Data Structures – Project #2 – Report #2 – Beyza Nur KAYA – 150120077 – 28.12.2022

In this project we were asked to do an implementation of AVL and Splay trees and compare their comparison and rotation numbers.

To begin, I started with the AVL trees. I created a structure for AVL and named it AVLNode. It has the properties of key, height, frequency, and two AVLNode pointers for left and right child. I typedef the struct pointer as AVLNodePtr as well. In AVL trees, we have a balance property which is equal to a node's left subtree's height minus that node's right subtree's height. To implement this as code, I check whether the node I've given has the left and right subtree. If one of them does not exist then I return the non-existent's height as -1 and according to that, if the node does not have a right subtree, then the balance property is equal to left – (-1) which is left + 1. if the node does not have a left subtree, then the balance property is equal to (-1) - right. If both exist, then their difference is returned. Then I prepared the base rotations for the AVL Trees. They are Left-Left rotation and Right-Right rotation. I can find the Left-Right and Right-Left (double) rotations using the base Left-Left and Right-Right cases. To give a more detailed explanation, I will just talk about the Left-Left rotation method. This method returns an AVLNodePtr, it increases the rotation number by one and creates a rootLeft pointer with tree's left side. After the rotation we need to put the root's left child's right child into a different location, so I make a pointer for that too. After creating and assigning the pointers, now it is time to do rotation. rootLeft will have a right child as tree, and tree's left child now will be the pointer we created as root's left child's right child. Then the new root will be the rootLeft and we will be returning that as the pointer and the root for the upcoming. Also, for the RR case, it is pretty like the LL case but this time it will be between root's right child and root's right child's left child. After creating the base cases and checks for the AVL insertion, now we can write the insert for the AVL trees.

Firstly, we need to allocate a memory when the tree is null and give it its data. Then, we will search for a place to insert it if it is not null. While finding the insertion place, the comparison counter goes up by one if it can go through the if I put. After finding the insertion place which is written as recursive like a simple BST, I change its height while checking if it does have a left subtree or right subtree or both – since if it does not have one of them then we will have a segmentation fault or more commonly null pointer errors, and later check if it is balanced according to the balance property and the method I've talked about. There are 4 cases of violation of the balance property and now we will talk about them:

LL: For this case, the if conditions are balance value being greater than 1, (left has more height), and the key is less than the tree's left key. If conditions are met, then we will do a simple LL rotation and return the tree so that it can continue to recursion.

RR: For this case, the if conditions are balance value being less than -1, (right has more height), and the key is greater than the tree's right key. If conditions are met, then we will do a simple RR rotation and return the tree so that it can continue to recursion.

LR: For this case, the if conditions are balance value being greater than 1, (left has more height), and the key is greater than the tree's left key. If conditions are met, then we will do a LR rotation/double rotation which will be one right-right rotation in the tree's left and one left-left rotation in the whole tree. After that it will return the new tree so that it can continue to recursion.

RL: For this case, the if conditions are balance value being less than -1, (right has more height), and the key is less than the tree's right key. If conditions are met, then we will do a RL

rotation/double rotation which will be one left-left rotation in the tree's right and one right-right rotation in the whole tree. After that it will return the new tree so that it can continue to recursion.

If the key is found in the tree, we won't do these rotations instead we will simply increase comparison and frequency and return the tree.

That means, the AVL trees are done and now it is time to talk about the AVL trees.

Like AVL trees, in the splay trees I've also created a structure. That structure has key, frequency and two pointers, right and left child as its properties. Then I typedef both the structure and its pointer as SplayNode and SplayNodePtr. In the splay trees, we will first insert the key then we will do the splaying – which is putting the key we got from the text to the top even if we have inserted it before – search in splay trees. After writing the AVL trees, implementing the splay trees wasn't that hard since we use the same rotations. For the AVL trees, I've written an insertion method that is the same as the BST insertions since I don't splay in the insertion method. To explain briefly, it looks whether the tree is null at first. If it is null, then it allocates memory and if it is allocated it will give its properties and return that node and if the memory cannot be allocated it will print a message, then return null. If it is not the case that we have as a base case – which is tree being null, we will search for the place to insert the node and it will recursively go left or right according to the key value and the root's key value. Every time we go through a subtree recursively, the number of comparisons is incremented by 1. If the key I've put exists, the frequency property is updated and will simply return the tree. At the end it will return the tree after all of the conditions.

Now it's time to consider the splay operation which is a bit harder and easier at the same time. In this case the rotations are nearly the same as AVL, but we have some different cases and need to do more rotations to make it a splay tree rotation. So, for these we have a zig and zag case which is simply the same as the RR and LL of the AVL tree. For this reason, I've called them the same not to be confusing. They do the same rotation as the AVL and for the left_left_rotationSplay method, we move a node which is in the left-left to up and it will create a rootLeft which points to the tree's left subtree and create a rootLeftCRightC which points to the rootLeft's right and then make the rootLeft's right assigned as the tree itself and the tree's left child as rootLeftCRightC, and then the new root rootLeft is returned. The reason why we put the rootLeft's right child into a new place is since we are going to replace the rootLeft's right child with the root itself, it will be in void, and we need to put these values, so we won't lose any data. The same goes for the right_right_rotationSplay. Instead of the left and left's right child we will do it with right and right's left child. As I've said before this is the same AVL rotation, and it is enough to create all rotations for splay trees.

In the main splay method, my method takes the tree and key as usual but also, we have a rootkey value, which is the newly inserted tree's root's key value. First, we are going to call the splay method to find where the key is actually, and when it is found, it will do the cases, if the tree is null, it will return null. Now for the cases, we have exactly 6 cases:

X-R: In this case, the node we want to splay is the child of the root.

