Data Structures and Algorithms

Prepared by: Mohamed Ayman

Algorithm Engineer at Valeo

Deep Learning Researcher and Teaching Assistant

at The American University in Cairo (AUC)

spring 2020

sw.eng.MohamedAyman@gmail.com

linkedin.com/in/cs-MohamedAyman

github.com/cs-MohamedAyman

codeforces.com/profile/Mohamed_Ayman

Lecture 10

Self Balancing Binary Search Tree

AVL Tree

Course Roadmap

Part 2: Non-Linear Data Structures

Lecture 8: Binary Tree

Lecture 9: Binary Search Tree

Lecture 10: Self Balancing Binary Search Tree

Lecture 11: Binary Heap Tree

Lecture 12: Hash Table

Lecture 13: Graph

Lecture 14: STL in C++ (Non-Linear Data Structures)

Lecture Agenda

We will discuss in this lecture the following topics

1- Introduction to AVL Tree

2- Rotation Operation

3- Insertion Operation

4- Deletion Operation

5- Search Operation

6- Traverse Operation

7- Time Complexity & Space Complexity


Let's
START UP

Lecture Agenda

Section 1: Introduction to AVL Tree

Section 2: Rotation Operation

Section 3: Insertion Operation

Section 4: Deletion Operation

Section 5: Search Operation

Section 6: Traverse Operation

Section 7: Time Complexity & Space Complexity

Introduction to AVL Tree

- In the early 60's G.M. Adelson-Velsky and E.M. Landis invented the first self-balancing binary search tree data structure, calling it AVL Tree. An AVL tree is a binary search tree with a self-balancing condition stating that the difference between the height of the left and right subtrees cannot be no more than one, This condition, restored after each tree modification, forces the general shape of an AVL tree.

Georgy Adelson-Velsky
Jan 1922 - Apr 2014 (aged 92)

Evgenii Landis
Oct 1921 - Dec 1997 (aged 76)

Introduction to AVL Tree

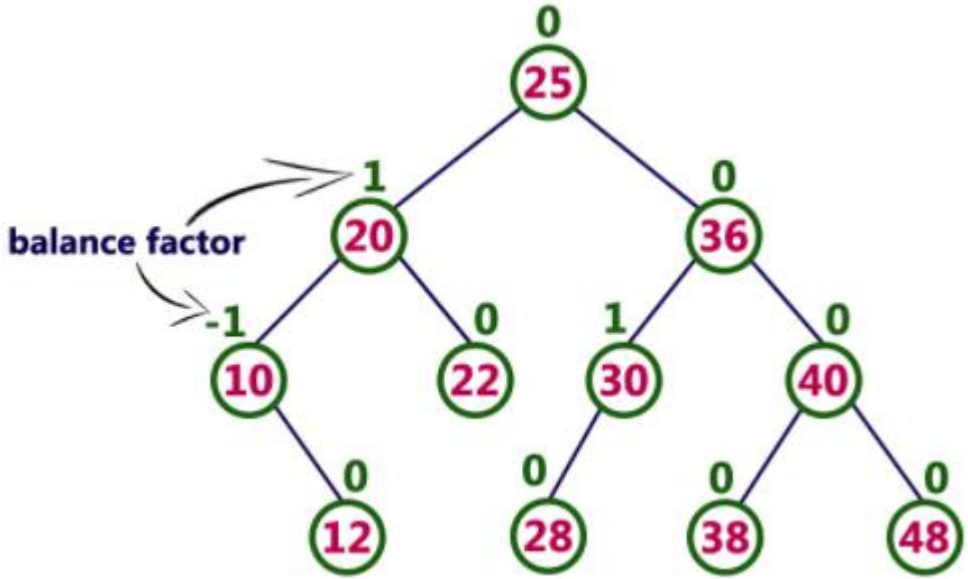
- Why balance is so important? Consider a binary search tree obtained by starting with an empty tree and inserting some values in the following order 1,2,3,4,5. The BST represents the worst case scenario in which the running time of all common operations such as search, insertion and deletion are $\Theta(n)$. By applying a balance condition we ensure that the worst case running time of each common operation is $\Theta(\log n)$. The height of an AVL tree with n nodes is $\Theta(\log n)$ regardless of the order in which values are inserted.
- The AVL balance condition, known also as the node balance factor represents an additional piece of information stored for each node. This is combined with a technique that efficiently restores the balance condition for the tree. In an AVL tree the inventors make use of a well-known technique called tree rotation.
- AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right sub-trees cannot be more than one for all nodes. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if for every node, height of its children differ by at most one.

Introduction to AVL Tree

- This tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree. Every AVL Tree is a binary search tree but all the Binary Search Trees need not to be AVL trees.

- An AVL tree node

```
struct node {
    int data;
    node* left;
    node* right;
    int height;
};
```



Lecture Agenda

- ✓ Section 1: Introduction to AVL Tree
- **Section 2: Rotation Operation**
- Section 3: Insertion Operation
- Section 4: Deletion Operation
- Section 5: Search Operation
- Section 6: Traverse Operation
- Section 7: Time Complexity & Space Complexity

Rotation Operation - AVL Tree

- A tree rotation is a constant time operation on a binary search tree that changes the shape of a tree while preserving standard BST properties. There are left and right rotations both of them decrease the height of a BST by moving smaller subtrees down and larger subtrees up.
- The algorithm that we present here verifies that the left and right subtrees differ at most in height by 1. If this property is not present then we perform the correct rotation. Notice that we use two new algorithms that represent double rotations. These algorithms are named Left and Right, and Right and Left Rotation.
- The algorithms are self documenting in their names, e.g. Left and Right Rotation first performs a left rotation and then subsequently a right rotation. The AVL balance condition, known also as the node balance factor represents an additional piece of information stored for each node. This is combined with a technique that efficiently restores the balance condition for the tree. In an AVL tree the inventors make use of a well-known technique called tree rotation.
- In AVL tree, after performing every operation, like insertion and deletion we need to check the balance factor of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. We use rotation operations to make the tree balanced whenever the tree is becoming imbalanced due t any operation. Rotation operations are used to make a tree balanced. There are four rotations and they are classified into two types.

Rotation Operation - AVL Tree

- Single Rotation:

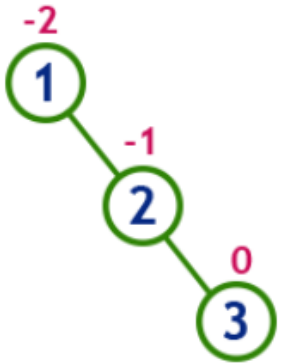
- Left Rotation (LL Rotation): In LL Rotation every node moves one position to left from the current position. To understand LL Rotation, let us consider following situation in an AVL Tree.
- Right Rotation (RR Rotation): In LL Rotation every node moves one position to right from the current position. To understand LL Rotation, let us consider following situation in an AVL Tree.

- Double Rotation:

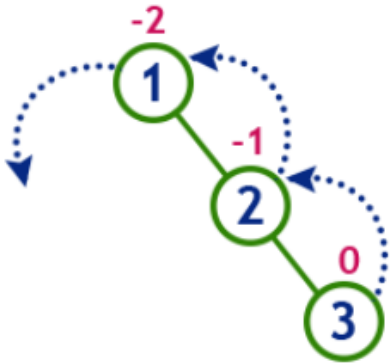
- Left Right Rotation (LR Rotation): The LR Rotation is combination of single left rotation followed by single right rotation. In LR Rotation, first every node moves one position to left then one position to right from the current position. To understand LR Rotation, let us consider the following situation in an AVL Tree.
- Right Left Rotation (RL Rotation): The RL Rotation is combination of single right rotation followed by single left rotation. In RL Rotation, first every node moves one position to right then one position to left from the current position. To understand LR Rotation, let us consider the following situation in an AVL Tree.

Rotation Operation - AVL Tree

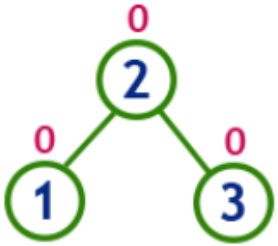
- Single Rotation:
- Left Rotation (LL Rotation):



Tree is imbalanced



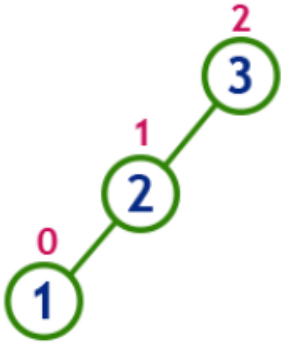
To make balanced we use LL Rotation which moves nodes one position to left



After LL Rotation Tree is Balanced

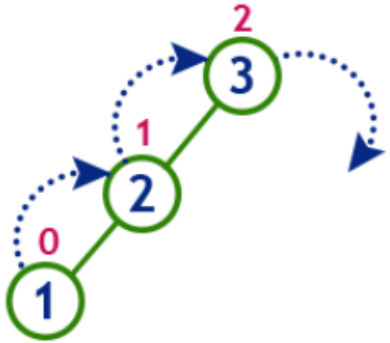
Rotation Operation - AVL Tree

- Single Rotation:
- Right Rotation (RR Rotation):

