MIT Art Design and Technology University

MIT School of Computing, Pune

Department of Computer Science and Engineering

Second Year B. Tech

Academic Year 2022-2023. (SEM-II)

Subject: Advance Data Structures Laboratory

Assignment 11

Assignment Title: A Dictionary stores keywords & its meanings. 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.

Aim: Implement Dictionary using AVL tree.

Prerequisite:

1. Basic concepts of Dictionary.
2. Basic concepts of AVL tree and its rotations.
3. Procedure to create height balanced tree.

Objectives:

Implement a Program in python for the following operations:

1. Create a Dictionary using any data structure.
2. Use AVL tree for performing different operations on dictionary.

Outcomes:

Upon Completion of the assignment the students will be able to

1. Create dictionary using any data structure.

2. Understand and analyse creation of dictionary using AVL tree.

Theory:

AVL tree is a self-balancing BST, where the difference between heights of left and right subtrees cannot be more than one for all nodes. Named after their inventors, Adelson-Velskii and Landis.

Searching of desired node is faster due to balancing of tree height.

Definition-

- An AVL tree is a balanced binary search tree.
- In an AVL tree, balance factor of every node is either -1, 0 or +1.
- Every subtree is an AVL tree.

Balance Factor = Height of Left Subtree – Height of Right Subtree

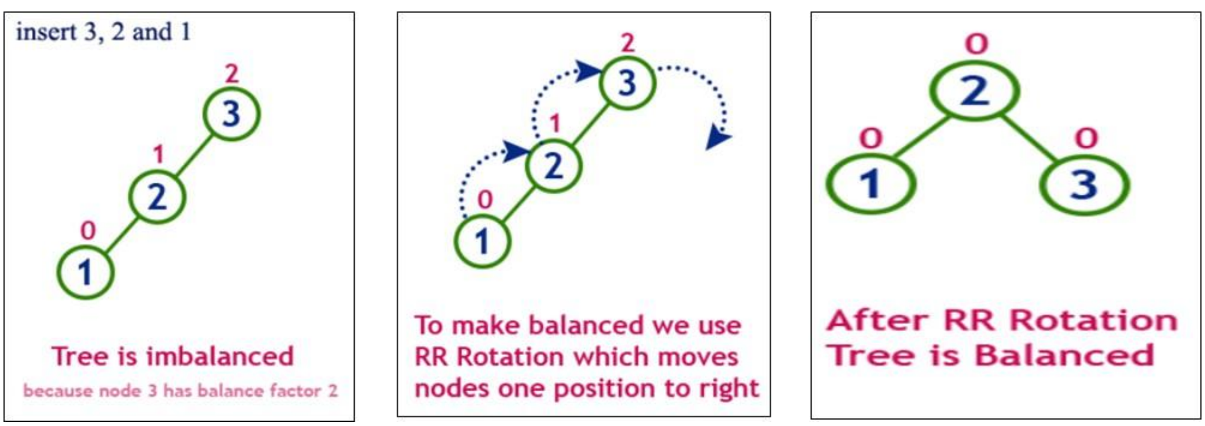
AVL Tree Rotations:

Rotation is the process of moving nodes either to left or to right to make the tree balanced.

- Left Rotation (LL Rotation)
- Right Rotation (RR Rotation)
- Left Right Rotation (LR Rotation)
- Right Left Rotation (RL Rotation)

