

MIT WORLD PEACE UNIVERSITY

Advanced Data Structures
Second Year B. Tech, Semester 4

IMPLEMENTATION OF AVL AS A DATA STRUCTURE

ASSIGNMENT NO. 9

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

April 26, 2023

Contents

1 Objectives	1
2 Problem Statement	1
3 Theory	1
3.1 Different Cases of Rotations in AVL Tree	1
3.2 Construction of AVL Tree as a Data Structure for Creation of Dictionary	2
3.3 Searching and Deleting in AVL Tree	3
3.4 Advantages of AVL Tree over Other Data Structures	3
3.5 Disadvantages of AVL Tree over Other Data Structures	3
4 Platform	4
5 Test Conditions	4
6 Input and Output	4
7 Pseudo Code	4
8 Time Complexity	6
8.1 Creation, Searching, Insertion and Deletion in AVL Trees	6
9 Code	6
9.1 Program	6
10 Conclusion	13
11 FAQ	14

1 Objectives

1. To study the concept of AVL trees
2. To study different rotations applied on AVL tree

2 Problem Statement

A Dictionary stores keywords and its meaning. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword.

3 Theory

An AVL (Adelson-Velskii and Landis) tree is a self-balancing binary search tree in which the heights of the left and right subtrees of any node differ by at most one. The balancing property of AVL trees ensures that the worst-case time complexity of search, insert and delete operations is $O(\log n)$, where n is the number of nodes in the tree.

The height of a node is the number of edges in the longest path from the node to a leaf. An AVL tree is balanced if and only if the heights of its left and right subtrees differ by at most one. If the height difference is more than one, the tree is rebalanced by performing one or more rotations.

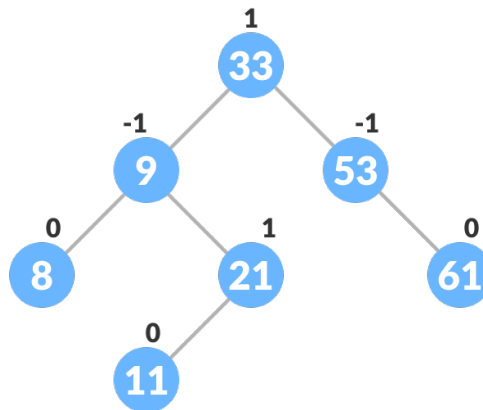


Figure 1: Example of an AVL Tree

3.1 Different Cases of Rotations in AVL Tree

There are four possible cases of rotation in AVL trees:

1. *Left-Left (LL)*: Case: This case occurs when the height of the left subtree of a node is greater than the height of the right subtree by more than one, and the height of the left subtree's left

child is greater than or equal to the height of its right child. To fix this, we perform a right rotation at the unbalanced node.

2. *Right-Right (RR)*: Case: This case occurs when the height of the right subtree of a node is greater than the height of the left subtree by more than one, and the height of the right subtree's right child is greater than or equal to the height of its left child. To fix this, we perform a left rotation at the unbalanced node.
3. *Left-Right (LR)*: Case: This case occurs when the height of the left subtree of a node is greater than the height of the right subtree by more than one, and the height of the left subtree's right child is greater than the height of its left child. To fix this, we perform a left rotation at the left child, followed by a right rotation at the unbalanced node.
4. *Right-Left (RL)*: Case: This case occurs when the height of the right subtree of a node is greater than the height of the left subtree by more than one, and the height of the right subtree's left child is greater than the height of its right child. To fix this, we perform a right rotation at the right child, followed by a left rotation at the unbalanced node.

3.2 Construction of AVL Tree as a Data Structure for Creation of Dictionary

AVL trees are commonly used as data structures for the creation of dictionaries. A dictionary is a collection of key-value pairs, where each key is associated with a value. In an AVL tree-based dictionary, the keys are stored in the tree, and the associated values are stored in the nodes.

