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. 1. To study the concept of AVL trees
2. 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.

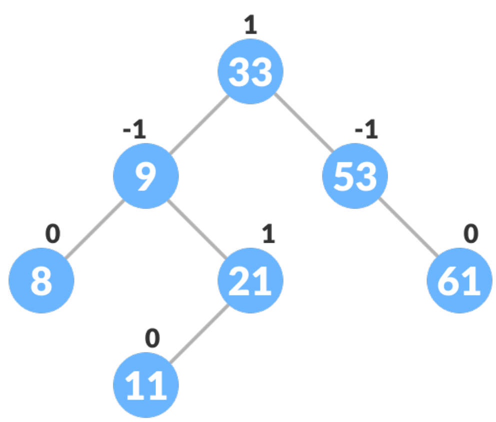


Figure 1: Example of an AVL Tree

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.

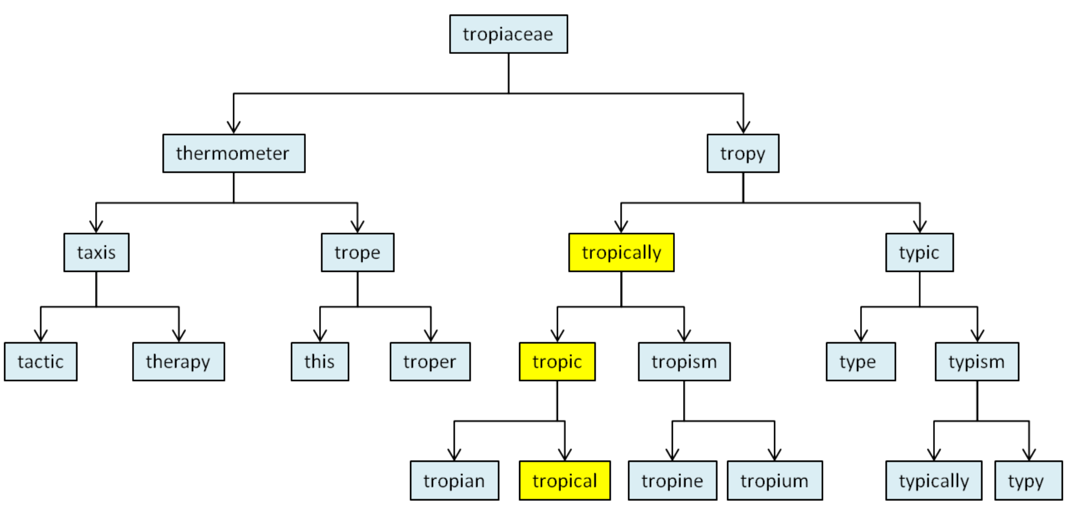


Figure 2: Example of an AVL Tree being used as a Dictionary

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

```
    struct Node
        string word
        string definition
        struct Node *l
        struct Node *r
        friend class AVL_Tree

    class AVL_Tree
    public:
        Node *head
        AVL_Tree()
            head = new Node
            head->l = NULL
            head->r = NULL
            head->definition = "AVL Dictionary"
            head->word = "Head"
```

Pseudo Code for Rotations of AVL Trees

```
    Node *RR_rotation(Node *parent)
    Node *t
    t = parent->r
    parent->r = t->l
    t->l = parent
    cout << "Right-Right Rotation Performed" << endl
    return t
  Node *LL_rotation(Node *parent)
    Node *t
```

```
    t = parent->l
    parent->l = t->r
    t->r = parent
    cout << "Left-Left Rotation Performed" << endl
    return t
  Node *LR_rotation(Node *parent)
    Node *t
    t = parent->l
    parent->l = RR_rotation(t)
    cout << "Left-Right Rotation Performed" << endl
    return LL_rotation(parent)
  Node *RL_rotation(Node *parent)
    Node *t
    t = parent->r
    parent->r = LL_rotation(t)
    cout << "Right-Left Rotation Performed" << endl
    return RR_rotation(parent)
```

Pseudo Code for Balancing of AVL Trees

```
    Node *balance_AVL_Tree(Node *t)
    int bal_factor = find_difference(t)
    if (bal_factor > 1)
      if (find_difference(t->l) > 0)
        t = LL_rotation(t)
      else
        t = LR_rotation(t)
    else if (bal_factor < -1)
      if (find_difference(t->r) > 0)
        t = RL_rotation(t)
      else
        t = RR_rotation(t)
    return t
```