LL Rotations – Single Right Rotation

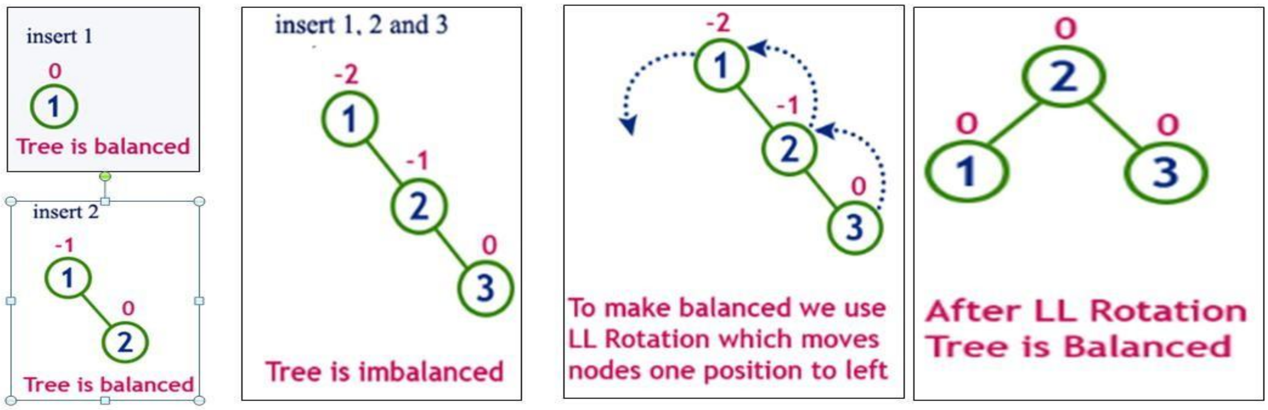
- When tree is unbalanced because of adding left child in the left subtree of an unbalanced node, we need to use LL rotations to make balanced tree.
- In LL Rotation, every node moves one position to right from the current position.



RR Rotations – Single Left Rotation

- When tree is unbalanced because of adding right child in the right subtree of an unbalanced node, we need to use RR rotations to make balanced tree.

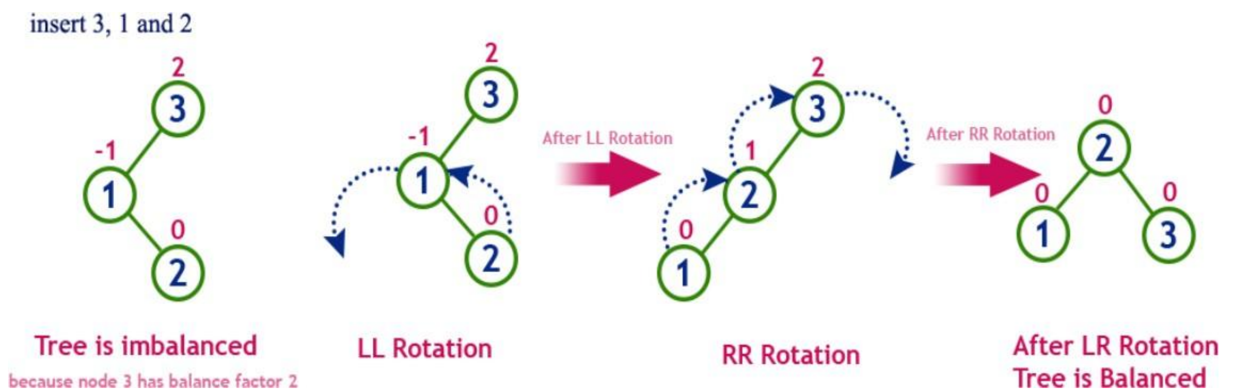
- In RR Rotation, every node moves one position to left from the current position.



LR Rotations – Double Rotations

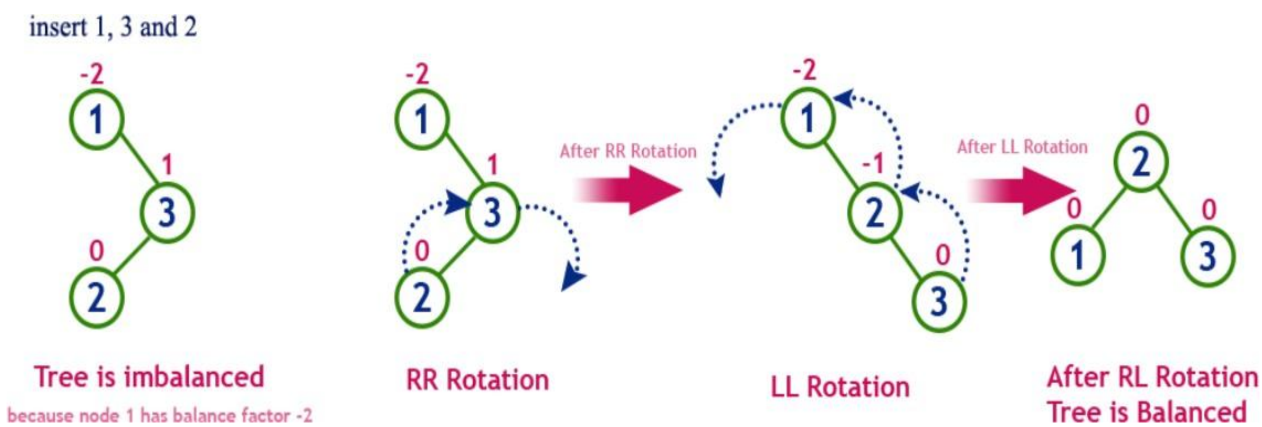
The LR Rotation is sequence of single left rotation followed by single right rotation.

