Data Structures, Lab
CRN: 19897

Salvador Robles Herrera
ID: 80683116

Lab #4 Report

Problem 1

This problem consists on implementing a self-balancing binary search tree.

Problem 2

For this problem it is needed to return the sum of the values of all nodes that are within a certain range. The given root is from a Binary Search Tree and the range is between two values min_val and max_val (inclusive). The return value therefore is an integer.

In the original method called I have a condition that checks whether the given root is null or not, if it is null just return 0 since there are no nodes satisfying the range, else the method will call a helper method. The first thing in the method is to check if the given root is null or not (if it is a leaf), if it is return 0. The second thing I did was to check if the value of the node, passed as parameter in the helper method, is in the range or not. If the value is in the range, it is added to a previously declared variable.

Note that to not make unnecessary recursive calls and since the tree is BST, recursively call the left node or the right only if needed. It will only recursively call the helper method with the original node's left child, if the value of the current node is bigger than or equals to the minimum value in the range, this due that all values to the left of that node would be less than the required range. With the same process to not make unnecessary calls to the right of the BST, the function checks if the value of the node is less than or equal to the maximum value in the range, this due that all values to right of the node would be more than the specified range.

The running time of this code would be O(n) since the range could consist of all the values in the tree, it depends on the number of elements in the range.

The test cases I used helped me to test code's functionality of the code, and to test it when there is one element in the range and when there are no elements that satisfy the specific range.

```
bst = AVLTree()
bst.insert(8)
bst.insert(18)
bst.insert(5)
bst.insert(15)
bst.insert(17)
bst.insert(40)
bst.insert(80)

#Test case #1: Simple case to just test funcitonality of the method
print("What I got: ", range_sum(bst.root, 16, 90), "Expected: 155")

#Test case #2: Just one element in the range
print("What I got: ", range_sum(bst.root, 7, 8), "Expected: 8")
```

Salvador Robles Herrera
ID: 80683116

```
#Test case #2: Where there are no elements that satisfy the range
print("What I got: ", range_sum(bst.root, 6, 7), "Expected: 0")
```

Problem 3

This problem asks to check if the binary tree is “univalued”, meaning that every node in the tree has the same value. The method receives the root of the binary tree and the return value is Boolean.

As always, my approach was to firstly check if the root tree is null or not, if its null I just returned False. If not, then it will call a recursive helper method with the same root as the node passed as a parameter and a value which will be the value of the root of said tree (this to remember the value in the root and that will have to be the same for the other nodes so that the return value is True).

For the recursive helper method, at the beginning if the node as parameter is null or not, if it is return True (it would be a leaf of the tree). There is a condition after that checks if the current node is not equal to the value passed as argument, if not that would mean that the tree is not “univalued”. Then it will be needed to check recursively to the left and right child of the current node, if both calls return True then the whole tree is “univalued”, otherwise at some point of the tree there is a node which is different from the value in the root.

The running time of this code would be O(n) since the method traverses through all the nodes of the tree if the tree is “univalued”.

The cases I used was when the tree has all nodes containing the same value and when the tree has at least two different nodes with different values. The following tests acted as expected:

```
bst2 = AVLTree()
bst2.insert(8)
bst2.insert(8)
bst2.insert(8)
bst2.insert(8)

#Test case #1: Simple case to just test funcitonality of the method when t
he expected value is True
print("What I got: ", is_univalued(bst2.root), " Expected: True")

bst2.insert(9)

#Test case #2: Test when there is at least one element different from the
rest
print("What I got: ", is_univalued(bst2.root), " Expected: False")
```

Salvador Robles Herrera
ID: 80683116

Problem 4

This problem receives a root of a binary tree, it requires the method to return a list of elements, where each element has the average value of nodes at each level of the tree.

As always, the first thing to do is to check if the given root is null or not, if it is just return an empty list. If not, I created a recursive helper method that receives four arguments: the current node, a list containing the number of elements at every level, a list containing the sum of the elements in every level, a list containing all levels (0 to h where h is height) and the current level of the node. This recursive method doesn't return any values, it just updates the passed arguments in the original method, so the method has a condition that if the node is null it doesn't update the passed arguments.

The idea is to traverse through all the nodes in the tree and add the value, contained by the current node, to its respective level of the respective argument/list. Then, it adds one to the number of elements in that level (which is a list passed as an argument) and finalizes with two recursive calls updating the current node with its left and right child. Note that there must be a condition at the beginning that checks if the level of the current node is already in the list that contains of the levels, if not it means that that node is the first node to be traversed by the method so it needs to all the lists as parameters must be updated to contain a space for that level.

The running time of this method will be O(n) since it is necessary to traverse through all the nodes in the tree and note that there is a constant number of operations of constant time.

I tested my algorithm with two different cases and trees, I didn't think of any special cases so I just tried the following tests that acted as expected:

```
bst3 = AVLTree()
bst3.insert(8)
bst3.insert(7)
bst3.insert(9)
bst3.insert(8)

#TEST case 1
print("What I got: ", average_of_levels(bst3.root), "Expected: [8.0, 8.0,
8.0]")

bst4 = AVLTree()
bst4.insert(8)
bst4.insert(18)
bst4.insert(5)
bst4.insert(15)
```

Salvador Robles Herrera
ID: 80683116

```
#TEST case 2
print("What I got: ", average_of_levels(bst4.root), "Expected: [8.0, 11.5,
 15.0]")
```