Pseudo Code for Insertion of AVL Trees

```
  Node *insert_words(Node *r, string word, string definition)
    if (r == NULL)
      r = new Node
      r->word = word
      r->definition = definition
      r->l = NULL
      r->r = NULL
      return r
    else if (strcmp(word.c_str(), r->word.c_str()) < 0)
      r->l = insert_words(r->l, word, definition)
      r = balance_AVL_Tree(r)
    else if (strcmp(word.c_str(), r->word.c_str()) >= 0)
      r->r = insert_words(r->r, word, definition)
      r = balance_AVL_Tree(r)
    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

```
#include <iostream>
#include <cstdio>
#include <sstream>
#include <algorithm>
#include <string.h>
#define pow2(n) (1 << (n))
using namespace std;
struct Node
{
  string word;
  string definition;
  struct Node *l;
  struct Node *r;
  friend class AVL_Tree;
};

class AVL_Tree
{
public:
  Node *head;
  AVL_Tree()
  {
    head = new Node;
    head->l = NULL;
    head->r = NULL;
    head->definition = "AVL Dictionary";
    head->word = "Head";
  }

  int find_height(Node *t)
  {
    int h = 0;
    if (t != NULL)
    {
      int l_height = find_height(t->l);
      int r_height = find_height(t->r);
      int max_height = max(l_height, r_height);
      h = max_height + 1;
    }
    return h;
  }
  int find_difference(Node *t)
```

```
{
  int l_height = find_height(t->l);
  int r_height = find_height(t->r);
  int b_factor = l_height - r_height;
  return b_factor;
}
Node *RR_rotation(Node *parent)
{
  Node *t;
  t = parent->r;
  parent->r = t->l;
  t->l = parent;
  cout << "Right-Right Rotation Performed" << endl;
  return t;
}
Node *LL_rotation(Node *parent)
{
  Node *t;
  t = parent->l;
  parent->l = t->r;
  t->r = parent;
  cout << "Left-Left Rotation Performed" << endl;
  return t;
}
Node *LR_rotation(Node *parent)
{
  Node *t;
  t = parent->l;
  parent->l = RR_rotation(t);
  cout << "Left-Right Rotation Performed" << endl;
  return LL_rotation(parent);
}
Node *RL_rotation(Node *parent)
{
  Node *t;
  t = parent->r;
  parent->r = LL_rotation(t);
  cout << "Right-Left Rotation Performed" << endl;
  return RR_rotation(parent);
}
Node *balance_AVL_Tree(Node *t)
{
  int bal_factor = find_difference(t);
  if (bal_factor > 1)
  {
    if (find_difference(t->l) > 0)
      t = LL_rotation(t);
    else
      t = LR_rotation(t);
  }
  else if (bal_factor < -1)
  {
    if (find_difference(t->r) > 0)
      t = RL_rotation(t);
    else
      t = RR_rotation(t);
  }
  return t;
}
```

```
Node *insert_words(Node *r, string word, string definition)
{
  if (r == NULL)
  {
    r = new Node;
    r->word = word;
    r->definition = definition;
    r->l = NULL;
    r->r = NULL;
    return r;
  }
  else if (strcmp(word.c_str(), r->word.c_str()) < 0)
  {
    r->l = insert_words(r->l, word, definition);
    r = balance_AVL_Tree(r);
  }
  else if (strcmp(word.c_str(), r->word.c_str()) >= 0)
  {
    r->r = insert_words(r->r, word, definition);
    r = balance_AVL_Tree(r);
  }
  return r;
}
void display_AVL_tree(Node *p, int l)
{
  int i;
  if (p != NULL)
  {
    display_AVL_tree(p->r, l + 1);
    cout << endl;
    if (p == head)
      cout << "Root -> ";
    for (i = 0; i < l && p != head; i++)
      cout << endl;
    cout << p->word << ": " << p->definition << endl;
    display_AVL_tree(p->l, l + 1);
  }
}
void inorder_traversal(Node *t)
{
  if (t == NULL)
    return;
  inorder_traversal(t->l);
  cout << t->word << " ";
  inorder_traversal(t->r);
}
void preorder_traversal(Node *t)
{
  if (t == NULL)
    return;
  cout << t->word << " ";
  preorder_traversal(t->l);
  preorder_traversal(t->r);
}
void postorder_traversal(Node *t)
{
  if (t == NULL)
    return;
  postorder_traversal(t->l);
```