- When tree is unbalanced because of adding right child in the left subtree of an unbalanced node, we need to use LR rotations to make balanced tree.
- Tree is unbalanced because of adding right child in the left subtree of an unbalanced node. The LR Rotation is sequence of single left rotation followed by single rightrotation.



RL Rotations – Double Rotations

- When tree is unbalanced because of adding left child in the right subtree of an unbalanced node, we need to use RL rotations to make balanced tree.
- The RL Rotation is sequence of single right rotation followed by single left rotation.



Creation of AVL tree:

- Insert the element in the AVL tree in the same way the insertion is performed in BST.
- After insertion, check the balance factor of each node of the resulting tree and consider following cases for same.
- **Case 1: Balance factor of each node IS {-1,0,1}**
    - The tree is balanced.
    - Conclude the operation.
    - Insert the next element if any.
- **Case2: Balance factor of each node is NOT {-1,0,1}**
    - The tree is imbalanced.
    - Perform the suitable **ROTATION(s)** to balance the tree.
    - After the tree is balanced, insert the next element if any.

Conclusion:

We have created dictionary using AVL tree.

Frequently Asked Questions:

1. List out properties of AVL tree.
2. What is a need of AVL tree?
3. Create AVL tree for following elements:
   21, 26, 30, 9, 4, 14, 28, 18, 15, 10

MIT SCHOOL OF ENGINEERING
Rajbaug, Loni-Kalbhor, Pune

MIT-ADT UNIVERSITY
PUNE, INDIA

Name : Sourav Shailesh Toshniwal
Class : SY CSE - 1
Roll no. : 2213047.
Subject : ADSL

