COP5536 PROGRAMMING ASSIGNMENT REPORT

Name: Prapti Akolkar

UFID: 22287019

UF Mail ID: praptiakolkar@ufl.edu

Directory Structure:

main.cpp: The program consists the implementation of the AVL Tree which has the operations to initialize, insert, delete and search in the AVL tree.

MakeFile: This is used to compile the program with the help of the command make.

To Compile and execute the program:

- Compile the program using make command
- After successful compilation, execute the program using the command ./avltree [INPUT_FILE]
- As per the requirement, the output.txt will be created and will contain the desired results in it of the operations mentioned in the input file.

class AVLNode: for the node structure of the AVL tree

Data variables:

```
int key;
AVLNode *left_Node;
AVLNode *right_Node;
int height;
```

- key variable stores the value of the node
- left_Node is a pointer to the left child of the tree
- right_Node is a pointer to the right child of the tree
- height stores the height of the tree rooted at the current node

Functions:

- **AVLNode* newNode(int key_value):** This function is called when there is a insert of new node into the AVL Tree. The key_value is the value of the node to be created and the left_Node and right_Node point to NULL.
- **AVLNode* rotateRight(AVLNode *rotateNode):** This function is called when we need to perform the right rotation on the rotateNode
- **AVLNode* rotateLeft(AVLNode *rotateNode):** This fuction is called when we need to perform the left rotation on the rotateNode
- **int max(int number1, int number2):** To find the maximum among the two numbers and return it.
- **int getHeightOfTree(AVLNode *temp):** To obtain the height of the temp node and return it.

- **int getBalanceFactor(AVLNode *Node)** returns the balance factor of the Node which is the difference between the heights of the left subtree and right subtree.
- **AVLNode* minValueNode(AVLNode *Node):** This function returns the smallest valued node from the tree.

Insert(key): AVLNode* insert(AVLNode* root, int key_value): This function is to perform the insert operation in the AVL tree.

The following are the three cases of insert:

1. When the tree is empty, the new node will be the root of the tree.
2. When the key_value to be inserted is less than the root value, the key_value is to be inserted in the left subtree of the root and it is stored as the left child of the root.
3. When the key_value to be inserted is greater than the root value, the key_value is to be inserted in the right subtree of the root and it is stored as the right child of the root.
4. When key_value is equal to the current value of the node, then we do not make duplicate insertion.
5. In case the tree becomes unbalanced, we proceed to perform any of the four cases based on the key_value and balance factor:
    a. If balance_factor is less than –1 and key_value>node->right_Node->key then it is the Right Right case and perform rotateLeft(Node)
    b. If balance_factor is less than –1 and key_value<node->right_Node->key then it is the Right Left case and perform rotateRight(Node->right_Node) and then rotateLeft(Node)
    c. If balance_factor is greater than 1 and key_value<node->left_Node->key then it is Left Left case and perform rotateRight(Node)
    d. If balance_factor is greater than 1 and key_value>node->left_Node->key then it is Left Right case and perform rotateRight(Node->right_Node) and then rotateLeft(Node)

Return the root node.

Delete(key): AVLNode* deleteNode(AVLNode *root, int key_value):

1. When the current root points to NULL, there is no possibility of deleting a node and return NULL
2. When key_value<root->key, then the key to be deleted is in the left subtree
3. When the key_value>root->key, then the key to be deleted is in the right subtree
4. If the key_value==root->key, then it is the node that is to be deleted and we will have the following cases:
    a. In case the node to be deleted has no child then just make the root point to NULL
    b. In case the node has a child either left child or a right child store the child in a temp_node, then make root=temp_node i.e., copy the contents of the child in the root and free(temp_node)
    c. In case the node has two children, then we replace the root with the smallest value node in its right subtree which is the temp_node and perform deleteNode(root->right_Node, temp_node->key) that is delete temp_node from its previous place
    d. In case the tree has just one node, delete it and return NULL

5. Then update the height of the root and calculate the balance_factor of the root and check for the imbalance of the tree. There are four such cases of imbalance:
    a. If balance_factor is <–1 and getBalanceFactor(root->right_Node)<=0 then it is the Right Right case and return rotateLeft(root)
    b. If balance_factor <–1 and getBalanceFactor(root->right_Node)>0 then it is the Right Left case and perform root->right_Node=rotateRight(root->right_Node) and then return rotateLeft(root)
    c. If balance_factor > 1 and getBalanceFactor(root->left_Node)>=0 then it is Left Left case and perform rotateRight(root)
    d. If balance_factor>1 and getBalanceFactor(root->left_Node)<0 then it is Left Right case and perform root->left_Node=rotateLeft(root->left_Node) and then return rotateRight(root)
6. Return the root

Search(key):

This is a function long search(AVLNode *root, int targetKey) to perform search of a key in the tree.

1. In case root==NULL then return LONG_MIN
2. Else run a while loop until root==NULL and if targetkey>root->key then root=root->right_Node, if targetkey<root->key then root=root->left_Node and if targetkey==root->key then we found the required key and return root->key

Search(key1,key2):

This is a function void search(AVLNode *root, int key1, int key2, vector<int>& temp to perform the search of keys in the range from key1 to key2 and the result is stored in the temp vector.

1. In case root==NULL then we return NULL
2. Else perform search(root->left_Node, key1,key2,temp) and when key1<=root->key and root->key<=key2 then push_back(root->key) in the temp vector
3. Then perform search(root->right_Node,key1,key2, temp)

Main:

int main(int argc, char **argv):

1. For the file operations, we used Input and output file handlers and to store the output we create output.txt.
2. We obtain the input using ifstream and the filename is in argv[1]. We use the variable stringInput to read the input file.
3. Using substr function to obtain the operation name which are:
    a. Initialize: To create the avl tree by making the root=NULL

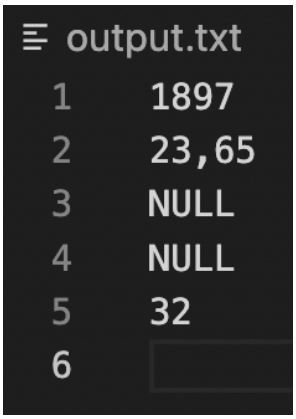
b. Insert: First we get the key to be inserted and convert it to an integer using stoi function and then we perform insert operation
c. Delete: First we get the key to be deleted and convert it to an integer using stoi function and then we perform delete operation
d. Search: We need to search the given key in the AVL Tree. Based on the number of keys given in the input, there are two types of searches:
    i. Search(key): get the key from the input and convert it to an integer using stoi function and perform the search operation and returns LONG_MIN we output NULL else output the root key
    ii. Search the keys in the range of key1 and key2: First, get the two keys from the input to perform search where the result is the temp_node vector which has all the keys that are in the range of key1 and key2 and then we create an ans string where we append all the keys in the temp_node separated by commas.

4. Finally, we close the input and output file handlers.

Sample Input File:

```
Initialize()
Insert(21)
Insert(108)
Insert(5)
Insert(1897)
Insert(4325)
Delete(108)
Search(1897)
Insert(102)
Insert(65)
Delete(102)
Delete(21)
Insert(106)
Insert(23)
Search(23,99)
Insert(32)
Insert(220)
Search(33)
Search(21)
Delete(4325)
Search(32)
```

Sample Output File:



Time Complexities of the operations:

Insert: O(log n)

Delete: O(log n)

Search: O(log n)

S**earch in range:** O(n)