

COP 5536 Spring 2022

Advanced Data structures

Project - AVL Tree

Name: Yashaswini Kondakindi

UF ID: 7061-1190

Email ID: y.kondakindi@ufl.edu

Summary of the Project:

In this project, implementation of Initialize, Insert, Delete and Search operations of an AVL tree using Java programming language.

The definition of an AVL tree is a height-balanced binary search tree in which each node has a balance factor that is determined by subtracting the height of the node's right subtree from the height of its left subtree. If each node's balance factor falls between -1 and 1, the tree is considered to be balanced; otherwise, the tree needs to be balanced.

Steps to compile and execute the program:

Step1 : Open the terminal/ command prompt and set up the path to the folder that contains project files.

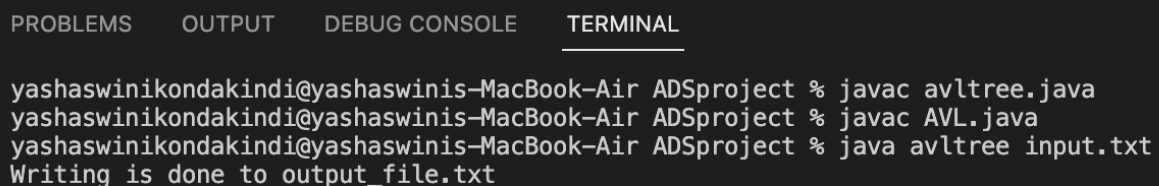
Step 2: Run the command "make avltree" to compile.

Step 3: Run command

"javac avltree "

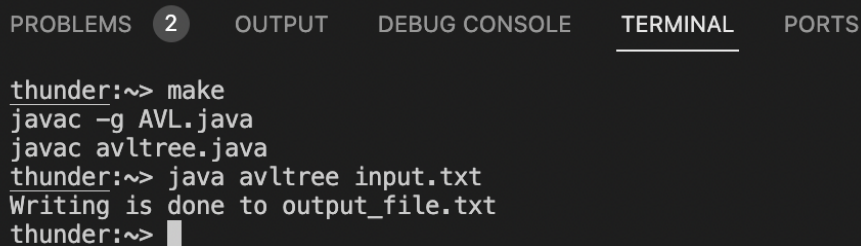
"javac AVL"

"java avltree input.txt"



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
yashaswinikondakindi@yashaswinis-MacBook-Air ADSproject % javac avltree.java
yashaswinikondakindi@yashaswinis-MacBook-Air ADSproject % javac AVL.java
yashaswinikondakindi@yashaswinis-MacBook-Air ADSproject % java avltree input.txt
Writing is done to output_file.txt
```

In thunder,



```
PROBLEMS  2  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
thunder:~> make
javac -g AVL.java
javac avltree.java
thunder:~> java avltree input.txt
Writing is done to output_file.txt
thunder:~> █
```

Step4: Then output will be generated in file “output_file.txt”

Step5: To clean the files execute
“Make clean”

Operations:

Initialize: This method creates a new AVLTree

Insert(key): This method Inserts a new node into the AVLTree with the node value as the given key.

Delete(key): This method deletes node with the given key from the AVLtree

Search(key): This method returns the value of the node if a node with given keys exists.

Search(key1, key2): This method returns the values within the given range
key1 <= key <= key2

Structure of Project File:

avltree.java:

This file contains the main function which is the starting point for our project. This file deals with Input file reading, routing the methods and generating the output text file.

AVL.java:

This file contains the declarations and implementations of functions : Insert, Delete, Search, Rotation of trees, and tree traversals.

Makefile:

This file contains instructions for compilation and execution of our project along with clean.

AVL Tree Node Structure:

The node of the AVL tree contains key, height of the node, left subtree, right subtree.

