



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
PES UNIVERSITY**

(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

February - May 2023

Report on

“AVL Tree Implementation/Visualisation”

**Master of Technology
in
Computer Science & Engineering**

UE22CS642A – Experiential Learning (ISA Component)

Submitted by:

SHALINI T (PES1PG22CS045)

1st Semester M.tech

We are using the following operations for the implementation of AVL tree

Balance and Update Height

```

32
33
34 //Balance function computes the balance factor of the node
35 int Balance(Node key)
36 {
37     if (key == null)
38         return 0;
39
40     else
41         return ( Height(key.right) - Height(key.left) );
42 }
43
44 //updateHeight function updates the height of the node
45 void updateHeight(Node key)
46 {
47     int l = Height(key.left);
48     int r = Height(key.right);
49
50     key.height = Math.max(l , r) + 1;
51 }
52
53

```

Build failed, do you want to continue?

. The balance factor is the difference between the height of the right subtree and the height of the left subtree. The method first checks if the input node is null, indicates an empty subtree, then return 0. otherwise return the balance of the tree using height method defined
Update Height: this method takes input and updates its height

Balance Tree

```

105 Mini > J AVI.java > T AVLTree > @ rotateRight(Node)
106
107 // balanceTree function balances the tree using rotations after an insertion or deletion
108 Node balanceTree(Node root)
109 {
110     updateHeight(root);
111
112     int balance = Balance(root);
113
114     if (balance > 1) //R
115     {
116         if (Balance(root.right) < 0) //RL
117         {
118             root.right = rotateRight(root.right);
119             return rotateLeft(root);
120         }
121         else //RR
122             return rotateLeft(root);
123     }
124
125     if (balance < -1) //L
126     {
127         if (Balance(root.left) > 0) //LR
128         {
129             root.left = rotateLeft(root.left);
130             return rotateRight(root);
131         }
132         else //LL
133             return rotateRight(root);
134     }
135
136     return root;
137 }
138
139 Node root;
140

```

In this function, It first updates the height of the current node and then calculates the balance factor.

It checks If the balance is greater, lesser than 1.

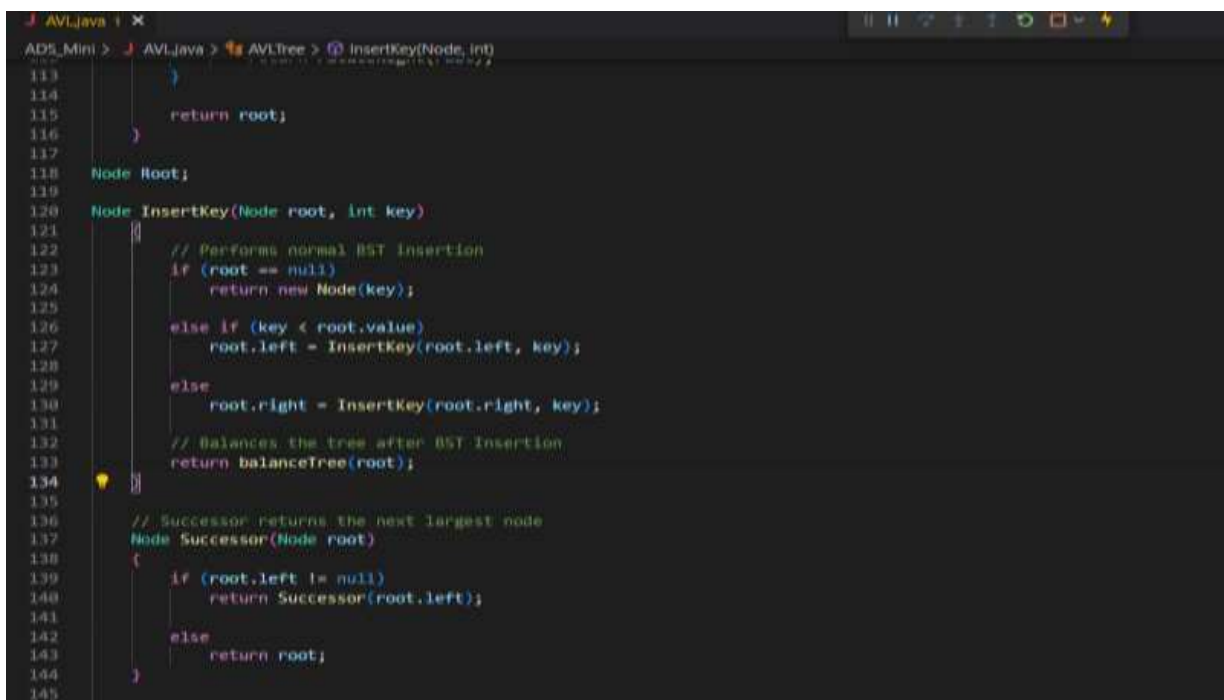
If the balance greater than one the node is considered right heavy balance using right rotation. And the code checks if BF of right child less than 0 – RL rotate, right rotate on right child and left rotate on root to balance

And balance factor of right child is greater than 0, indicates right rotation (RR) the code performs left rotate on root node

If balance less than -1 considered as left heavy. Needs to balance using left rotate. And if less than -1 check the BF of left child of root greater than 0 – LR Rotation. Left rotate on left child of root and right rotate on root. If BF less than 0 – LL rotate, right rotate on the root node

If the balance factor are in range 1, -1, 0 the tree is said to be balanced.