To construct an AVL tree-based dictionary, we start with an empty AVL tree. For each key-value pair to be inserted, we perform a binary search in the tree to find the correct position for insertion. If the key is already present in the tree, we update its value. If the key is not present, we create a new node with the key-value pair and insert it into the tree. After insertion, we check if the tree is still balanced. If it is not, we perform one or more rotations to balance it.

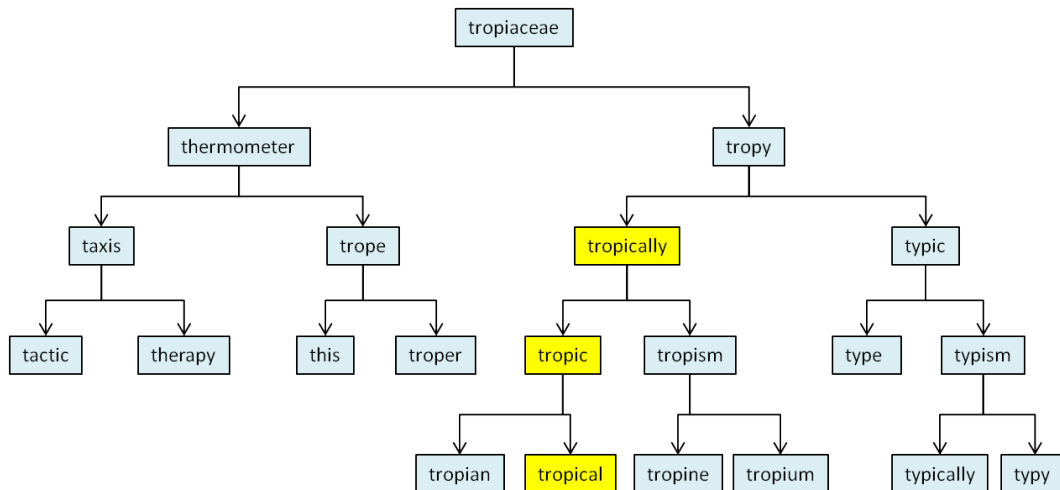


Figure 2: Example of an AVL Tree being used as a Dictionary

3.3 Searching and Deleting in AVL Tree

To search for a key in the AVL tree-based dictionary, we perform a binary search in the tree. If the key is found, we return its associated value. Otherwise, we return a null value to indicate that the key is not present in the dictionary.

To delete a key from the AVL tree-based dictionary, we perform a binary search to find the node containing the key. If the key is found, we delete the node and perform one or more rotations to balance the tree if necessary. If the node to be deleted has two children, we replace it with the successor or predecessor node, which is the node with the smallest or largest key in its right or left subtree, respectively.

3.4 Advantages of AVL Tree over Other Data Structures

The AVL tree-based dictionary has several advantages over other data structures, such as hash tables or binary search trees.

1. AVL trees are self-balancing, which means that the height of the left and right subtrees of any node differ by at most one. This ensures that the worst-case time complexity of search, insert, and delete operations is $O(\log n)$, where n is the number of nodes in the tree. Hash tables have an average-case time complexity of $O(1)$, but their worst-case time complexity can be $O(n)$ if all the keys hash to the same slot. Binary search trees have a worst-case time complexity of $O(n)$ if the tree is unbalanced.
2. AVL trees are more space-efficient than hash tables. The space complexity of an AVL tree is $O(n)$, where n is the number of nodes in the tree. The space complexity of a hash table is $O(n)$, where n is the number of key-value pairs in the table.
3. AVL trees have a guaranteed worst-case time complexity of $O(\log n)$ for search, insert, and delete operations, regardless of the distribution of the keys. Hash tables have an average-case time complexity of $O(1)$, but their worst-case time complexity can be $O(n)$ if all the keys hash to the same slot. Binary search trees have a worst-case time complexity of $O(n)$ if the tree is unbalanced.