```
public class Node {  
  
    int value;  
  
    int height;  
  
    Node left;  
  
    Node right;  
  
    // Node class constructor  
  
    public Node(int key) {  
  
        this.value = key;  
  
    }  
  
}  
  
public Node root = null;
```

Function Prototypes:

Insert(key):

```
// this is the Insert method to insert a key into the avltree
```

```
public void insert(int key) {  
    root = insert(root, key);  
}
```

```
// this is the insert method to insert a key in avltree: helper
```

```
private Node insert(Node node, int key) {  
    if (node == null) {  
        return new Node(key);  
    } else if (node.value > key) {  
        node.left = insert(node.left, key);  
    } else if (node.value < key) {  
        node.right = insert(node.right, key);  
    } else {  
        throw new RuntimeException("duplicate Key is generated");  
    }  
    return return_balance_tree_node(node);  
}
```

Delete():

```
// this is the Delete method to delete a key in the avltree
```

```
public void delete(int key) {  
    root = delete(root, key);  
}
```

```
// this is the delete method to delete a key in avltree: helper
```

```
private Node delete(Node node, int key) {  
    if (node == null) {  
        return node;  
    } else if (node.value > key) {  
        node.left = delete(node.left, key);  
    } else if (node.value < key) {  
        node.right = delete(node.right, key);  
    } else {  

```

```

        if (node.left == null || node.right == null) {
            node = (node.left == null) ? node.right : node.left;
        } else {
            Node most_Left_Child = most_left_child(node.right);
            node.value = most_Left_Child.value;
            node.right = delete(node.right, node.value);
        }
    }
    if (node != null) {
        node = return_balance_tree_node(node);
    }
    return node;
}

```

Search(Key1, key2):

```

// this is the Search method to search keys between t1 and t2

    public void searchInGivenRange( Node node, int t1, int t2, List<String>
result_search_avl ){
        if(node == null || (t1>t2)) return;
        int d = node.value;
        if( d >= t1 && d <= t2 ) result_search_avl.add(String.valueOf(d));
        if( t1 < d ) searchInGivenRange( node.left, t1, t2, result_search_avl );
        if( t2 > d ) searchInGivenRange( node.right, t1, t2, result_search_avl );
    }

```

Search(Key):

```

// this is the Search method to search a key in the avltree

    public String search(int key) {
        Node this_node = root;
        while (this_node != null) {
            if (this_node.value == key)
                return Integer.toString(this_node.value);
            this_node = this_node.value < key ? this_node.right :
this_node.left;
        }
        return "NULL";
    }

```

Rotations of the tree:

```
// this rotate_right method to rotate tree right around the given node

private Node rotate_right(Node u) {
    Node a = u.left;
    Node b = a.right;
    a.right = u;
    u.left = b;
    updated_avl_height(u);
    updated_avl_height(a);
    return a;
}

// this rotate_left method to rotate tree left around the given node

private Node rotate_left(Node u) {
    Node a = u.right;
    Node b = a.left;
    a.left = u;
    u.right = b;
    updated_avl_height(u);
    updated_avl_height(a);
    return a;
}
```

Functional Description of the program:

This is the project flow.

AVL.java:

Initialize ():

First line read is Initilaize().

```
// this method is called Initialization to create a new avltree

public static avltree Initialize() {
    avltree init_avl_tree = new avltree();
    return init_avl_tree;
}
```

Insert(key):

This function gets called when Input(key) is the line number in the input file.

If the tree is empty, create a node and return it

Recursively find the appropriate position if the key is less than root.

Using recursion, if the key is greater than root, go to the right subtree and find the right place.

Once the tree is inserted, calculate the height of the inserted node and store it, then calculate the balance factor.

Whenever a node has a balance factor over 1 or -1, we need to rotate it to balance.

·If balance factor > 1, then we rotate the tree towards the left subtree.

- If the key to insert is less than parent, then we do a right rotation.
- Otherwise, we need to do a left rotation followed by a right rotation.

```
/ this is the insert method to insert a key in avltree: helper

private Node insert(Node node, int key) {
    if (node == null) {
        return new Node(key);
    } else if (node.value > key) {
        node.left = insert(node.left, key);
    } else if (node.value < key) {
        node.right = insert(node.right, key);
    } else {
        throw new RuntimeException("duplicate Key is generated");
    }
    return return_balance_tree_node(node);
}
```

We rotate the tree if balance factor - 1, since the tree is imbalanced towards the right subtree.

When the key to be inserted is bigger than the parent, we rotate left.

We'll need to do a Right rotation and a Left rotation otherwise.