Assignment - 11

1. List out properties of AVL tree.

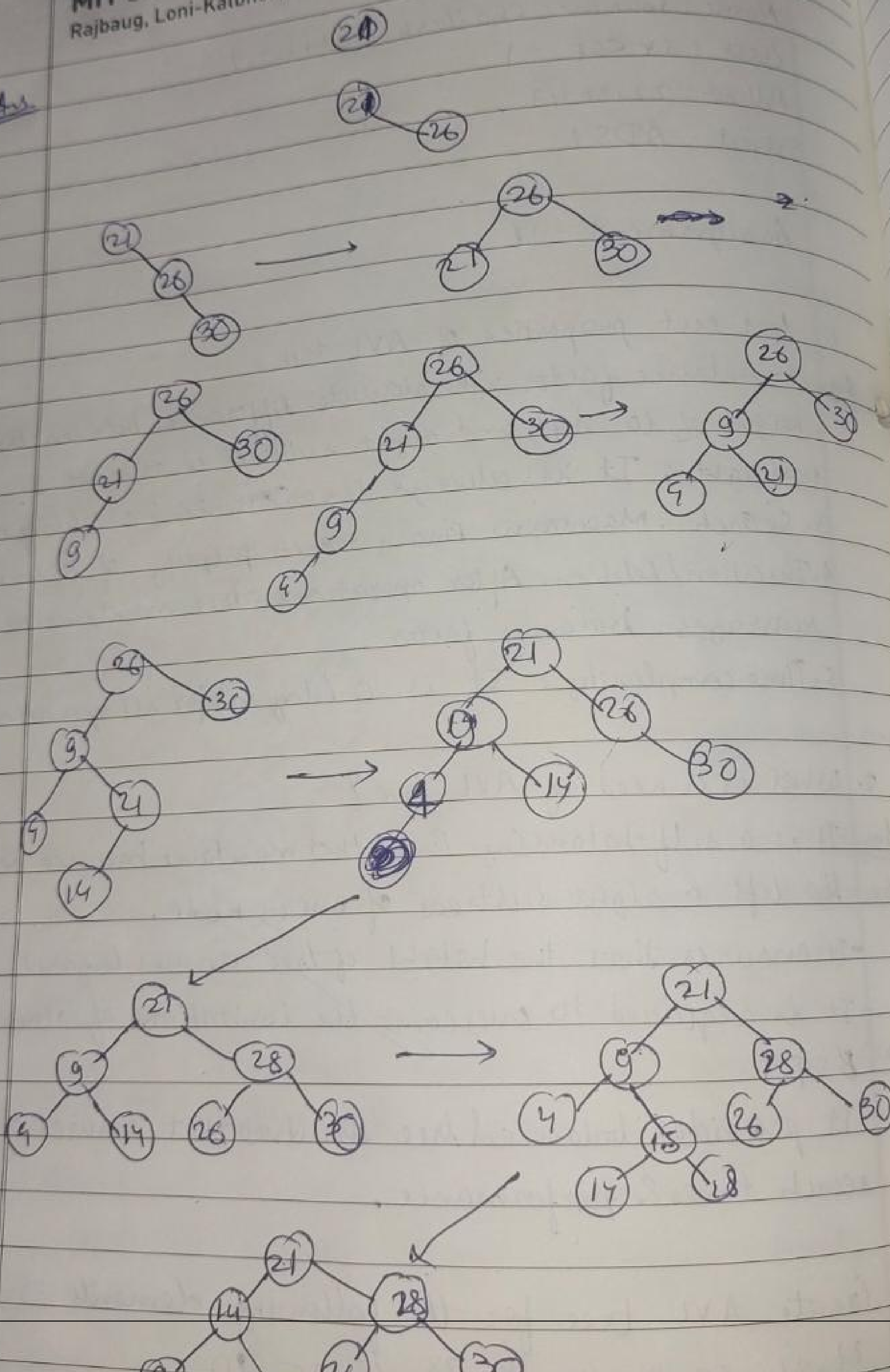
Ans 1. Balance factor : The absolute difference between the height of its left and right subtrees is at most 1.
2. Height : It is always logarithmic i.e $O(\log n)$.
3. Search : Maintains Binary search property of $O(\log n)$
4. Insertion/Deletion: After operations, tree maintains or rearrages balance factor.
5. Time complexity : It is $O(\log n)$ for all operations.

2. What is a need of AVL tree?

Ans - It is a self-balancing BST that maintains balance between the left & right subtrees of every node.
- It ensures that the height of tree remains logarithmic.
- It is required to overcome the limitations of standard BST.
- It provides balanced tree structure that ensures efficient search times & performance.

3. Create AVL tree for the following elements :
21, 26, 30, 9, 4, 14, 28, 18, 15, 10

Ans.