3.5 Disadvantages of AVL Tree over Other Data Structures

1. Overhead: Maintaining the balance of AVL trees requires additional operations and memory compared to regular binary search trees, which can add overhead to the implementation.
2. Rotations: Whenever an insertion or deletion violates the balance condition of an AVL tree, rotations are required to restore the balance. These rotations can be complex and time-consuming, especially in larger trees.
3. Limited use: AVL trees are not always the best choice for certain types of applications. For example, in situations where insertions and deletions are infrequent and searches are more common, a simple binary search tree may be a better choice.
4. Implementation complexity: Implementing an AVL tree can be more complex than implementing a regular binary search tree, which can lead to more errors and bugs in the code.

5. Memory overhead: In some cases, AVL trees may consume more memory than other types of binary search trees, due to their need to maintain balance. This can be a concern in memory-constrained environments or when dealing with very large data sets.

4 Platform

Operating System: Arch Linux x86-64

IDEs or Text Editors Used: Visual Studio Code

Compilers : g++ and gcc on linux for C++

5 Test Conditions

1. Input min 10 elements.

6 Input and Output

1. The Elements in Ascending Order, or Descending order

7 Pseudo Code

Pseudo Code for Creation of AVL Trees

```
1  struct Node
2      string word
3      string definition
4      struct Node *l
5      struct Node *r
6      friend class AVL_Tree
7
8  class AVL_Tree
9  public:
10     Node *head
11     AVL_Tree()
12         head = new Node
13         head->l = NULL
14         head->r = NULL
15         head->definition = "AVL Dictionary"
16         head->word = "Head"
```

Pseudo Code for Rotations of AVL Trees

```
1  Node *RR_rotation(Node *parent)
2  Node *t
3  t = parent->r
4  parent->r = t->l
5  t->l = parent
6  cout << "Right-Right Rotation Performed" << endl
7  return t
8  Node *LL_rotation(Node *parent)
9  Node *t
```

```
10     t = parent->l
11     parent->l = t->r
12     t->r = parent
13     cout << "Left-Left Rotation Performed" << endl
14     return t
15 Node *LR_rotation(Node *parent)
16     Node *t
17     t = parent->l
18     parent->l = RR_rotation(t)
19     cout << "Left-Right Rotation Performed" << endl
20     return LL_rotation(parent)
21 Node *RL_rotation(Node *parent)
22     Node *t
23     t = parent->r
24     parent->r = LL_rotation(t)
25     cout << "Right-Left Rotation Performed" << endl
26     return RR_rotation(parent)
```

Pseudo Code for Balancing of AVL Trees

```
1     Node *balance_AVL_Tree(Node *t)
2     int bal_factor = find_difference(t)
3     if (bal_factor > 1)
4         if (find_difference(t->l) > 0)
5             t = LL_rotation(t)
6         else
7             t = LR_rotation(t)
8     else if (bal_factor < -1)
9         if (find_difference(t->r) > 0)
10            t = RL_rotation(t)
11        else
12            t = RR_rotation(t)
13    return t
```

Pseudo Code for Insertion of AVL Trees

```
1     Node *insert_words(Node *r, string word, string definition)
2     if (r == NULL)
3         r = new Node
4         r->word = word
5         r->definition = definition
6         r->l = NULL
7         r->r = NULL
8         return r
9     else if (strcmp(word.c_str(), r->word.c_str()) < 0)
10        r->l = insert_words(r->l, word, definition)
11        r = balance_AVL_Tree(r)
12    else if (strcmp(word.c_str(), r->word.c_str()) >= 0)
13        r->r = insert_words(r->r, word, definition)
14        r = balance_AVL_Tree(r)
15    return r
```