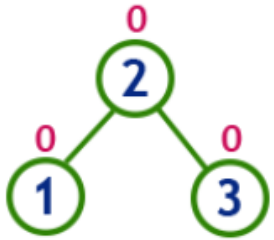


Tree is imbalanced

because node 3 has balance factor 2



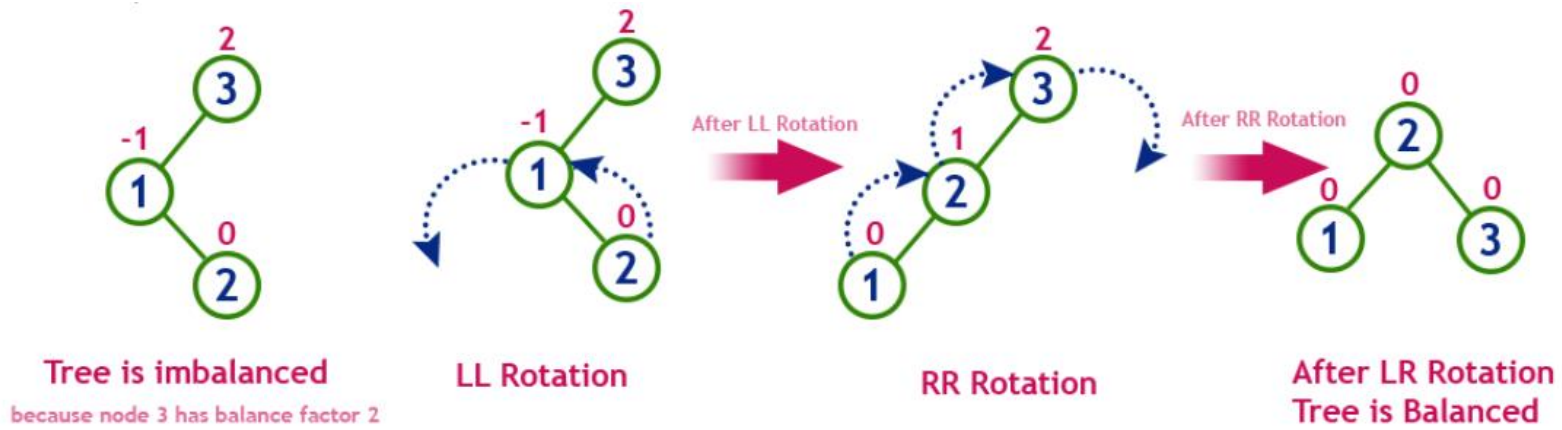
To make balanced we use RR Rotation which moves nodes one position to right



After RR Rotation Tree is Balanced

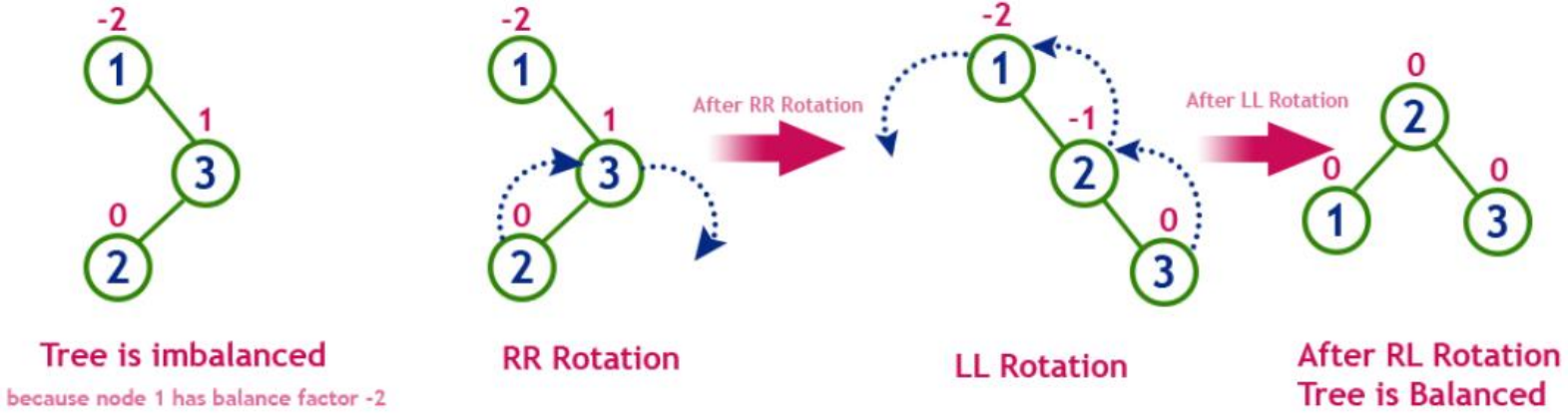
Rotation Operation - AVL Tree

- Double Rotation:
- Left Right Rotation (LR Rotation):



Rotation Operation - AVL Tree

- Double Rotation:
- Right Left Rotation (RL Rotation):

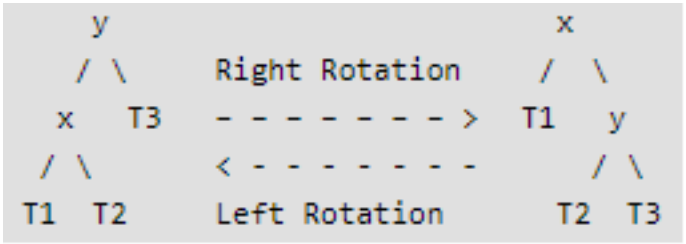


Height Method - AVL Tree

```
// This function computes the height of a tree, Height is the number of nodes
// along the longest path from the root down to the farthest leaf node
int height(node* curr) {
    // base case we reach a null node
    if (curr == NULL)
        return 0;
    // return height of root of this subtree
    return curr->height;
}
```

Right Rotation Method - AVL Tree

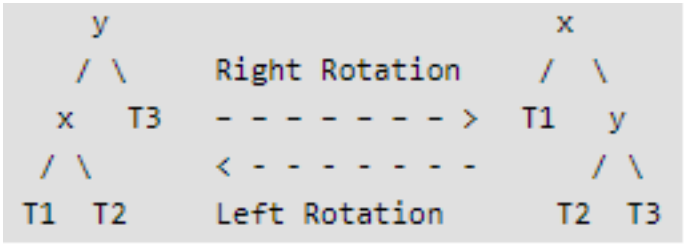
```
// This function computes right rotate subtree rooted with y
node* right_rotate(node* y) {
    // check if the given node and it's left branch are NULLs
    if (y == NULL || y->left == NULL)
        return NULL;
    // constract x and T2 pointers (check slides)
    node* x  = y->left;
    node* T2 = x->right;
    // perform rotation
    x->right = y;
    y->left  = T2;
    // update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    // return new root
    return x;
}
```



AVL-Tree.cpp

Left Rotation Method - AVL Tree

```
// This function computes left rotate subtree rooted with x
node* left_rotate(node* x) {
    // check if the given node and it's left branch are NULLs
    if (x == NULL || x->right == NULL)
        return NULL;
    // constract y and T2 pointers (check slides)
    node* y  = x->right;
    node* T2 = y->left;
    // perform rotation
    y->left  = x;
    x->right = T2;
    // update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;
    // return new root
    return y;
}
```



AVL-Tree.cpp

Lecture Agenda

- ✓ Section 1: Introduction to AVL Tree
- ✓ Section 2: Rotation Operation
- **Section 3: Insertion Operation**
- Section 4: Deletion Operation
- Section 5: Search Operation
- Section 6: Traverse Operation
- Section 7: Time Complexity & Space Complexity

Insertion Operation - AVL Tree

- AVL insertion operates first by inserting the given value the same way as BST insertion and then by applying rebalancing techniques if necessary. The latter is only performed if the AVL property no longer holds, that is the left and right subtrees height differ by more than 1. Each time we insert a node into an AVL tree:

1. We go down the tree to find the correct point at which to insert the node, in the same manner as for BST insertion; then

2. we travel up the tree from the inserted node and check that the node balancing property has not been violated; if the property hasn't been violated then we need not rebalance the tree, the opposite is true if the balancing property has been violated.

- In an AVL tree, the insertion operation is performed with $\Theta(\log n)$ time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows:

- ***Insert Algorithm***:

Insert the new element into the tree using BST insertion logic, then check the Balance Factor

1. ***if left left case then right rotation***
2. ***if right right case then left rotation***
3. ***if left right case then left rotation and right rotation***
4. ***if right left case then right rotation and left rotation***

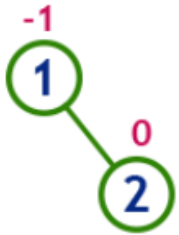
Insertion Operation - AVL Tree (Example 1)

- To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property keys(left) < key(root) < keys(right) by Left Rotation and/or Right Rotation. The Following Examples insert these values in an AVL Tree (1, 2, 3, 4, 5, 6, 7, 8).

- Insert 1

Tree is balanced

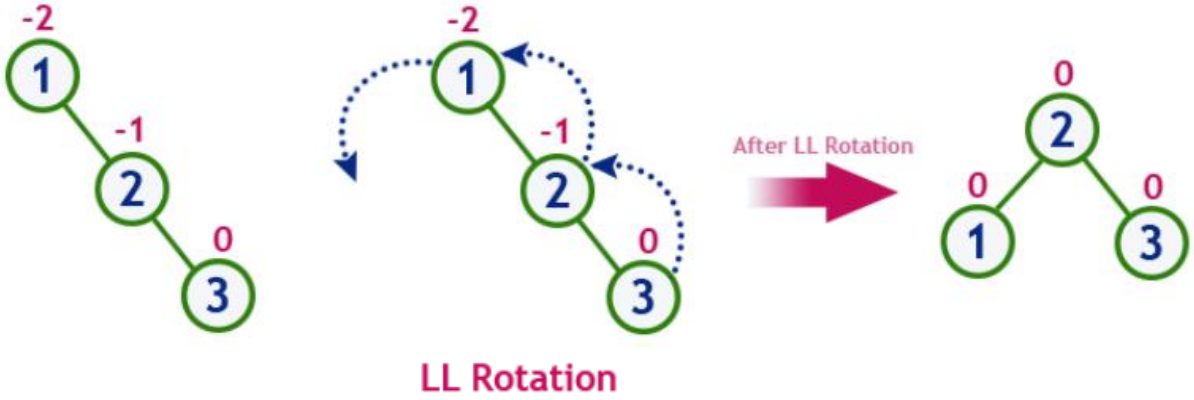
- Insert 2



Tree is balanced

Insertion Operation - AVL Tree (Example 1)