```
    postorder_traversal(t->r);
    cout << t->word << " ";
  }
};

int main()
{
  int c, i;
  string word, definition;
  AVL_Tree avl;
  while (1)
  {
    cout << "1.Insert Element into the tree" << endl;
    cout << "2.show Balanced AVL Tree" << endl;
    cout << "3.InOrder traversal" << endl;
    cout << "4.PreOrder traversal" << endl;
    cout << "5.PostOrder traversal" << endl;
    cout << "6.Exit" << endl;
    cout << "Enter your Choice: ";
    cout << endl
       << endl;
    cin >> c;
    switch (c)
    {
    case 1:
      cout << "Enter the word to be inserted: ";
      cin >> word;
      cout << "Enter the definition of the word: ";
      cin >> definition;
      avl.head = avl.insert_words(avl.head, word, definition);
      break;
    case 2:
      if (avl.head == NULL)
      {
        cout << "Tree is Empty" << endl;
        continue;
      }
      cout << "Balanced AVL Tree:" << endl;
      avl.display_AVL_tree(avl.head, 1);
      cout << endl;
      break;
    case 3:
      cout << "Inorder Traversal:" << endl;
      avl.inorder_traversal(avl.head);
      cout << endl;
      break;
    case 4:
      cout << "Preorder Traversal:" << endl;
      avl.preorder_traversal(avl.head);
      cout << endl;
      break;
    case 5:
      cout << "Postorder Traversal:" << endl;
      avl.postorder_traversal(avl.head);
      cout << endl;
      break;
    case 6:
      exit(1);
      break;
```

```cpp
    default:
      cout << "Wrong Choice" << endl;
    }
  }
  return 0;
}
```

```
-> 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
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Pineapple
Enter the definition of the word: fruit
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Apple
Enter the definition of the word: Fruit
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Keys
Enter the definition of the word: object
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Laptop
Enter the definition of the word: computer
Right-Right Rotation Performed
Left-Right Rotation Performed
```

```
Left-Left Rotation Performed
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Guava
Enter the definition of the word: fruit
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Arc_Reactor
Enter the definition of the word: heart
Left-Left Rotation Performed
Right-Left Rotation Performed
Right-Right Rotation Performed
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Suit
Enter the definition of the word: if_ur_nothing_without_it_you_shouldnt_have_it
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Genius
Enter the definition of the word: Tony
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Billionnaire
Enter the definition of the word: Tony
```

```
Left-Left Rotation Performed
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

1
Enter the word to be inserted: Philanthropist
Enter the definition of the word: Tony
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

2
Balanced AVL Tree:

Suit: if_ur_nothing_without_it_you_shouldnt_have_it
Pineapple: fruit
Philanthropist: Tony
Laptop: computer
Keys: object
Root -> Head: AVL Dictionary
Guava: fruit
Genius: Tony
Billionnaire: Tony
Arc_Reactor: heart
Apple: Fruit

1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

3
Inorder Traversal:
Apple Arc_Reactor Billionnaire Genius Guava Head Keys Laptop Philanthropist
    Pineapple Suit
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

4
Preorder Traversal:
Head Arc_Reactor Apple Genius Billionnaire Guava Laptop Keys Pineapple
    Philanthropist Suit
```

```
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:

5
Postorder Traversal:
Apple Billionnaire Guava Genius Arc_Reactor Keys Philanthropist Suit Pineapple
    Laptop Head
1.Insert Element into the tree
2.show Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 6
```

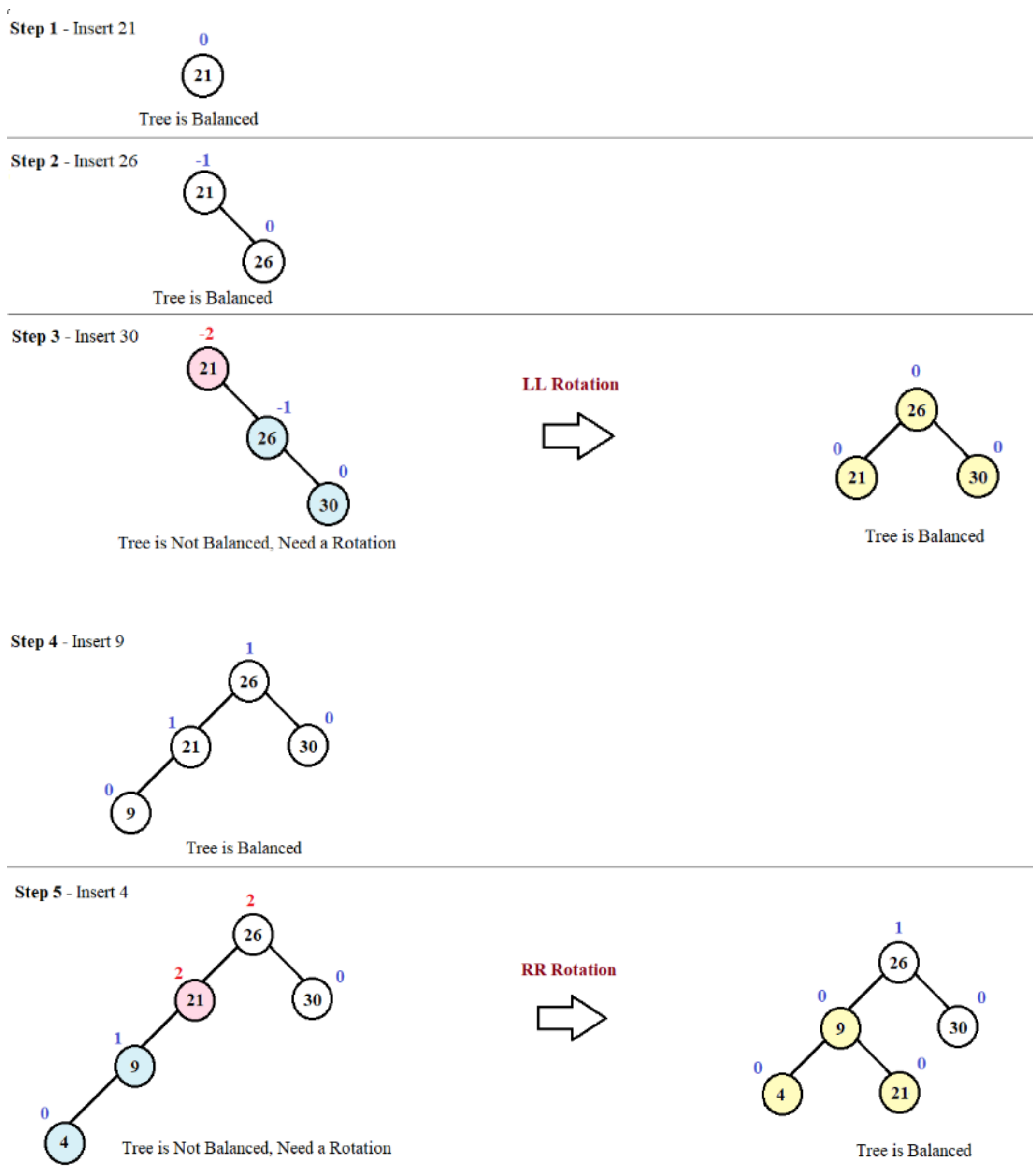
10 Conclusion

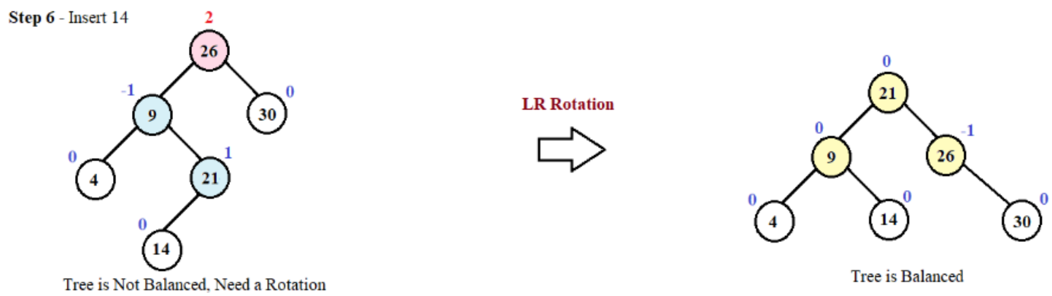
Thus, we have understood the importance and use of AVL trees as a Data structure, and how they are better and more efficient than Binary Search Trees.

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





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.