ADS ASSIGNMENT 1

KRISHNA KIREETI RAYAPROLU

6303 1300

rayaprolu.k@ufl.edu

AVL Tree: It is a self-balancing BST where the **absolute** difference between the heights of left and right subtrees cannot be greater than 1.

Since AVL tree is a type of BST, all the properties of BST applies to avl tree as well.

The program implements a simple AVL tree and its functions such as Initialization, Insertion, Deletion, and Search.

Class Node: A node is a basic building block of the avl tree. The class node has the following properties:

Value: Node has a value of type integer.

Left: Node might have a left neighbor.

Right: Node might have a right neighbor.

Height: By default a node has a height of 1.

Initialize Method:

```
public void Initialize(){
        //resets the current avl tree
         root=null;
 }
```

The Initialize method creates a new avl tree with null root. It takes no arguments and returns no values.

Insert Method:

```
public Node Insert(Node node, int key)
```

The prototype for Insert method is a s shown in the image above.

Arguments:

1) root of avl tree

2) key of the node to be inserted

Returns:

Avl tree after insertion

As mentioned before, an avl tree is a type of BST. So the new node will be placed at an appropriate location based on its key value.

If the value of its key is greater than the current node, it will be placed somewhere in the right subtree of the current node, else, it will be placed somewhere in the left subtree of the current node.

Once this is done, the actual properties of an avl tree are to be tested. Meaning, we have to check if the tree is balanced. An avl tree is said to be balanced if the **absolute** difference between the heights of left and right subtrees cannot be greater than 1.

If the avl tree is balanced, we are done with our insertion. Else, appropriate tree rotations are done until the tree gets balanced.

The method returns the root of the tree after insertion is done.

Delete Method:

```
public Node Delete(Node node, int key){
```

The prototype for Delete method is a s shown in the image above.

Arguments:

1)root of avl tree

2) key of the node to be deleted

Returns:

Avl tree after insertion

Since avl tree is a type of BST, initially we search for the node to be deleted. Based on the key value and the current node value, we either search in the left sub tree or the right subtree of the current node. Once the node to be deleted is found, we follow the same rules to delete the node as that of a BST tree.

Case 1: The node to be deleted has no children.

In this case, we are free to delete the node.

Case 2: The node to be deleted has a single child.

In this case, we delete the node and replace it with child.

Case 3: The node to be deleted has two children.

In such case we can replace the current node with either rightmost node of the left subtree or leftmost node of the right subtree.

Once the appropriate node is deleted and replaced, as with the insertion, we need to check if the tree is balanced. Else, perform appropriate rotations.

deleteParentOfTwoChildren Method:

```
public void deleteParentOfTwoChildren(Node node)
```

The prototype of the method can be seen in the picture above. It is a helper function to process deletion of a node with two children.

Arguments:

1) It takes the parent node to be deleted as an argument and performs in place deletion of the node.

findMaximumLeftChildMethod:

```
public Node findMaximumLeftChild(Node node)
```

The prototype of the method can be seen in the picture above. It is a helper function to find the largest leftmost child of the node to be deleted.

Arguments:

It takes the parent node to be deleted as an argument.

Returns:

The rightmost child in the left subtree of the parent node to be deleted.

Helper functions for Balancing Tree:

calculateBalanceFactor Method:

```
int calculateBalanceFactor(Node node)
```

The prototype of the method can be seen in the picture above. It is a helper function to calculate the balance factor at each node.

Balance factor at each node is the difference between the height of its left subtree and right subtree. A node is said to be balanced if its balance factor is in the range [-1,1] inclusive. Else, appropriate rotations need to be performed.

Arguments:

Node at which balance factor needs to be calculated

Returns:

Integer value of the balance factor of the node.

leftRotate Method:

```
public Node leftRotate(Node a)
```

The prototype of the method can be seen in the picture above. It is a helper function to left rotate an imbalanced node.

Arguments:

Takes the node to be rotated as an argument

Returns:

The rotated node.

rightRotate Method:

```
public Node rightRotate(Node b)
```

The prototype of the method can be seen in the picture above. It is a helper function to left rotate an imbalanced node.

Arguments:

Takes the node to be rotated as an argument

Returns:

The rotated node.

Search Method:

```
public Node Search(Node node,int key){
```

The prototype for search method is as shown. It takes the avl tree root, key to be searched as arguments and return the node that is found.

The search algorithm is similar to that of Binary Search tree. If the value of current node is greater than the key, we search in the left sub tree. Else, we search in the right sub tree.

Search Method (Search in a range):

```
public ArrayList<Node> Search(Node node, int key1, int key2)
```

The prototype for search method is as shown. It takes the avl tree node, keys for upper limit and lower limit ranges as inputs and returns a list of nodes in the range.

The algorithm used for searching is in-order traversal. This is chosen because the list of nodes returned by this Search method should be in ascending order.

inorderTraversalMethod:

```
public ArrayList<Node> inorderTraversal(Node node, int key1, int key2, ArrayList<Node> nodeList)
```

The prototype for in-order traversal method is as shown above.

It is a helper function which takes avl tree root, key 1, key 2 and list of nodes as arguments and returns a list of nodes that are found in the range of key1 and key 2.

File Structure:

1) **avltree.java:** Contains the class avltree and main method. The entry point for execution is in this file.
2) **Node.java:** Contains the class Node.
3) **Makefile:** make file to create java class files which can be further executed.
4) **Input.txt:** Contains the input data which the main method reads and makes function calls. The input file name is not hard coded and it can have any name. Preferably it should be a text file.
5) **output_file.txt(optional):** contains output data for test case 1. After each test this file gets overridden with new output. The file name output_file.txt is hardcoded.

Input Format: The input should be given in a text file where each method call should be written in a new line.

Spaces before the actual data in each line are omitted by the program.

Example for input:

Initialize()

Insert(12)

Delete(12)

Search(13)

Search(13,15)

Output Format: The output will be printed on to a text file named output_file.txt. Initialize, Insert and Delete will not print anything in the output file.

Search(int a) prints the a in the new line. If a is not found in the tree then NULL will be printed.

Search(int a, int b) prints all the nodes in the avl tree within the range a, b inclusive. These values will be comma separated.

If nothing is found, NULL will be printed.

For each execution output file will be overridden.

Procedure to run the program:

Step 1: Inside the directory, create an input.txt file of your choice.

Step 2: Run the "make" command to create compile code.

Note: For windows, you might want to use "MinGW32-make" command.

If you want to use java compiler, use the command

javac avltree.java

Step 3: Execute the code with the command

java avltree file_name

Step 4: Check the output_file.txt file that is created for results.

Structure of the program:

```
Class avltree{

Class properties;

//get data from main method

Class Methods()

//return data to main method

}

Main(){

Read input file()

Method calls()

Write to output file()

}
```