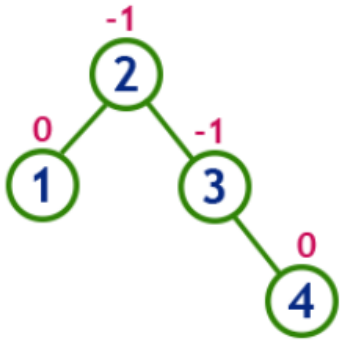
- Insert 3



Tree is imbalanced

Tree is balanced

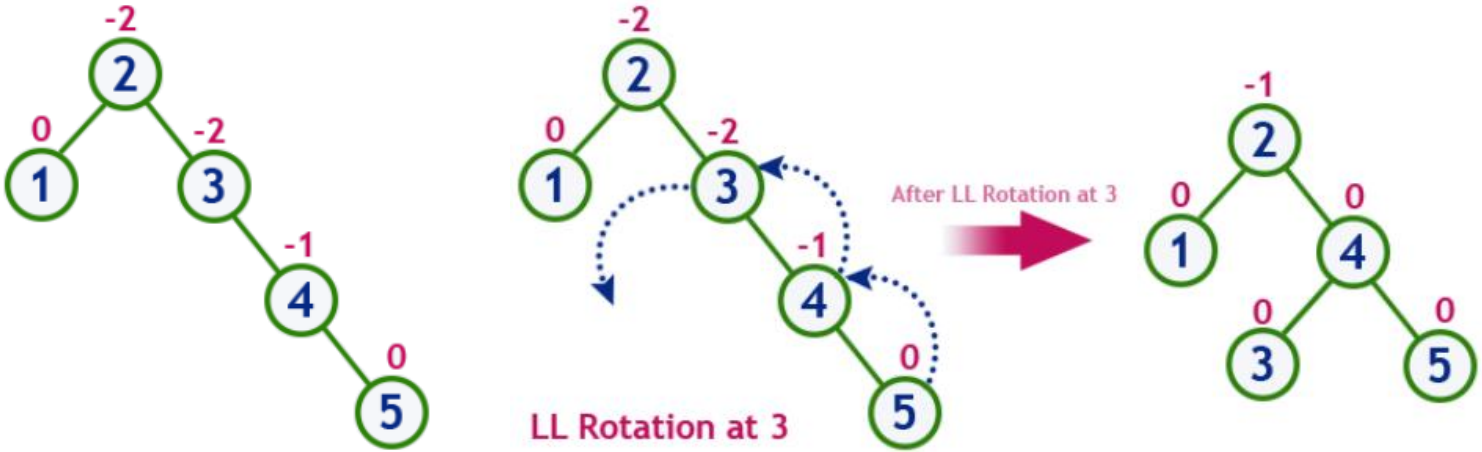
- Insert 4



Tree is balanced

Insertion Operation - AVL Tree (Example 1)

- Insert 5

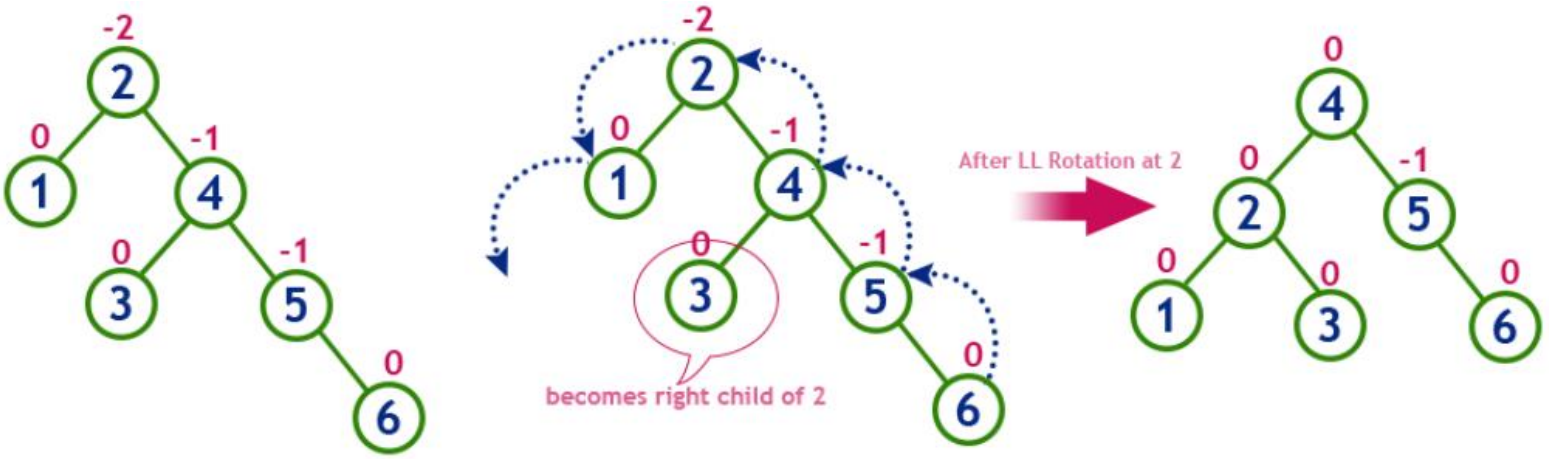


Tree is imbalanced

Tree is balanced

Insertion Operation - AVL Tree (Example 1)

- Insert 6

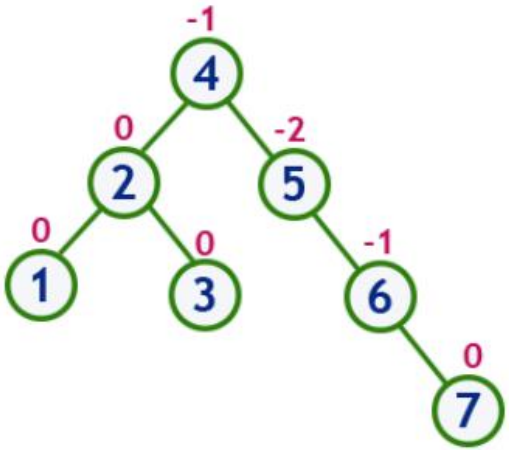


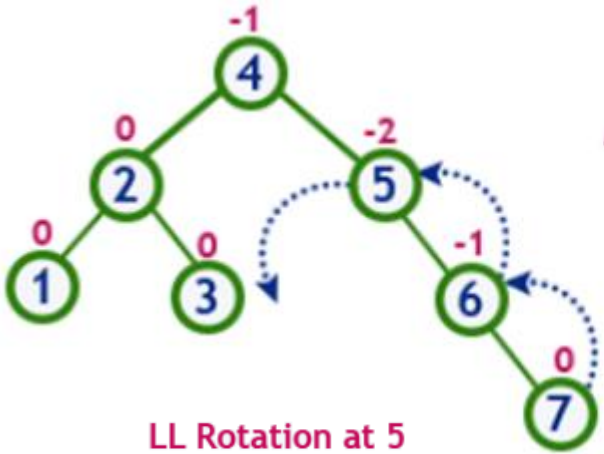
Tree is imbalanced

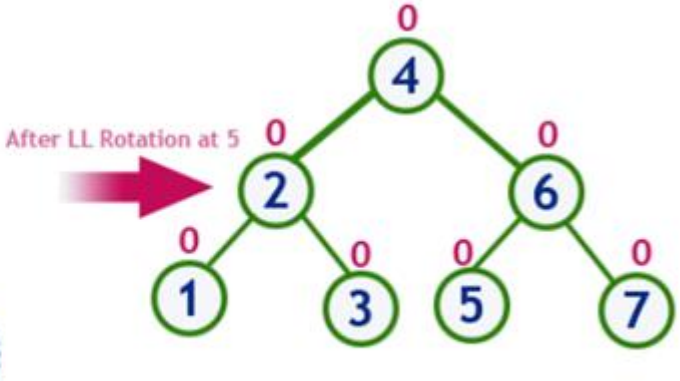
Tree is balanced

Insertion Operation - AVL Tree (Example 1)

- Insert 7





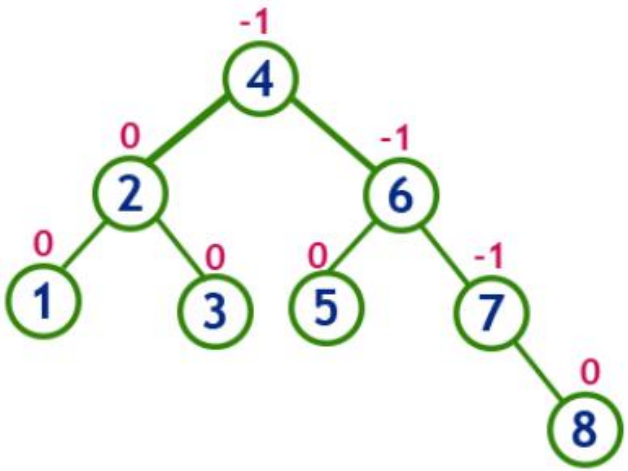


Tree is imbalanced

Tree is balanced

Insertion Operation - AVL Tree (Example 1)