8 Time Complexity

8.1 Creation, Searching, Insertion and Deletion in AVL Trees

- Time Complexity:

$$O(n \log(n))$$

- Space Complexity:

$$O(n)$$

9 Code

9.1 Program

```
1 #include <iostream>
2 #include <cstdio>
3 #include <sstream>
4 #include <algorithm>
5 #include <string.h>
6 #define pow2(n) (1 << (n))
7 using namespace std;
8 struct Node
9 {
10     string word;
11     string definition;
12     struct Node *l;
13     struct Node *r;
14     friend class AVL_Tree;
15 };
16
17 class AVL_Tree
18 {
19 public:
20     Node *head;
21     AVL_Tree()
22     {
23         head = new Node;
24         head->l = NULL;
25         head->r = NULL;
26         head->definition = "AVL Dictionary";
27         head->word = "Head";
28     }
29
30     int find_height(Node *t)
31     {
32         int h = 0;
33         if (t != NULL)
34         {
35             int l_height = find_height(t->l);
36             int r_height = find_height(t->r);
37             int max_height = max(l_height, r_height);
38             h = max_height + 1;
39         }
40         return h;
41     }
42     int find_difference(Node *t)
```



```
43 {
44     int l_height = find_height(t->l);
45     int r_height = find_height(t->r);
46     int b_factor = l_height - r_height;
47     return b_factor;
48 }
49 Node *RR_rotation(Node *parent)
50 {
51     Node *t;
52     t = parent->r;
53     parent->r = t->l;
54     t->l = parent;
55     cout << "Right-Right Rotation Performed" << endl;
56     return t;
57 }
58 Node *LL_rotation(Node *parent)
59 {
60     Node *t;
61     t = parent->l;
62     parent->l = t->r;
63     t->r = parent;
64     cout << "Left-Left Rotation Performed" << endl;
65     return t;
66 }
67 Node *LR_rotation(Node *parent)
68 {
69     Node *t;
70     t = parent->l;
71     parent->l = RR_rotation(t);
72     cout << "Left-Right Rotation Performed" << endl;
73     return LL_rotation(parent);
74 }
75 Node *RL_rotation(Node *parent)
76 {
77     Node *t;
78     t = parent->r;
79     parent->r = LL_rotation(t);
80     cout << "Right-Left Rotation Performed" << endl;
81     return RR_rotation(parent);
82 }
83 Node *balance_AVL_Tree(Node *t)
84 {
85     int bal_factor = find_difference(t);
86     if (bal_factor > 1)
87     {
88         if (find_difference(t->l) > 0)
89             t = LL_rotation(t);
90         else
91             t = LR_rotation(t);
92     }
93     else if (bal_factor < -1)
94     {
95         if (find_difference(t->r) > 0)
96             t = RL_rotation(t);
97         else
98             t = RR_rotation(t);
99     }
100     return t;
101 }
```

```
102 Node *insert_words(Node *r, string word, string definition)
103 {
104     if (r == NULL)
105     {
106         r = new Node;
107         r->word = word;
108         r->definition = definition;
109         r->l = NULL;
110         r->r = NULL;
111         return r;
112     }
113     else if (strcmp(word.c_str(), r->word.c_str()) < 0)
114     {
115         r->l = insert_words(r->l, word, definition);
116         r = balance_AVL_Tree(r);
117     }
118     else if (strcmp(word.c_str(), r->word.c_str()) >= 0)
119     {
120         r->r = insert_words(r->r, word, definition);
121         r = balance_AVL_Tree(r);
122     }
123     return r;
124 }
125 void display_AVL_tree(Node *p, int l)
126 {
127     int i;
128     if (p != NULL)
129     {
130         display_AVL_tree(p->r, l + 1);
131         cout << endl;
132         if (p == head)
133             cout << "Root -> ";
134         for (i = 0; i < l && p != head; i++)
135             cout << endl;
136         cout << p->word << ": " << p->definition << endl;
137         display_AVL_tree(p->l, l + 1);
138     }
139 }
140 void inorder_traversal(Node *t)
141 {
142     if (t == NULL)
143         return;
144     inorder_traversal(t->l);
145     cout << t->word << " ";
146     inorder_traversal(t->r);
147 }
148 void preorder_traversal(Node *t)
149 {
150     if (t == NULL)
151         return;
152     cout << t->word << " ";
153     preorder_traversal(t->l);
154     preorder_traversal(t->r);
155 }
156 void postorder_traversal(Node *t)
157 {
158     if (t == NULL)
159         return;
160     postorder_traversal(t->l);
```

```
161     postorder_traversal(t->r);
162     cout << t->word << " ";
163 }
164 };
165
166 int main()
167 {
168     int c, i;
169     string word, definition;
170     AVL_Tree avl;
171     while (1)
172     {
173         cout << "1.Insert Element into the tree" << endl;
174         cout << "2.show Balanced AVL Tree" << endl;
175         cout << "3.InOrder traversal" << endl;
176         cout << "4.PreOrder traversal" << endl;
177         cout << "5.PostOrder traversal" << endl;
178         cout << "6.Exit" << endl;
179         cout << "Enter your Choice: ";
180         cout << endl;
181         << endl;
182         cin >> c;
183         switch (c)
184         {
185             case 1:
186                 cout << "Enter the word to be inserted: ";
187                 cin >> word;
188                 cout << "Enter the definition of the word: ";
189                 cin >> definition;
190                 avl.head = avl.insert_words(avl.head, word, definition);
191                 break;
192             case 2:
193                 if (avl.head == NULL)
194                 {
195                     cout << "Tree is Empty" << endl;
196                     continue;
197                 }
198                 cout << "Balanced AVL Tree:" << endl;
199                 avl.display_AVL_tree(avl.head, 1);
200                 cout << endl;
201                 break;
202             case 3:
203                 cout << "Inorder Traversal:" << endl;
204                 avl.inorder_traversal(avl.head);
205                 cout << endl;
206                 break;
207             case 4:
208                 cout << "Preorder Traversal:" << endl;
209                 avl.preorder_traversal(avl.head);
210                 cout << endl;
211                 break;
212             case 5:
213                 cout << "Postorder Traversal:" << endl;
214                 avl.postorder_traversal(avl.head);
215                 cout << endl;
216                 break;
217             case 6:
218                 exit(1);
219                 break;
```