Code:

```
#include <iostream>
#include <cstdio>
#include <sstream>
#include <algorithm>
#define pow2(n) (1 << (n))
using namespace std;

struct avl_node
{
    int data;
    struct avl_node *left;
    struct avl_node *right;
} *root;
```

```
class avlTree
{
public:
    int height(avl_node *);
    int diff(avl_node *);
    avl_node *rr_rotation(avl_node *);
    avl_node *ll_rotation(avl_node *);
    avl_node *lr_rotation(avl_node *);
    avl_node *rl_rotation(avl_node *);
    avl_node *balance(avl_node *);
    avl_node *insert(avl_node *, int);
    void display(avl_node *, int);
    void inorder(avl_node *);
    void preorder(avl_node *);
    void postorder(avl_node *);
    avlTree()
    {
        root = NULL;
    }
};
```

```
int main()
{
    int choice, item;
    avlTree avl;
    while (1)
    {
        cout << "\n---------------------" << endl;
        cout << "AVL Tree Implementation" << endl;
        cout << "\n---------------------" << endl;
        cout << "1.Insert Element into the tree" << endl;
        cout << "2.Display 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: ";
        cin >> choice;
        switch (choice)
        {
        case 1:
            cout << "Enter value to be inserted: ";
            cin >> item;
            root = avl.insert(root, item);
            break;
        case 2:
```

```
            if (root == NULL)
            {
                cout << "Tree is Empty" << endl;
                continue;
            }
            cout << "Balanced AVL Tree:" << endl;
            avl.display(root, 1);
            break;
        case 3:
            cout << "Inorder Traversal:" << endl;
            avl.inorder(root);
            cout << endl;
            break;
        case 4:
            cout << "Preorder Traversal:" << endl;
            avl.preorder(root);
            cout << endl;
            break;
        case 5:
            cout << "Postorder Traversal:" << endl;
            avl.postorder(root);
            cout << endl;
            break;
        case 6:
            exit(1);
            break;
        default:
            cout << "Wrong Choice" << endl;
        }
    }
    return 0;
}
```

```
int avlTree::height(avl_node *temp)
{
    int h = 0;
    if (temp != NULL)
    {
        int l_height = height(temp->left);
        int r_height = height(temp->right);
        int max_height = max(l_height, r_height);
        h = max_height + 1;
    }
    return h;
}
```

```
int avlTree::diff(avl_node *temp)
{
    int l_height = height(temp->left);
    int r_height = height(temp->right);
    int b_factor = l_height - r_height;
    return b_factor;
}
```

```
avl_node *avlTree::rr_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->right;
    parent->right = temp->left;
    temp->left = parent;
    return temp;
```

```
}
```

```
avl_node *avlTree::ll_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->left;
    parent->left = temp->right;
    temp->right = parent;
    return temp;
}
```

```
avl_node *avlTree::lr_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->left;
    parent->left = rr_rotation(temp);
    return ll_rotation(parent);
}
```

```
avl_node *avlTree::rl_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->right;
    parent->right = ll_rotation(temp);
    return rr_rotation(parent);
}
```

```
avl_node *avlTree::balance(avl_node *temp)
{
    int bal_factor = diff(temp);
    if (bal_factor > 1)
    {
        if (diff(temp->left) > 0)
            temp = ll_rotation(temp);
        else
            temp = lr_rotation(temp);
    }
    else if (bal_factor < -1)
    {
        if (diff(temp->right) > 0)
            temp = rl_rotation(temp);
        else
            temp = rr_rotation(temp);
    }
    return temp;
}
```

```
avl_node *avlTree::insert(avl_node *root, int value)
{
    if (root == NULL)
    {
        root = new avl_node;
        root->data = value;
        root->left = NULL;
        root->right = NULL;
        return root;
    }
    else if (value < root->data)
    {
        root->left = insert(root->left, value);
        root = balance(root);
```

```
    }
    else if (value >= root->data)
    {
        root->right = insert(root->right, value);
        root = balance(root);
    }
    return root;
}
```

```
void avlTree::display(avl_node *ptr, int level)
{
    int i;
    if (ptr != NULL)
    {
        display(ptr->right, level + 1);
        printf("\n");
        if (ptr == root)
            cout << "Root -> ";
        for (i = 0; i < level && ptr != root; i++)
            cout << "        ";
        cout << ptr->data;
        display(ptr->left, level + 1);
    }
}
```

```
void avlTree::inorder(avl_node *tree)
{
    if (tree == NULL)
        return;
    inorder(tree->left);
    cout << tree->data << "  ";
    inorder(tree->right);
}
```

```
void avlTree::preorder(avl_node *tree)
{
    if (tree == NULL)
        return;
    cout << tree->data << "  ";
    preorder(tree->left);
    preorder(tree->right);
}
```

```
void avlTree::postorder(avl_node *tree)
{
    if (tree == NULL)
        return;
    postorder(tree->left);
    postorder(tree->right);
    cout << tree->data << "  ";
}
```

Output:

```
PS C:\SOURAV\CODE\C++ language codes\ADS assignment> cd "c:\SOURAV\CODE\C++ language codes\ADS assignment\" ; if ($?) { g++ A

---------------------
AVL Tree Implementation

---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 2

---------------------
AVL Tree Implementation

---------------------
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 5

---------------------
AVL Tree Implementation

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

                5
Root -> 2
                2
```