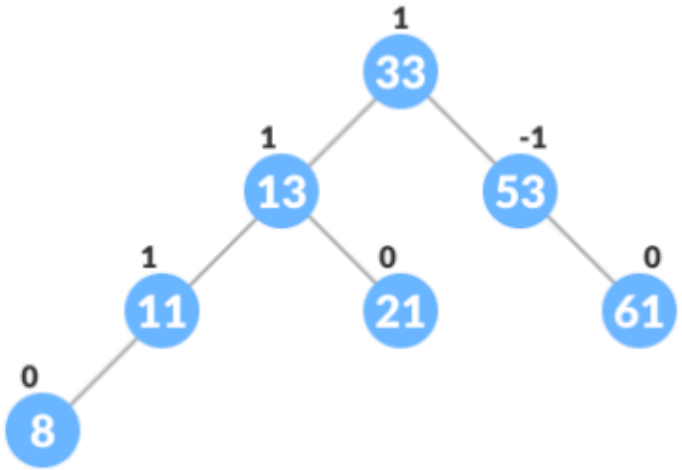
- Insert 8

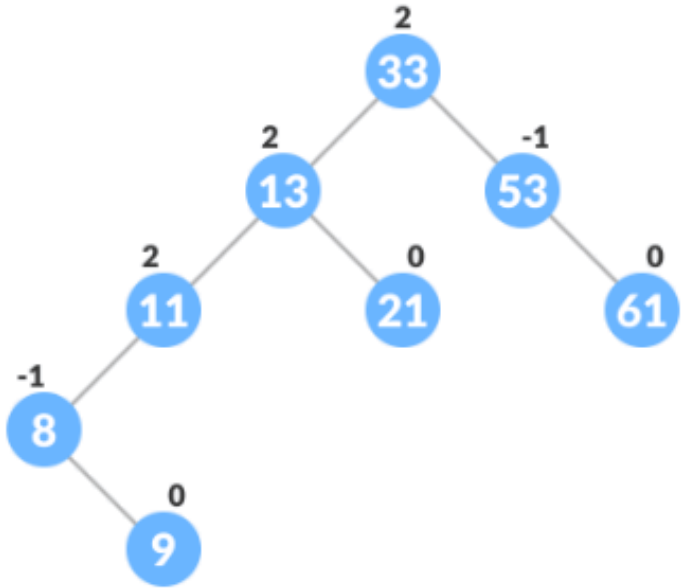


Tree is balanced

Insertion Operation - AVL Tree (Example 2)

- Insert 9

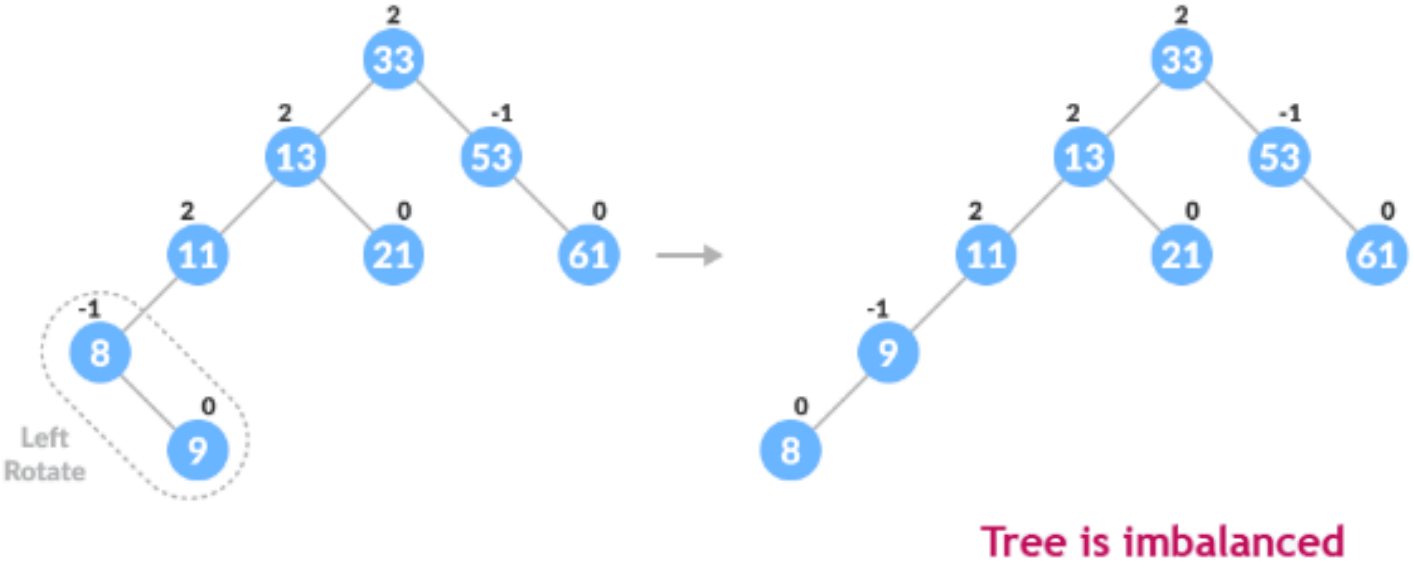




Tree is imbalanced

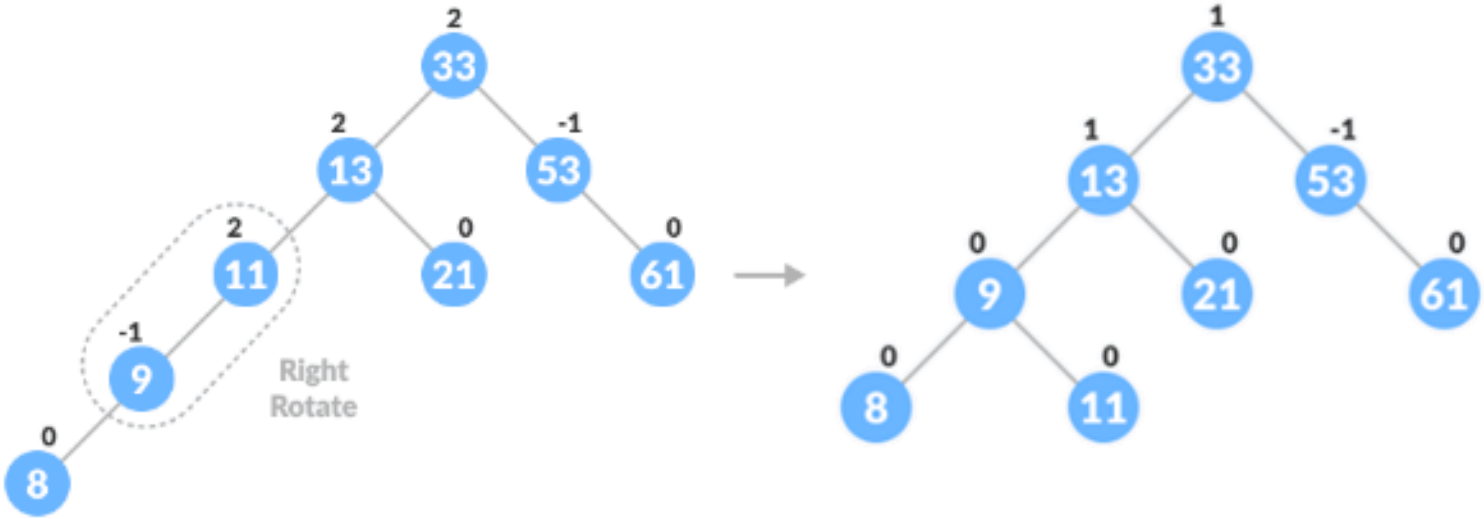
Insertion Operation - AVL Tree (Example 2)

- Insert 9



Insertion Operation - AVL Tree (Example 2)

- Insert 9



Tree is balanced

Balance Factor Method - AVL Tree

```
// This function calculates the balance factor of the given node
int calc_balance_factor(node* curr) {
    // base case we reach a null node
    if (curr == NULL)
        return 0;
    // return the difference between the height of left and right subtrees
    return height(curr->left) - height(curr->right);
}
```

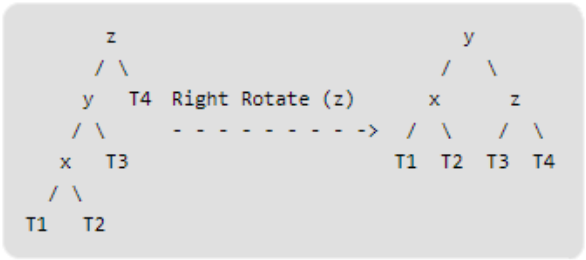
Insertion Operation - AVL Tree

```
// This function inserts a new node with given key in the AVL tree
node* insert_node(node* curr, int new_data) {
    // base case we reach a null node
    if (curr == NULL) {
        node* new_node = new node();
        new_node->data = new_data;
        new_node->height = 1;
        return new_node;
    }
    // repeat the same definition of insert at left or right subtrees
    if (new_data < curr->data)
        curr->left  = insert_node(curr->left, new_data);
    else if (new_data > curr->data)
        curr->right = insert_node(curr->right, new_data);
    // duplicate keys are not allowed in the binary search tree
    else
        return curr;
    // update the height of this curr node
    curr->height = 1 + max(height(curr->left), height(curr->right));
```

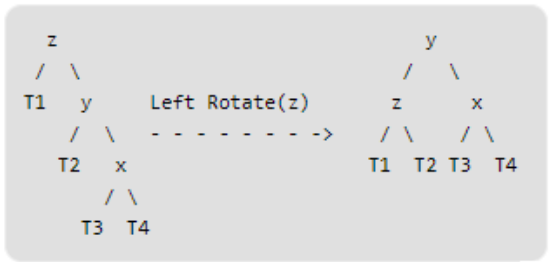
Insertion Operation - AVL Tree

```
// update the height of this curr node
curr->height = 1 + max(height(curr->left), height(curr->right));
// get the balance factor of this curr node
// to check whether this node became unbalanced
int balance = calc_balance_factor(curr);
// If this node becomes unbalanced, then there are 4 cases
// case I: left left case
if (balance > 1 && new_data < curr->left->data)
    return right_rotate(curr);
// case II: right right case
if (balance < -1 && new_data > curr->right->data)
    return left_rotate(curr);
```