Advanced Data Structures - Assignment 9

```
220     default:
221         cout << "Wrong Choice" << endl;
222     }
223 }
224 return 0;
225 }
```

```
1 -> krishnaraj@Krishnaraj-Arch -> /run/media/krishnaraj/Classes/University/Second
   Year/Second Semester/Advanced Data Structures/Programs -> main -> cd "/run/
   media/krishnaraj/Classes/University/Second Year/Second Semester/Advanced Data
   Structures/Programs/" && g++ Assignment_9.cpp -o Assignment_9 && "/run/media/
   krishnaraj/Classes/University/Second Year/Second Semester/Advanced Data
   Structures/Programs/"Assignment_9
2 1.Insert Element into the tree
3 2.show Balanced AVL Tree
4 3.InOrder traversal
5 4.PreOrder traversal
6 5.PostOrder traversal
7 6.Exit
8 Enter your Choice:
9
10 1
11 Enter the word to be inserted: Pineapple
12 Enter the definition of the word: fruit
13 1.Insert Element into the tree
14 2.show Balanced AVL Tree
15 3.InOrder traversal
16 4.PreOrder traversal
17 5.PostOrder traversal
18 6.Exit
19 Enter your Choice:
20
21 1
22 Enter the word to be inserted: Apple
23 Enter the definition of the word: Fruit
24 1.Insert Element into the tree
25 2.show Balanced AVL Tree
26 3.InOrder traversal
27 4.PreOrder traversal
28 5.PostOrder traversal
29 6.Exit
30 Enter your Choice:
31
32 1
33 Enter the word to be inserted: Keys
34 Enter the definition of the word: object
35 1.Insert Element into the tree
36 2.show Balanced AVL Tree
37 3.InOrder traversal
38 4.PreOrder traversal
39 5.PostOrder traversal
40 6.Exit
41 Enter your Choice:
42
43 1
44 Enter the word to be inserted: Laptop
45 Enter the definition of the word: computer
46 Right-Right Rotation Performed
47 Left-Right Rotation Performed
```