- If it is in the left, then we need to do a left_left_rotation, it will bring the node we want to splay to the top and will return the tree.
- If it is in the right, then we need to do a right_right_rotation, it will bring the node we want to splay to the top and will return the tree.

Zig-Zig or Zag-Zag: In this case we need to move the node we want to splay up twice.

- If it exists in the left-left, then we will do the left_left_rotation twice since we need X move up twice, after doing the rotation two times, we will return the tree.
- If it exists in the right-right, then we will do the right_right_rotation twice since we need X move up twice, after doing the rotation two times, we will return the tree.

Zig-Zag or Zag-Zig case: In this case we need to move the node we want to splay one left and one right, or one right and one left up and at the end there will be two rotations, same as the double rotation in AVL (RL and LR).

- If the node we want to splay exists in the right-left, first we do a left_left_rotation in the right subtree of the tree and then do a right_right_rotation in the whole tree. After doing the rotation twice, we will return the tree.
- If the node we want to splay exists in the left-right, first we do a rigth_right_rotation in the left subtree of the tree and then do a left_left_rotation in the whole tree. After doing the rotation twice, we will return the tree.

After all these cases, we return the tree again to ensure we return it since it is recursive.

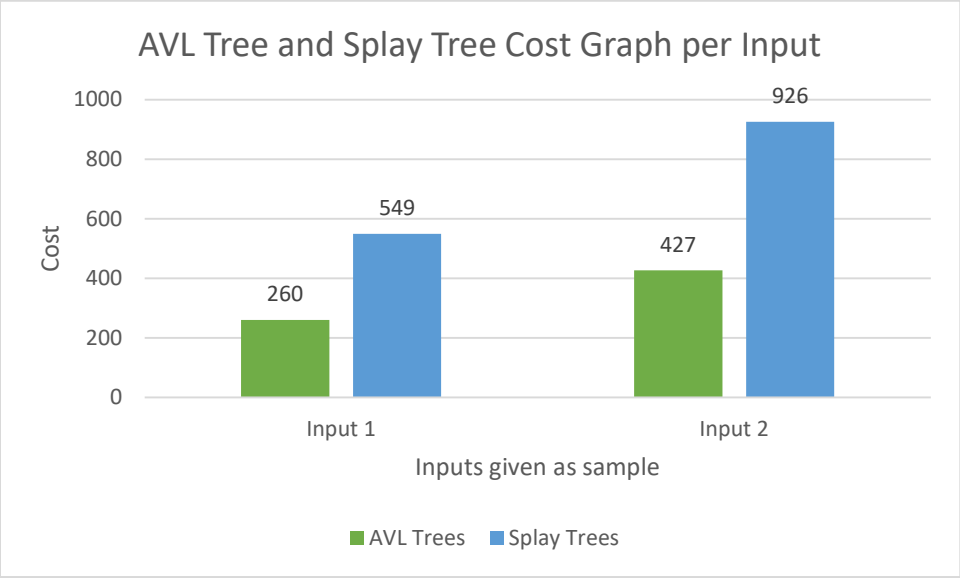
And at the end, I've 2 print pre-order methods for each tree types and here I take the pointer to pointer to the tree and if the *tree is not null, print the key first and then call the method with its left recursively and when it is done printing the left, it will start to print the right recursively since it is preorder (NLR).

At the bottom, I have the main method. It takes arguments and I initialize a number here to read the keys to zero. Then for the AVL tree, I create pointer as null and open the file for AVL, I did the reading separately for each case. I used fscanf to read from file and since the given file is in the format of number blank number blank, which is "%d ". While it gives true, 1, it will call the insertToAVL method recursively and its return value will be assigned to the treeAVL. After reading is done, it prints the tree preorder and prints the total cost, number of comparison and number of rotations. Then close the file for the AVL.

Later, it will open a file for splaying and create pointer for the splay, treeSplay. treeSplay is assigned as NULL at first. While we can read the file, it will call the insertSplay method and assign its return value to treeSplay. After that it will splay the key to the up and assign its return value to treeSplay. After done reading the file, it will print the tree preorder and print the total cost, number of comparisons and number of rotations. Then close the file for the splay as well.

This was how the implementation was done in my code. Before these implementations, I've tried to it in an iterative way but it seemed to give the cost a bit higher due to its harder to write in non-recursive way and I need to check everything, then I made the implementation with recursion like it was thought in the lectures for the Binary Search Trees and even though it was harder to understand this way, it turned with the same costs and values. In the writing of the code, I've got plenty of segmentation faults and memory leaks, but I managed to overcome these struggles as well.

Now it is time to compare these trees. The outputs for the splay and AVL differ and in the bigger size outputs it nearly doubles.



As it is seen in the table for the input 1, the Splay tree's cost is a bit more than the AVL tree's cost's double. For the input 2, the Splay tree is more higher than the AVL as a ratio compared to input 1's ratio. The reason why this happens is, the AVL trees always gives the $\log_2 n$, the best case and don't do much of a rotation unlike splay trees, it does the rotations when the balance property is violated and do in some parts of the tree not the whole for every time.

For splay trees, we may have the best case $\log_2 n$ or $m * \log_2 n$, since we may have it more than the best case, as it is seen in the graphs, it differs a lot. Since the splay tree's purpose is to bring the frequently used elements to the top so that we can access faster, it may be done but still the AVL tree is superior to the splay trees. The reason for people to use the splay trees is that we may directly access the key we wanted to directly since it is near to the root. Also, while I was writing the BST insertion for the splay, I printed some cost values as well and its cost without rotations was higher than the AVL trees as well.