CISC-235 Data Structures W22
Assignment 2

February 5, 2022

How to Submit

You need to submit your python code for all questions. You must comment your code. Name your folder as **your netid-A2**, zip the folder and upload it to OnQ. Check whether your code is runable before submitting.

1 Binary Search Tree

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:

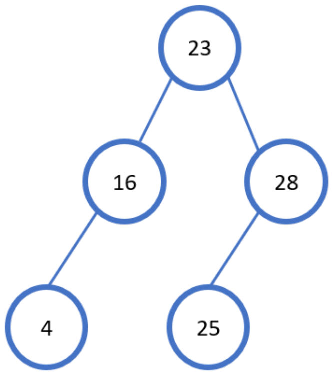

Figure 1: Binary Search Tree 1

Your task is to create **a class named BinarySearchTree** satisfying the following requirements (you can create more methods/attributes if needed):

1) 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.

2) Must have a **get_total_height** function that computes the sum of the heights of all nodes in the tree. Your get_total_height function should run in O(n) time.

3) Must have a **get_weight_balance_factor** 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. Note that, weight balance factor is different from the balance factor introduced in lecture.

   More specifically, weight balance factor of a binary tree is the maximum value of the absolute value of difference between the number of nodes in left subtree and 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. Why? We need to calculate the difference between the number of nodes in left and right subtree for all nodes in the tree. For root node 6, this value is 1 (3 nodes in right subtree - 2 nodes in left subtree). Similarly, for node 4, 9, 5, 8, 7, we have the difference at each node being 1, 2, 0, 1, 0. Thus the return value, is 2, which is the max of 1, 1, 2, 0, 1, 0 .

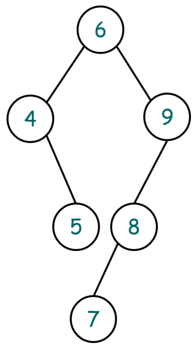

Figure 2: Binary Search Tree 2

4) Write a **serialization** and a **deserialization** function for your Binary-SearhTree function. The two functions should help with transferring your BST into a sequence of values and then taken a sequence of node information as input and reconstruct the tree.

5) Write test code for all requested functions. Hint, you should better write a function that can print/visualize the tree structure when testing.

2 AVLTreeMap

Implement a new data structure named **AVLTreeMap**. AVLTreeMap is just an AVL tree in disguise, where each node in the tree contains a "key-value" pair. We know that AVL tree is a special type of search tree. Similarily, in AVLTreeMap, the position of each node is determined based on the key. The key can be an integer or a string, and the value paired with each key can be a string or a list.

Your AVLTreeMap class should satisfy the following requirements (you can create more instance variables/functions if you want):

1) Each node has left and right child, key, value, and height of the node for adjusting tree structure.

2) Must have a **get(key)** function that returns the value if the given key exists in the AVL tree. Return null if the key isn't in the tree, or the associated value if it is.

3) Must have a **put(key, value)** function (i.e., insert a new key-value pair into the tree). Your put function should adjust the structure in order to maintain a valid AVL trees, i.e., perform rotations if needed and update node height if needed. You can assume that we do not have key-value pairs with the same key.

4) Create an AVLTreeMap by inserting the following key-value pairs in order: 15-bob, 20-anna, 24-tom, 10-david, 13-david, 7-ben, 30-karen, 36-erin, 25-david. Use the above AVLTreeMap test your get and put function.