Advanced Data Structures - Assignment 9

```
48 Left-Left Rotation Performed
49 1.Insert Element into the tree
50 2.show Balanced AVL Tree
51 3.InOrder traversal
52 4.PreOrder traversal
53 5.PostOrder traversal
54 6.Exit
55 Enter your Choice:
56
57 1
58 Enter the word to be inserted: Guava
59 Enter the definition of the word: fruit
60 1.Insert Element into the tree
61 2.show Balanced AVL Tree
62 3.InOrder traversal
63 4.PreOrder traversal
64 5.PostOrder traversal
65 6.Exit
66 Enter your Choice:
67
68 1
69 Enter the word to be inserted: Arc_Reactor
70 Enter the definition of the word: heart
71 Left-Left Rotation Performed
72 Right-Left Rotation Performed
73 Right-Right Rotation Performed
74 1.Insert Element into the tree
75 2.show Balanced AVL Tree
76 3.InOrder traversal
77 4.PreOrder traversal
78 5.PostOrder traversal
79 6.Exit
80 Enter your Choice:
81
82 1
83 Enter the word to be inserted: Suit
84 Enter the definition of the word: if_ur_nothing_without_it_you_shouldnt_have_it
85 1.Insert Element into the tree
86 2.show Balanced AVL Tree
87 3.InOrder traversal
88 4.PreOrder traversal
89 5.PostOrder traversal
90 6.Exit
91 Enter your Choice:
92
93 1
94 Enter the word to be inserted: Genius
95 Enter the definition of the word: Tony
96 1.Insert Element into the tree
97 2.show Balanced AVL Tree
98 3.InOrder traversal
99 4.PreOrder traversal
100 5.PostOrder traversal
101 6.Exit
102 Enter your Choice:
103
104 1
105 Enter the word to be inserted: Billionnaire
106 Enter the definition of the word: Tony
```

Advanced Data Structures - Assignment 9

```
107 Left-Left Rotation Performed
108 1.Insert Element into the tree
109 2.show Balanced AVL Tree
110 3.InOrder traversal
111 4.PreOrder traversal
112 5.PostOrder traversal
113 6.Exit
114 Enter your Choice:
115
116 1
117 Enter the word to be inserted: Philanthropist
118 Enter the definition of the word: Tony
119 1.Insert Element into the tree
120 2.show Balanced AVL Tree
121 3.InOrder traversal
122 4.PreOrder traversal
123 5.PostOrder traversal
124 6.Exit
125 Enter your Choice:
126
127 2
128 Balanced AVL Tree:
129
130 Suit: if_ur_nothing_without_it_you_shouldnt_have_it
131 Pineapple: fruit
132 Philanthropist: Tony
133 Laptop: computer
134 Keys: object
135 Root -> Head: AVL Dictionary
136 Guava: fruit
137 Genius: Tony
138 Billionnaire: Tony
139 Arc_Reactor: heart
140 Apple: Fruit
141
142 1.Insert Element into the tree
143 2.show Balanced AVL Tree
144 3.InOrder traversal
145 4.PreOrder traversal
146 5.PostOrder traversal
147 6.Exit
148 Enter your Choice:
149
150 3
151 Inorder Traversal:
152 Apple Arc_Reactor Billionnaire Genius Guava Head Keys Laptop Philanthropist
    Pineapple Suit
153 1.Insert Element into the tree
154 2.show Balanced AVL Tree
155 3.InOrder traversal
156 4.PreOrder traversal
157 5.PostOrder traversal
158 6.Exit
159 Enter your Choice:
160
161 4
162 Preorder Traversal:
163 Head Arc_Reactor Apple Genius Billionnaire Guava Laptop Keys Pineapple
    Philanthropist Suit
```

```
164 1.Insert Element into the tree
165 2.show Balanced AVL Tree
166 3.InOrder traversal
167 4.PreOrder traversal
168 5.PostOrder traversal
169 6.Exit
170 Enter your Choice:
171
172 5
173 Postorder Traversal:
174 Apple Billionnaire Guava Genius Arc_Reactor Keys Philanthropist Suit Pineapple
    Laptop Head
175 1.Insert Element into the tree
176 2.show Balanced AVL Tree
177 3.InOrder traversal
178 4.PreOrder traversal
179 5.PostOrder traversal
180 6.Exit
181 Enter your Choice: 6
```

10 Conclusion

Thus, we have understood the importance and use of AVL trees as a Data structure, and how they are better and more efficient than Binary Search Trees.

