree with only one node, i.e., the root node. You should use the Stack you created in Section 1.1.

1.4 Test Code and Style: 30 points

You must comment your code, and write test code. In your test code, you should:

1) Reconstruct a binary search tree T (should be the one shown in Figure 1) by reading a file containing the lines shown in Section 1.3.
2) Calculate the total depth of T.
3) Calculate the Weight Balance Factor of T.
4) Insert a new value, i.e., 5, into T.
5) Print path of searching the value 5.
6) Calculate the total depth of T.
7) Calculate the Weight Balance Factor of T.

2 Assignment Requirements

This assignment is to be completed individually (we have the right to run any clone detection tool). You need submit your documented source code (Stack class + BinarySearchTree class + test code). If you have multiple files, please combine all of them into a .zip archive and submit it to OnQ system. The .zip archive should contain your student number. Deadline: Feb-18 12:00am, 2019.