Left Left Case



Right Right Case

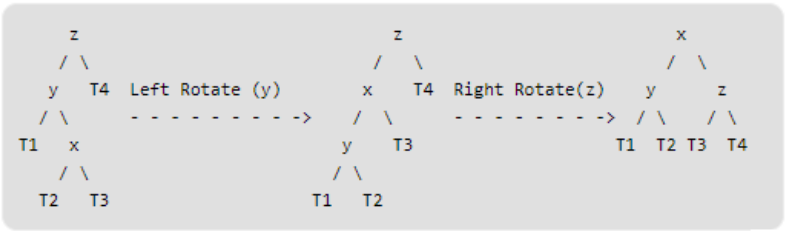


AVL-Tree.cpp

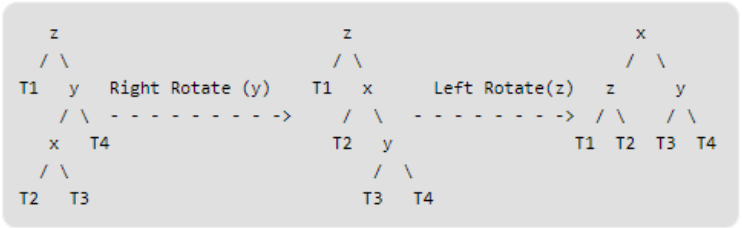
Insertion Operation - AVL Tree

```
    // case III: left right case
    if (balance > 1 && new_data > curr->left->data) {
        curr->left =  left_rotate(curr->left);
        return right_rotate(curr);
    }
    // case IV: right left case
    if (balance < -1 && new_data < curr->right->data) {
        curr->right = right_rotate(curr->right);
        return left_rotate(curr);
    }
    // return the (unchanged) node pointer
    return curr;
}
```

Left Right Case



Right Left Case



AVL-Tree.cpp

Lecture Agenda

- ✔ Section 1: Introduction to AVL Tree
- ✔ Section 2: Rotation Operation
- ✔ Section 3: Insertion Operation
- **Section 4: Deletion Operation**
- Section 5: Search Operation
- Section 6: Traverse Operation
- Section 7: Time Complexity & Space Complexity

Deletion Operation - AVL Tree

- In an AVL Tree, the deletion operation is similar to deletion operation in BST. But after every deletion operation we need to check with the Balance Factor condition. If the tree is balanced after deletion then go for next operation otherwise perform the suitable rotation to make the tree Balanced.
- Our balancing algorithm is like the one presented for our BST. The major difference is that we have to ensure that the tree still adheres to the AVL balance property after the removal of the node. If the tree doesn't need to be rebalanced and the value we are removing is contained within the tree then no further step are required. However, when the value is in the tree and its removal upsets the AVL balance property then we must perform the correct rotation(s).

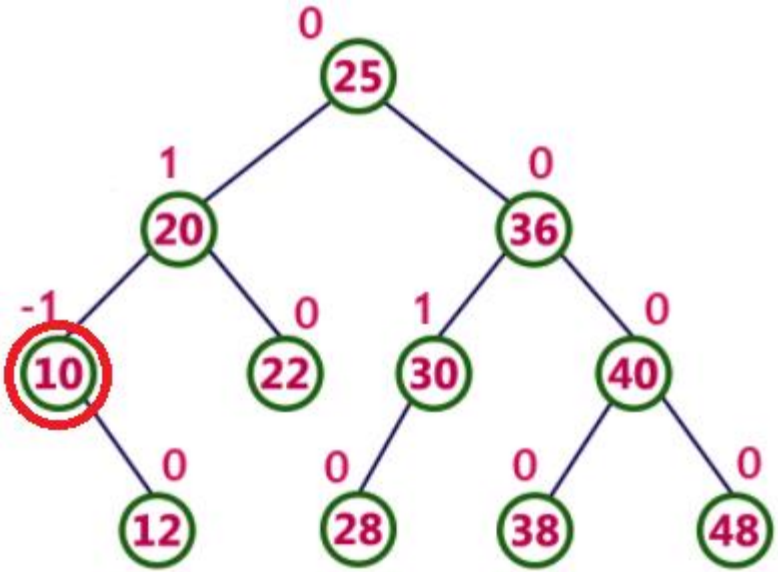
- ***Delete Algorithm***:

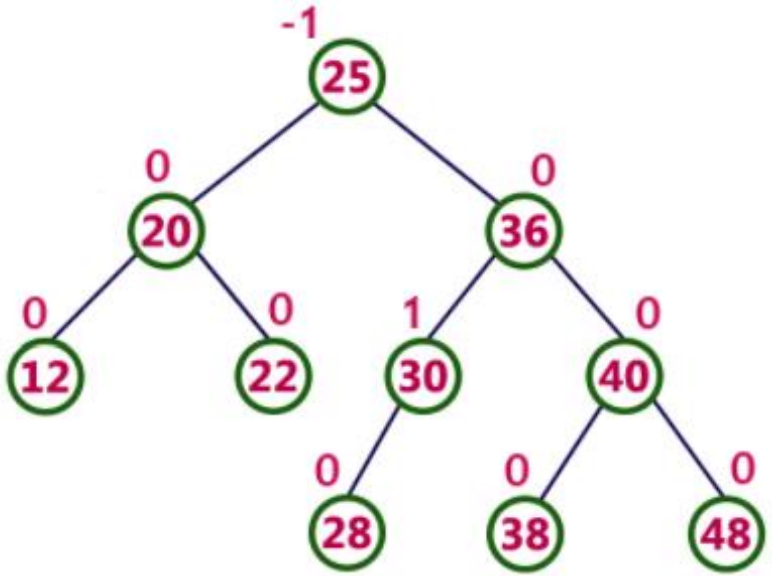
Delete the target element into the tree using BST deletion logic, then check the Balance Factor

1. ***if left left case then right rotation***
2. ***if right right case then left rotation***
3. ***if left right case then left rotation and right rotation***
4. ***if right left case then right rotation and left rotation***