Insert Key



```

113     }
114
115     return root;
116 }
117
118 Node root;
119
120 Node InsertKey(Node root, int key)
121 {
122     // Performs normal BST Insertion
123     if (root == null)
124         return new Node(key);
125
126     else if (key < root.value)
127         root.left = InsertKey(root.left, key);
128
129     else
130         root.right = InsertKey(root.right, key);
131
132     // Balances the tree after BST Insertion
133     return balanceTree(root);
134 }
135
136 // Successor returns the next largest node
137 Node Successor(Node root)
138 {
139     if (root.left != null)
140         return Successor(root.left);
141
142     else
143         return root;
144 }
145
  
```

This is implementation of insert key takes 2 arguments root and key. It compares the value of key to the node. If key less than root.value, code recurses on the left subtree. If key greater than root.value, code recurses on right subtree.

The successor function returns the next largest node. The code first check if the left subtree of root is non-empty if it is the code recurses on left subtree, if its empty returns the current node which is the successor

The time complexity of insert operation is $O(\log n)$

Delete Key

```

147 Node DeleteKey(Node root, int key)
148 {
149     // Performs standard BST deletion
150     if (root == null)
151         return null;
152
153     else if (key < root.value)
154         root.left = DeleteKey(root.left, key);
155
156     else if (key > root.value)
157         root.right = DeleteKey(root.right, key);
158
159     else
160     {
161         if (root.right == null)
162             root = root.left;
163
164         else if (root.left == null)
165             root = root.right;
166
167         else
168         {
169             Node temp = Successor(root.right);
170             root.value = temp.value;
171             root.right = DeleteKey(root.right, root.value);
172         }
173     }
174
175     if (root == null)
176         return null;
177
178     else
179         // Call Node delTree.balancetree(Node root)
180         return balanceTree(root);
181 }

```

The delete key function compares the key values, if key less than root the key recurses on left subtree by calling delete function and If key greater than root.value code recurses on right subtree. If key equal to 0 it checks if either left or right child is null. If both are not null it finds the successor and sets value of root to successor and recursively calls delete function.

The time complexity of delete key operation is $O(\log n)$

Find Key

```

183
184 // FindKey is used to search for a particular value given the root.
185 Node FindKey(Node root, int key)
186 {
187     if (root == null || key == root.value)
188         return root;
189
190     if (key < root.value)
191         return FindKey(root.left, key);
192
193     else
194         return FindKey(root.right, key);
195 }
196
197 // Utility Function for Insertion of num
198 void add(int key)
199 {
200     if (FindKey(root, key) == null)
201     {
202         root = InsertKey(root, key);
203         System.out.println("Insertion successful");
204     }
205
206     else
207         System.out.println("Key with the entered value already exists in the tree");
208 }
209
210 int search(int key)
211 {
212     if (FindKey(root, key) == null)
213         return 0;
214     else
215         return 1;
216 }
217

```

The Findkey function first checks if root is null or key equal to value of root. If either is true it returns root node. If value of key less than value of root code recursively searches in left subtree and if key greater than the root the function recursively searches the right subtree. The function continues to recursively search until desired key is found

The time complexity of Findkey operation is $O(\log n)$

Destroy Tree

```
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
Node destroytree(Node root)
{
    if (root != null)
    {
        destroytree(root.left);
        destroytree(root.right);

        root.left = null;
        root.right = null;
        root = null;
    }

    return root;
}
```

The function starts by checking if root is not null. If root is null it simply returns null. If root is not null function recursively calls destroy tree o left and right subtree. After deleting both subtree the function sets the both left and right pointers of the root node to null and then sets root itself to null. This ensures the entire tree rooted at root has been deleted.

The time complexity of Destroy tree operation is $O(n)$

Learning Outcomes of Project

Learned about the concept of balance tree and rotations used to balance the tree.
Concepts of AVL tree properties

OUTPUT

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/powershell

PS C:\Users\user\Desktop\AB_Minix > .\C:\Program Files\Java\jdk-11\bin\java.exe -cp src\bin\app-transport-rt,socket-server-rt,search-rt,sort-rt\classes\*.jar
C:\Users\user\Desktop\AB_Minix> java -cp src\bin\app-transport-rt,socket-server-rt,search-rt,sort-rt\classes\*.jar
1. Insert
2. Delete
3. Search
4. InOrder traversal
5. PreOrder traversal
6. PostOrder traversal
7. Destroy
8. Exit
9
Enter the elements to add and enter -999 to stop:
12
Insertion successful
13
Insertion successful
23
Insertion successful
34
Insertion successful
45
Insertion successful
56
Insertion successful
67
Insertion successful
78
Insertion successful
89
Insertion successful
21
Insertion successful
32
Insertion successful
43
Insertion successful
Insertion successful
```

```
1. Insert
2. Delete
3. Search
4. InOrder traversal
5. PreOrder traversal
6. PostOrder traversal
7. Destroy
8. Exit
9
Enter the element to be deleted:
89
Deletion successful

InOrder Traversal :
12 13 17 21 23 32 34 35 43 45 54 55 56 67 76 77 78 87 89
PreOrder Traversal :
34 21 13 12 17 23 32 54 45 43 35 54 55 76 78 67 77 89 87

1. Insert
2. Delete
3. Search
4. InOrder traversal
5. PreOrder traversal
6. PostOrder traversal
7. Destroy
8. Exit
9
Enter the element to be searched:
23
Found

InOrder Traversal :
12 13 17 21 23 32 34 35 43 45 54 55 56 67 76 77 78 87 89

1. Insert
2. Delete
3. Search
4. InOrder traversal
5. PreOrder traversal
6. PostOrder traversal
7. Destroy
```

Storage settings

Free up storage space

Your computer's running low on storage space. Visit storage settings to free some up.