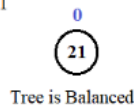
11 FAQ

1. Discuss AVL trees with suitable example?

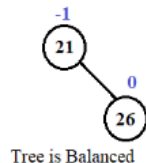
AVL trees are a type of self-balancing binary search tree that maintain a balance between left and right subtrees to ensure fast search, insertion, and deletion operations with a worst-case time complexity of $O(\log n)$, where n is the number of nodes in the tree.

Let us look at an example of an AVL tree. Consider the following AVL tree:
AVL Tree for the given Sequence 21, 26, 30, 9, 4, 14

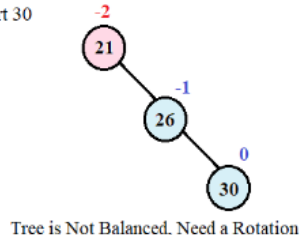
Step 1 - Insert 21



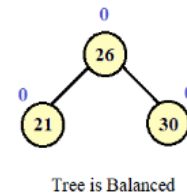
Step 2 - Insert 26



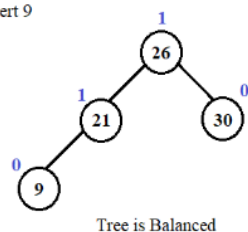
Step 3 - Insert 30



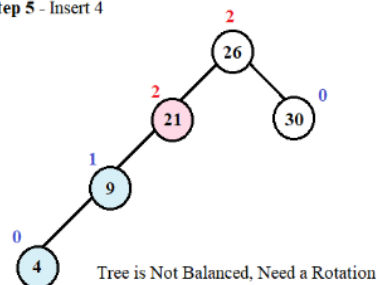
LL Rotation



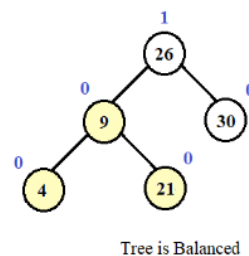
Step 4 - Insert 9



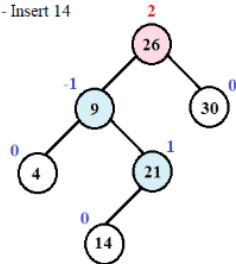
Step 5 - Insert 4



RR Rotation

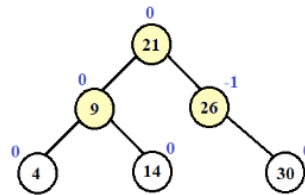


Step 6 - Insert 14



Tree is Not Balanced, Need a Rotation

LR Rotation



Tree is Balanced

2. Compute the time complexity of AVL tree creation?

The time complexity of creating an AVL tree depends on the number of elements n that need to be inserted. In the worst case, when the elements are inserted in sorted order, the time complexity of AVL tree creation is $O(n \log n)$, since each insertion may require a series of rotations to maintain balance. In the best case, when the elements are inserted in a balanced manner, the time complexity of AVL tree creation is $O(n)$. This is because each node is inserted and balanced in constant time, so the total time is proportional to the number of nodes inserted.

In practice, AVL trees are very efficient for creating and maintaining a dictionary because they provide fast search, insertion, and deletion operations. However, they do require additional memory to store the balance factor of each node, and the overhead of maintaining balance can add to the running time of operations. Overall, AVL trees are a good choice for applications that require a balanced binary search tree and where the number of insertions and deletions is not too frequent.