Deletion Operation - AVL Tree

- Delete 10

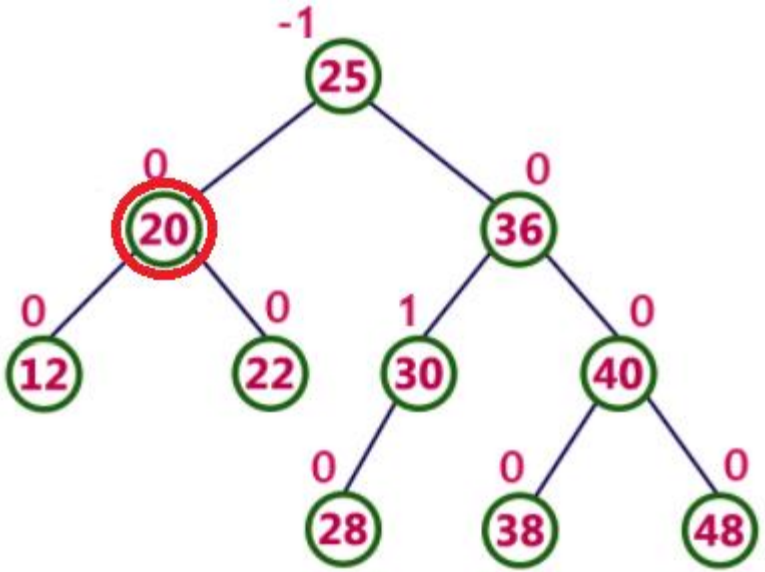


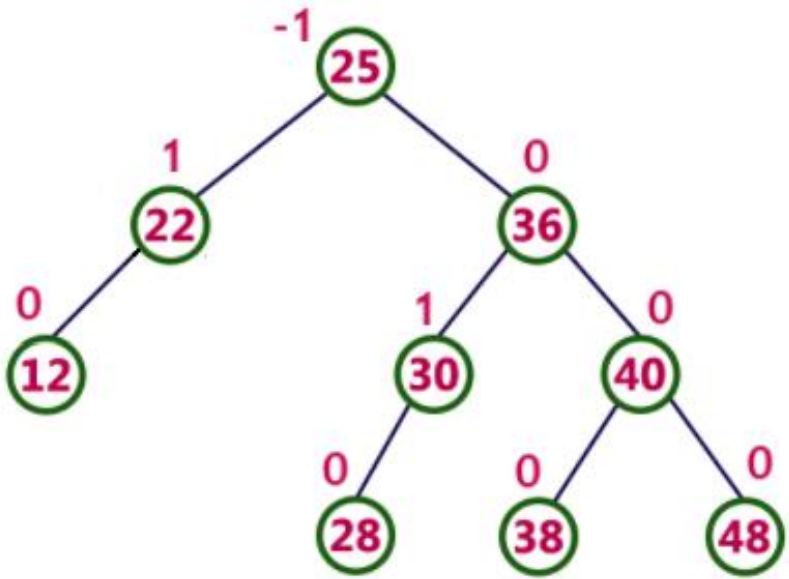


Tree is balanced

Deletion Operation - AVL Tree

- Delete 20

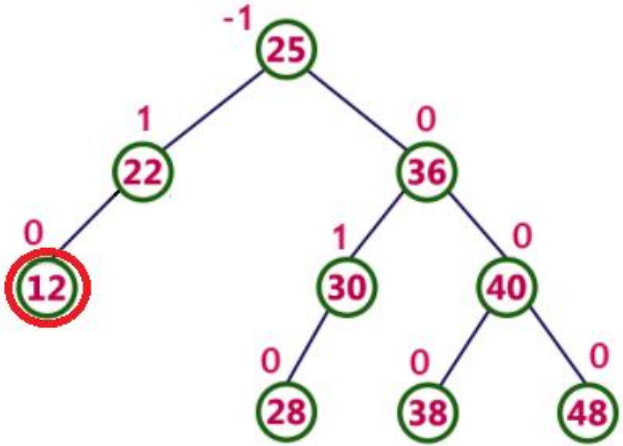


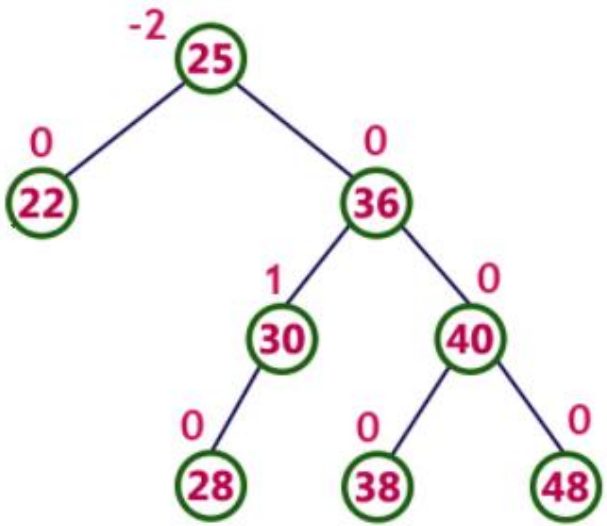


Tree is balanced

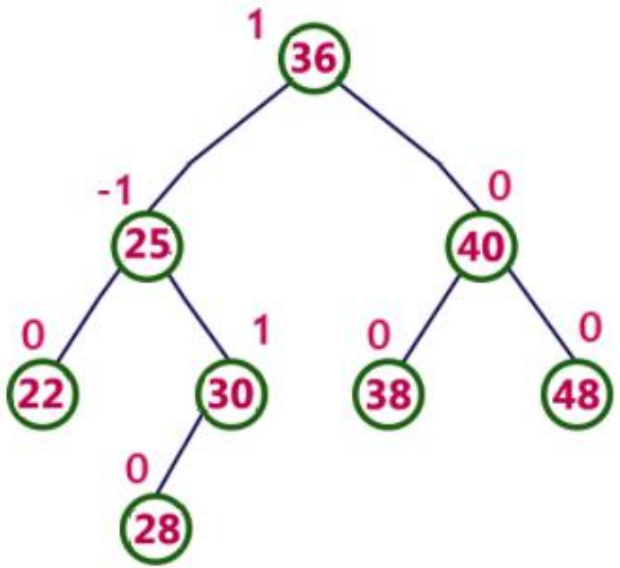
Deletion Operation - AVL Tree

- Delete 12





LL Rotation 25



Tree is imbalanced

Tree is balanced

Min & Max Methods - AVL Tree

```
// This function returns the node with minimum value found in the given tree
node* min_node(node* curr) {
    node* res = curr;
    // loop down to find the leftmost leaf
    while (res->left != NULL)
        res = res->left;
    // return the nearest data node of curr data node
    return res;
}

// This function returns the node with maximum value found in the given tree
node* max_node(node* curr) {
    node* res = curr;
    // loop down to find the rightmost leaf
    while (res->right != NULL)
        res = res->right;
    // return the nearest data node of curr data node
    return res;
}
```

Deletion Operation - AVL Tree

```
// This function deletes the given data in the AVL tree if exists
node* delete_node(node* curr, int data) {
    // base case we reach a null node
    if (curr == NULL)
        return curr;
    // repeat the same definition of insert at left or right subtrees
    if (data < curr->data)
        curr->left = delete_node(curr->left, data);
    else if (data > curr->data)
        curr->right = delete_node(curr->right, data);
    // if the given data is same as curr's data, then we will delete this node
    else {
        // node with no child
        if (curr->left == NULL && curr->right == NULL) {
            node* temp = curr;
            curr = NULL;
            delete(temp);
        }
```

Deletion Operation - AVL Tree

```
else {
    // node with no child
    if (curr->left == NULL && curr->right == NULL) {
        node* temp = curr;
        curr = NULL;
        delete(temp);
    }
    // node with only one child
    else if (curr->left == NULL) {
        node* temp = curr->right;
        curr = temp;
        delete(temp);
    }
    // node with only one child
    else if (curr->right == NULL) {
        node* temp = curr->left;
        curr = temp;
        delete(temp);
    }
```

Deletion Operation - AVL Tree

```
        // node with two children
        else {
            // get the inorder successor (smallest in the right subtree)
            node* temp = min_node(curr->right);
            // copy the inorder successor's content to this node
            curr->data = temp->data;
            // delete the inorder successor
            curr->right = delete_node(curr->right, temp->data);
        }
    }
    // return curr node if the tree had only one node
    if (curr == NULL)
        return curr;
    // update the height of this curr node
    curr->height = 1 + max(height(curr->left), height(curr->right));
    // get the balance factor of this curr node
    // to check whether this node became unbalanced
    int balance = calc_balance_factor(curr);
```

Deletion Operation - AVL Tree

```
    // If this node becomes unbalanced, then there are 4 cases
    // case I: left left case
    if (balance > 1 && calc_balance_factor(curr->left) >= 0)
        return right_rotate(curr);
    // case II: right right case
    if (balance < -1 && calc_balance_factor(curr->right) <= 0)
        return left_rotate(curr);
    // case III: left right case
    if (balance > 1 && calc_balance_factor(curr->left) < 0) {
        curr->left =  left_rotate(curr->left);
        return right_rotate(curr);
    }
    // case IV: right left case
    if (balance < -1 && calc_balance_factor(curr->right) > 0) {
        curr->right = right_rotate(curr->right);
        return left_rotate(curr);
    }
    // return the (unchanged) node pointer
    return curr;
}
```

AVL-Tree.cpp

Lecture Agenda

- ✔ Section 1: Introduction to AVL Tree
- ✔ Section 2: Rotation Operation
- ✔ Section 3: Insertion Operation
- ✔ Section 4: Deletion Operation
- **Section 5: Search Operation**
- Section 6: Traverse Operation
- Section 7: Time Complexity & Space Complexity

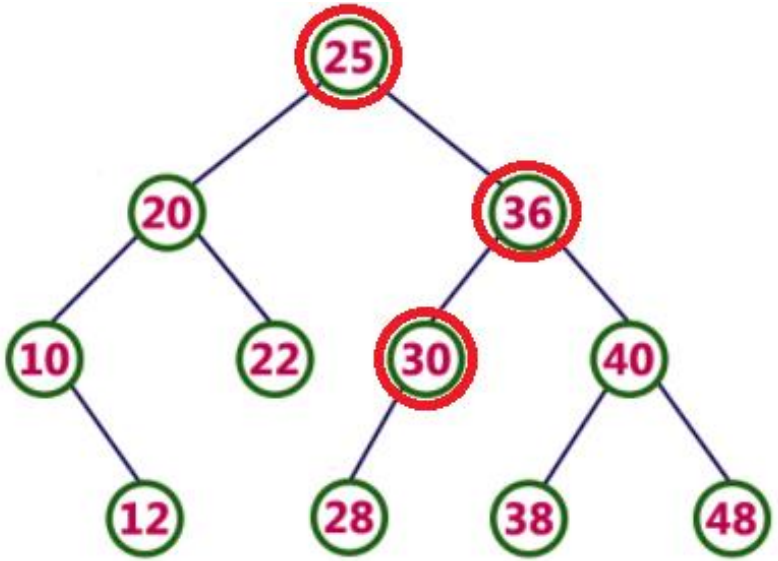
Search Operation - AVL Tree

- Searching a BST is even simpler than insertion. The pseudo-code is self-explanatory but we will look briefly at the premise of the algorithm nonetheless. We have talked previously about insertion, we go either left or right with the right subtree containing values that are › x where x is the value of the node we are inserting. When searching the rules are made a little more atomic and at any one time we have four cases to consider:

- ***Search Algorithm***:

1. ***if curr == NULL then return false***
2. ***if curr == data then return true***
3. ***if data < curr then***
4. ***go to step 1 with left node of curr***
5. ***if data > curr then***
6. ***go to step 1 with right node of curr***

- Example Searching for 30.



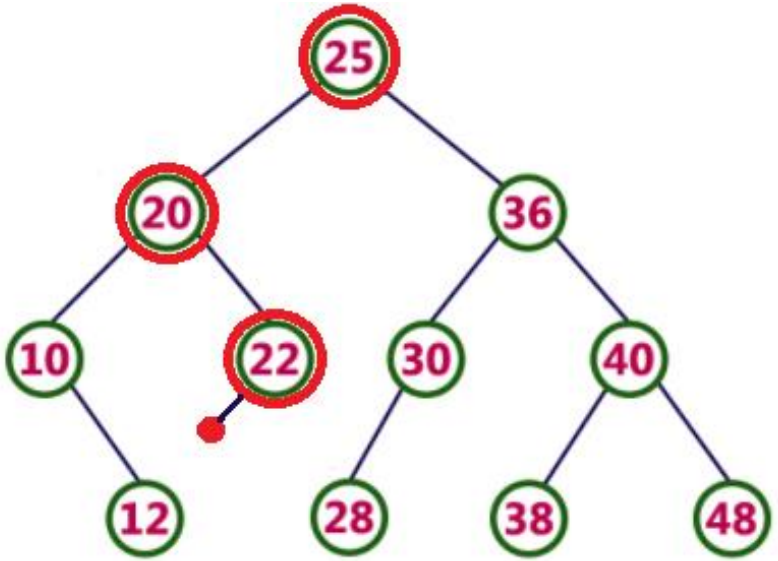
Search Operation - AVL Tree

- Searching a BST is even simpler than insertion. The pseudo-code is self-explanatory but we will look briefly at the premise of the algorithm nonetheless. We have talked previously about insertion, we go either left or right with the right subtree containing values that are › x where x is the value of the node we are inserting. When searching the rules are made a little more atomic and at any one time we have four cases to consider:

- ***Search Algorithm***:

1. ***if curr == NULL then return false***
2. ***if curr == data then return true***
3. ***if data < curr then***
4. ***go to step 1 with left node of curr***
5. ***if data > curr then***
6. ***go to step 1 with right node of curr***

- Example Searching for 21.



