**PROJECT REPORT**

NAME: Yigit Tuncer

STUDENT ID: 150121073

**GENERAL LAYOUT**

The program reads the input file and puts the values into a linked list. Afterwards an AVL tree is created using the insertAVL function. After this is done the values are reset and the splay tree is created using insertSplay. After these are done the total costs are printed to the terminal.

**MAX**

Returns the bigger integer out of the 2 inputs. If they are equal returns the second one.

**HEIGHT**

Finds the height of the node by comparing its sub-trees using the max function and then adding 1 to the larger one. If the node is nonexistent (NULL) returns 0;

**LEFTLEFTROTATION**

Does the left left rotation as shown in the slides for the class. It then increments the heights of the nodes whose heights have changed and returns the new “root” (The top of the new tree).

**RIGHTRIGHTROTATION**

Does the right right rotation as shown in the slides for the class. It then increments the heights of the nodes whose heights have changed and returns the new “root” (The top of the new tree).

**RIGHTLEFTROTATION**

Does a left left rotation on the right of the node and then does a right right rotation thus completing the double rotation.

**LEFTRIGHTROTATION**

Does a right right rotation on the left of the node and then does a left eft rotation thus completing the double rotation.

**INSERTAVL**

This function works recursively. First of all it finds where we need to insert our node by traversing the AVL tree. If a node has the key we are trying to insert then it returns the node itself and the tree remains the same. Once we insert the node the function starts traversing the recursive call stack and while it goes up, on each step it checks if there is a violation of the AVL tree property by finding the height difference between its 2 sub-trees. It also updates the nodes’ heights. If a violation is found it calls the appropriate function to do a transformation and protect the AVL property. If no problem is found then it just returns the node as is.

**ZIG**

This method does a single left rotation.

**ZAG**

This method does a single right rotation.

**ZIGZIG**

This method does a double left rotation.

**ZAGZAG**

This method does a double right rotation.

**INSERTSPLAY**

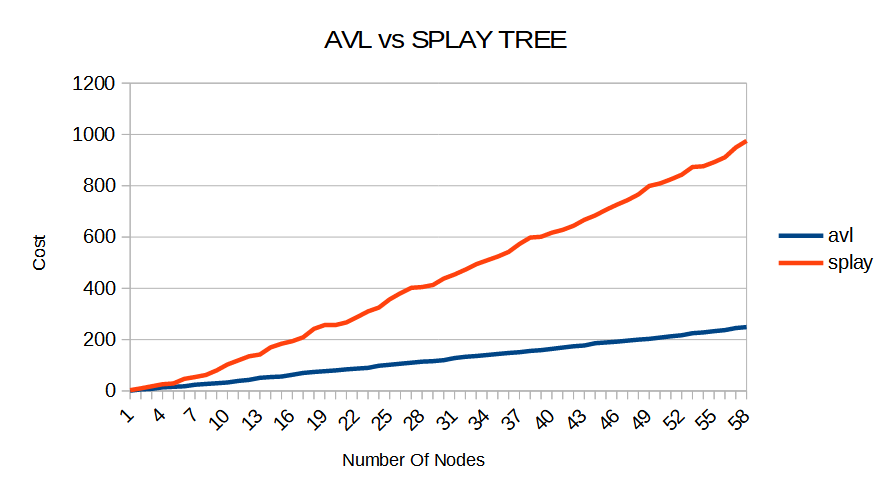
This method starts of exactly the same as the insertAVL method, as they are both binary search trees. After the insertion is complete, similarly it traverses the recursive call stack backwards but this time doesn’t check the heights nor does it update them as they are not used. It checks if the node we are currently on is the root, if so checks if it has the key we inserted. If this is true then the splay operation is complete. If this doesn’t happen then it looks for the key, which it will always find as the rest of the function carries the key up until it cant. If we are not at the root then we search for the key in the sub-trees of the node. We do this by firstly checking whether the key is larger or smaller than our current node then we check for the sub nodes. Once we find the key we do the appropriate transformation by calling the zig zag functions for the outer rotations and we call the double rotation functions used in the AVL tree implementation for the inner rotations as they are completely identical. The function doesn’t do any single rotation until it reaches the root as they are inefficient. If no problem is found then the function returns the node as is. The function stops once the root has the key that we entered.

**ADDNODETOLIST**

This function adds a node to our linked list. It does this by checking wether the list is empty. If so it makes the node the head. If not it loops until the end of the list and adds it there.

**COMPARISON**

In my implementation the AVL tree turned out to be faster for almost all inputs, and the difference grew as the input grew. The splay tree is slower because after each insertion it does a series of transformations while the AVL tree does it sometimes.

****