```
// return_balance_tree_node method to update balance factor of given node

private Node return_balance_tree_node(Node k) {
    updated_avl_height(k);
    int balance_avl = return_B_Factor(k);
    if (balance_avl > 1) {
        if (height(k.right.right) > height(k.right.left)) {
```

```

        k = rotate_left(k);
    } else {
        k.right = rotate_right(k.right);
        k = rotate_left(k);
    }
} else if (balance_avl < -1) {
    if (height(k.left.left) > height(k.left.right)) {
        k = rotate_right(k);
    } else {
        k.left = rotate_left(k.left);
        k = rotate_right(k);
    }
}
return k;
}

```

Delete(key):

- The L rotation should be applied if the node to be deleted is in the left subtree of the critical node; otherwise, the R rotation should be applied if it's in the right subtree of the critical node.
- We search for the node with a given value, once it's found, we replace it with the left predecessor or right successor.

```

// this is the delete method to delete a key in avltree: helper

private Node delete(Node node, int key) {
    if (node == null) {
        return node;
    } else if (node.value > key) {
        node.left = delete(node.left, key);
    } else if (node.value < key) {
        node.right = delete(node.right, key);
    } else {
        if (node.left == null || node.right == null) {
            node = (node.left == null) ? node.right : node.left;
        } else {
            Node most_Left_Child = most_left_child(node.right);
            node.value = most_Left_Child.value;
            node.right = delete(node.right, node.value);
        }
    }
}

```

```

    }

}

if (node != null) {
    node = return_balance_tree_node(node);
}

return node;
}

```

- Update the node's height and calculate the balance factor by using the height of the left subtree minus the height of the right subtree.
- Once the balance factor is calculated, we'll rotate the tree left and right to balance it.

Search (Key):

This function finds a node for a given key.

This function does in order traversal using recursion to find the node.

```

// this is the Search method to search a key in the avltree

public String search(int key) {
    Node this_node = root;
    while (this_node != null) {
        if (this_node.value == key)
            return Integer.toString(this_node.value);
        this_node = this_node.value < key ? this_node.right :
this_node.left;
    }
    return "NULL";
}

```

- In output_file.txt, we print the found node if it's found, otherwise null.

Search(key1,key2):

Using this function, you can find nodes in a given range ($\text{key1} \leq \text{key} \leq \text{key2}$).

We iterate through the tree using inorder traversal and store all the nodes that are in a given range in an array.

```

// this is the Search method to search keys between t1 and t2

public void searchInGivenRange( Node node, int t1, int t2, List<String>
result_search_avl ){

```



```

        if(node == null || (t1>t2)) return;
        int d = node.value;
        if( d >= t1 && d <= t2 ) result_search_avl.add(String.valueOf(d));
        if( t1 < d ) searchInGivenRange( node.left, t1, t2, result_search_avl );
        if( t2 > d ) searchInGivenRange( node.right, t1, t2, result_search_avl );
    }

```

rotate_left(Node u):

The AVL tree becomes unbalanced when a node is added to or deleted from the left subtree; hence, Left rotation is needed.

Once the rotation is done, we update the node's height and get the new head.

```

// this rotate_left method to rotate tree left around the given node

private Node rotate_left(Node u) {
    Node a = u.right;
    Node b = a.left;
    a.left = u;
    u.right = b;
    updated_avl_height(u);
    updated_avl_height(a);
    return a;
}

```

rotate_right(Node u):

The AVL tree becomes unbalanced when a node is added or deleted from the right subtree, so Right rotation is needed to fix the tree.

Once the rotation is done, we update the height of the node and give it a new head.

```

// this rotate_right method to rotate tree right around the given node

private Node rotate_right(Node u) {
    Node a = u.left;
    Node b = a.right;
    a.right = u;
    u.left = b;
    updated_avl_height(u);
    updated_avl_height(a);
    return a;
}

```

Output file:

The output goes into output_file.txt. Below is the generated output for the given input.

```
≡ output_file.txt
1  1897
2  23,65
3  NULL
4  NULL
5  32
6  
```

In thunder,

```
thunder:~> make
javac -g AVL.java
javac avltree.java
thunder:~> java avltree input.txt
Writing is done to output_file.txt
thunder:~> make clean
rm -f *.class output_file.txt
thunder:~> █
```