Search Operation - AVL Tree

```
// This function searches if the given data in the AVL tree
bool search(node* curr, int data) {
    // base case we reach a null node
    if (curr == NULL)
        return false;
    // check if we find the data at the curr node
    if (curr->data == data)
        return true;
    // repeat the same definition of search at left or right subtrees
    if (data < curr->data)
        return search(curr->left, data);
    else
        return search(curr->right, data);
}
```

Lecture Agenda

- ✔ Section 1: Introduction to AVL Tree
- ✔ Section 2: Rotation Operation
- ✔ Section 3: Insertion Operation
- ✔ Section 4: Deletion Operation
- ✔ Section 5: Search Operation
- **Section 6: Traverse Operation**
- Section 7: Time Complexity & Space Complexity

Traverse Operation - AVL Tree

- There are various strategies which can be employed to traverse the items in a tree. the choice of strategy depends on which node visitation order you require. In this section we will touch on the traversals that DSA provides on all data structures that derive from Binary Search Tree.
- Preorder: When using this algorithm, you visit the root first, then traverse the left subtree and finally traverse the right subtree.
- Inorder: When using this algorithm, you traverse the left subtree first, then visit the root and finally traverse the right subtree.
- Postorder: When using this algorithm, you traverse the left subtree first, then traverse the right subtree and finally visit the root.
- Breadth First: Traversing a tree in breadth first order yields the values of all nodes of a particular depth in the tree before any deeper ones. In other words, given a depth d we would visit the values of all nodes at d in a left to right fashion, then we would proceed to d + 1 and so on until we had no more nodes to visit. Traditionally breadth first traversal is implemented using a list (vector, resizable array, etc) to store the values of the nodes visited in breadth first order and then a queue to store those nodes that have yet to be visited.

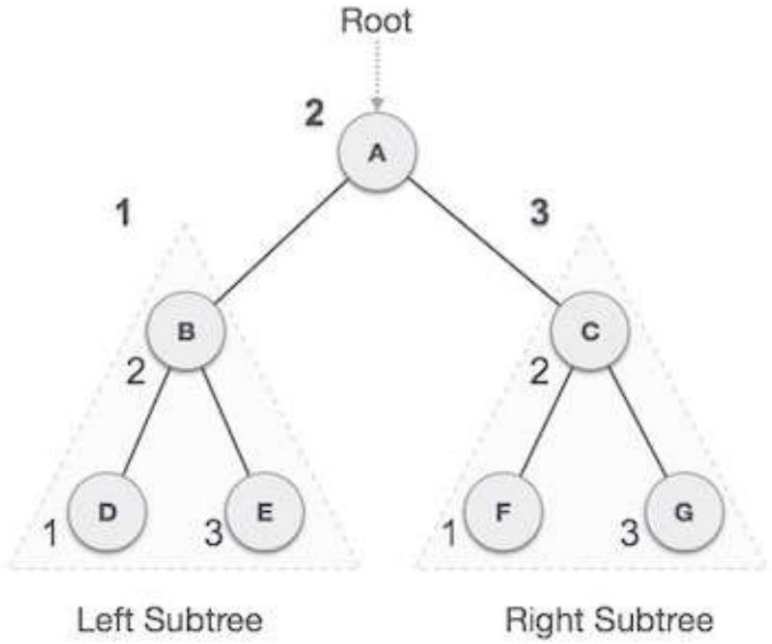
Traverse Operation - AVL Tree (in-order)

- In-order Traversal is the one of DFS Traversal of the tree. Therefore, we will start from the root node of the tree and go deeper-and-deeper into the left subtree with recursive manner. When we will reach to the left-most node with the above steps, then we will visit that current node and go to the left-most node of its right subtree (if exists).

- ***Traverse Algorithm***:

1. ***if curr == NULL then return***
2. ***go to step 1 with left node of curr***
3. ***print curr node data***
4. ***go to step 1 with right node of curr***

```
void inOrder(node* curr) {
    // base case we reach a null node
    if (curr == NULL)
        return;
    // repeat the same definition
    inOrder(curr->left);
    cout << curr->data << ' ';
    inOrder(curr->right);
}
```



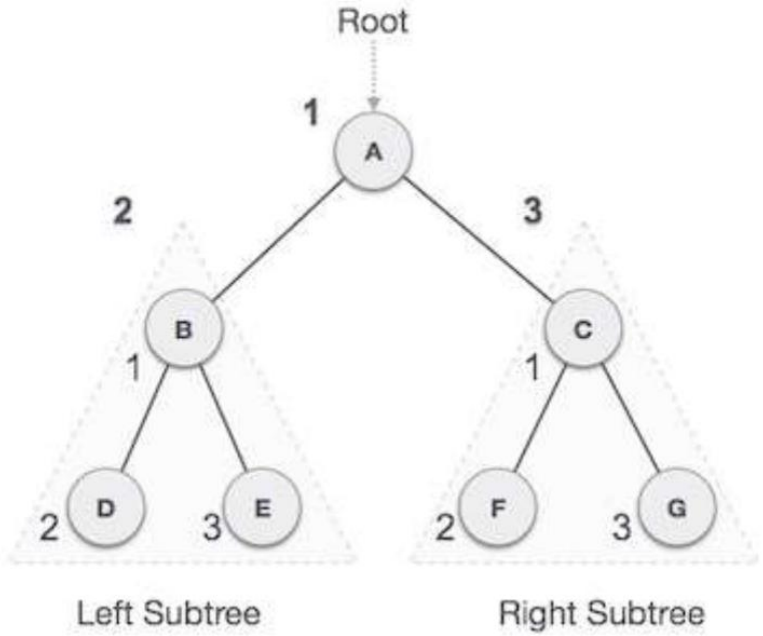
Traverse Operation - AVL Tree (pre-order)

- **Pre-order Traversal:** is the one of DFS Traversal of the tree. Therefore, we visit the current node first and then goes to the left sub-tree. After covering every node of the left sub-tree, we will move towards the right sub-tree and visit in a similar fashion.

- ***Traverse Algorithm***:

1. ***if curr == NULL then return***
2. ***print curr node data***
3. ***go to step 1 with left node of curr***
4. ***go to step 1 with right node of curr***

```
void preOrder(node* curr) {
    // base case we reach a null node
    if (curr == NULL)
        return;
    // repeat the same definition
    cout << curr->data << ' ';
    preOrder(curr->left);
    preOrder(curr->right);
}
```



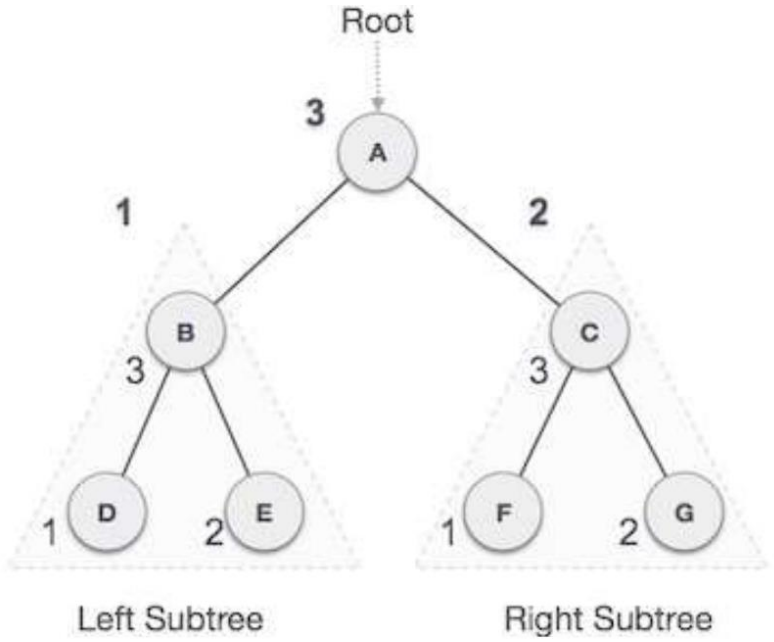
Traverse Operation - AVL Tree (post-order)

- Post-order Traversal: is the one of DFS Traversal of the tree. Therefore, we will start from the root node of the tree and go deeper-and-deeper into the left subtree with recursive manner. When we will reach to the left-most node with the above steps, then we will go to the left-most node of its right subtree (if exists) and finally we will visit that current node.

- ***Traverse Algorithm***:

1. ***if curr == NULL then return***
2. ***go to step 1 with left node of curr***
3. ***go to step 1 with right node of curr***
4. ***print curr node data***

```
void postOrder(node* curr) {
    // base case we reach a null node
    if (curr == NULL)
        return;
    // repeat the same definition
    postOrder(curr->left);
    postOrder(curr->right);
    cout << curr->data << ' ';
}
```



Functionality Testing - AVL Tree

- Initialize a global struct

```cpp
#include <bits/stdc++.h>
using namespace std;

// AVL tree node
struct node {
    int data;
    node* left;
    node* right;
    int height;
};

// Initialize a global pointer for root
node* root;
```

Functionality Testing - AVL Tree

- In the Main function:

```
root = insert_node(root, 350);
cout << "AVL Tree in-order traversal   : ";
inOrder(root);
cout << "\n";
cout << "AVL Tree pre-order traversal  : ";
preOrder(root);
cout << "\n";
cout << "AVL Tree post-order traversal : ";
postOrder(root);
```

- New Tree Diagram

- Expected Output:

```
AVL Tree in-order   : 350
AVL Tree pre-order  : 350
AVL Tree post-order : 350
```

AVL-Tree.cpp

Functionality Testing - AVL Tree

- In the Main function:

```
root = insert_node(root, 300);
cout << "AVL Tree in-order traversal   : ";
inOrder(root);
cout << "\n";
cout << "AVL Tree pre-order traversal  : ";
preOrder(root);
cout << "\n";
cout << "AVL Tree post-order traversal : ";
postOrder(root);
```

- Expected Output:

```
AVL Tree in-order   : 300 350
AVL Tree pre-order  : 350 300
AVL Tree post-order : 300 350
```

- Current Tree Diagram

```
350
```

- New Tree Diagram

```
    350
   /
300
```

Functionality Testing - AVL Tree

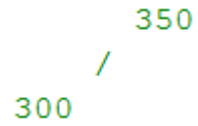
- In the Main function:

```
root = insert_node(root, 250);
cout << "AVL Tree in-order traversal   : ";
inOrder(root);
cout << "\n";
cout << "AVL Tree pre-order traversal  : ";
preOrder(root);
cout << "\n";
cout << "AVL Tree post-order traversal : ";
postOrder(root);
```

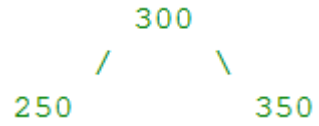
- Expected Output:

```
AVL Tree in-order   : 250 300 350
AVL Tree pre-order  : 300 250 350
AVL Tree post-order : 250 350 300
```

- Current Tree Diagram



- New Tree Diagram



Functionality Testing - AVL Tree

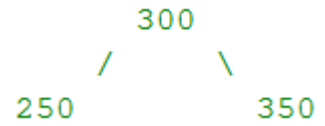
- In the Main function:

```
root = insert_node(root, 200);
cout << "AVL Tree in-order traversal   : ";
inOrder(root);
cout << "\n";
cout << "AVL Tree pre-order traversal  : ";
preOrder(root);
cout << "\n";
cout << "AVL Tree post-order traversal : ";
postOrder(root);
```

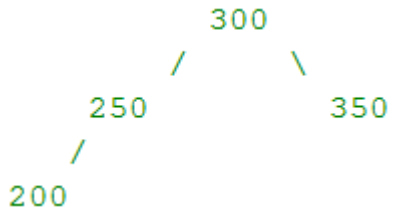
- Expected Output:

```
AVL Tree in-order   : 200 250 300 350
AVL Tree pre-order  : 300 250 200 350
AVL Tree post-order : 200 250 350 300
```

- Current Tree Diagram



- New Tree Diagram



AVL-Tree.cpp

Functionality Testing - AVL Tree

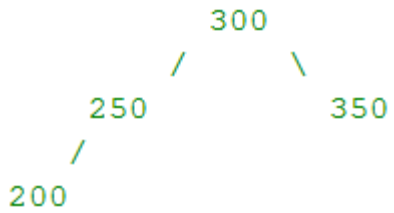
- In the Main function:

```
root = insert_node(root, 150);
cout << "AVL Tree in-order traversal   : ";
inOrder(root);
cout << "\n";
cout << "AVL Tree pre-order traversal  : ";
preOrder(root);
cout << "\n";
cout << "AVL Tree post-order traversal : ";
postOrder(root);
```

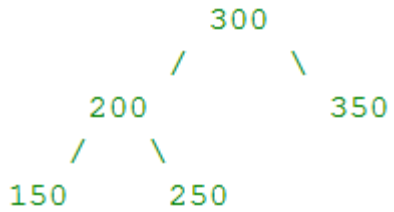
- Expected Output:

```
AVL Tree in-order   : 150 200 250 300 350
AVL Tree pre-order  : 300 200 150 250 350
AVL Tree post-order : 150 250 200 350 300
```

- Current Tree Diagram



- New Tree Diagram



AVL-Tree.cpp

Functionality Testing - AVL Tree

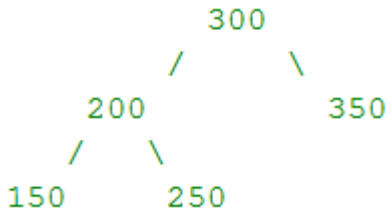
- In the Main function:

```
root = insert_node(root, 100);
cout << "AVL Tree in-order traversal   : ";
inOrder(root);
cout << "\n";
cout << "AVL Tree pre-order traversal  : ";
preOrder(root);
cout << "\n";
cout << "AVL Tree post-order traversal : ";
postOrder(root);
```

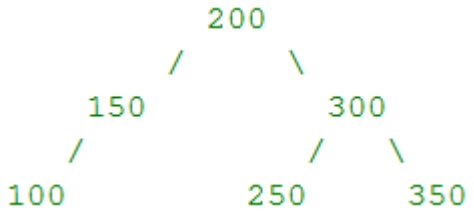
- Expected Output:

```
AVL Tree in-order   : 100 150 200 250 300 350
AVL Tree pre-order  : 200 150 100 300 250 350
AVL Tree post-order : 100 150 250 350 300 200
```

- Current Tree Diagram



- New Tree Diagram



AVL-Tree.cpp

Functionality Testing - AVL Tree

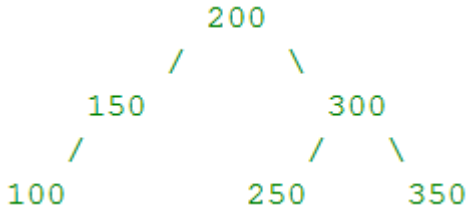
- In the Main function:

```
root = insert_node(root, 50);
cout << "AVL Tree in-order traversal   : ";
inOrder(root);
cout << "\n";
cout << "AVL Tree pre-order traversal  : ";
preOrder(root);
cout << "\n";
cout << "AVL Tree post-order traversal : ";
postOrder(root);
```

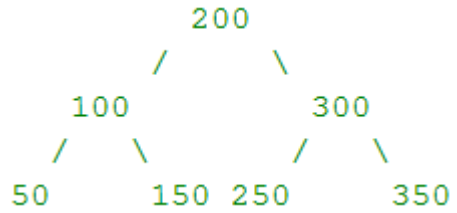
- Expected Output:

```
AVL Tree in-order   : 50 100 150 200 250 300 350
AVL Tree pre-order  : 200 100 50 150 300 250 350
AVL Tree post-order : 50 150 100 250 350 300 200
```

- Current Tree Diagram



- New Tree Diagram



AVL-Tree.cpp

Functionality Testing - AVL Tree

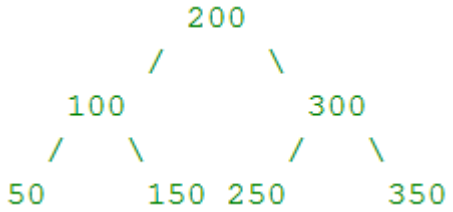
- In the Main function:

```
root = insert_node(root, 40);
cout << "AVL Tree in-order traversal   : ";
inOrder(root);
cout << "\n";
cout << "AVL Tree pre-order traversal  : ";
preOrder(root);
cout << "\n";
cout << "AVL Tree post-order traversal : ";
postOrder(root);
```

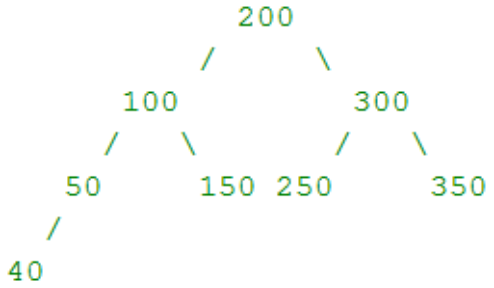
- Expected Output:

```
AVL Tree in-order   : 40 50 100 150 200 250 300 350
AVL Tree pre-order  : 200 100 50 40 150 300 250 350
AVL Tree post-order : 40 50 150 100 250 350 300 200
```

- Current Tree Diagram



- New Tree Diagram



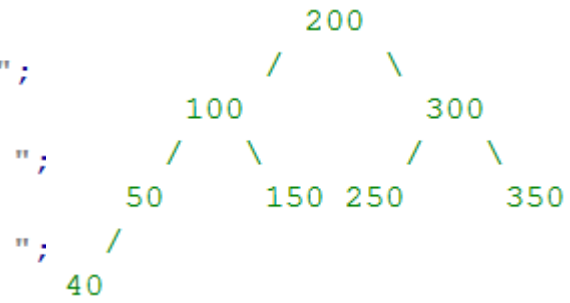
AVL-Tree.cpp

Functionality Testing - AVL Tree

- In the Main function:

```
cout << "The minimum value in the tree of root 200 is: ";
cout << min_node(root)->data << '\n';
cout << "The maximum value in the tree of root 200 is: ";
cout << max_node(root)->data << '\n';
cout << "The minimum value in the tree of root 100  is: ";
cout << min_node(root->left)->data << '\n';
cout << "The maximum value in the tree of root 100  is: ";
cout << max_node(root->left)->data << '\n';
cout << "The minimum value in the tree of root 300 is: ";
cout << min_node(root->right)->data << '\n';
cout << "The maximum value in the tree of root 300 is: ";
cout << max_node(root->right)->data << '\n';
```

- Current Tree Diagram



- Expected Output:

```
The minimum value in the tree of root 200 is: 40
The maximum value in the tree of root 200 is: 350
The minimum value in the tree of root 100 is: 40
The maximum value in the tree of root 100 is: 150
The minimum value in the tree of root 300 is: 250
The maximum value in the tree of root 300 is: 350
```

AVL-Tree.cpp

Functionality Testing - AVL Tree

➢ In the Main function:

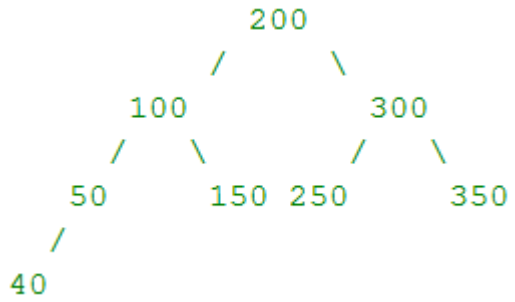
```
if (search(root, 150))
    cout << "element " << 150 << " in the tree\n";
else
    cout << "element " << 150 << " not in the tree\n";

if (search(root, 100))
    cout << "element " << 100 << " in the tree\n";
else
    cout << "element " << 100 << " not in the tree\n";

if (search(root, 75))
    cout << "element " << 75 << " in the tree\n";
else
    cout << "element " << 75 << " not in the tree\n";

if (search(root, 95))
    cout << "element " << 95 << " in the tree\n";
else
    cout << "element " << 95 << " not in the tree\n";
```

➢ Current Tree Diagram



➢ Expected Output:

```
element 150 in the tree
element 100 in the tree
element 75 not in the tree
element 95 not in the tree
```

AVL-Tree.cpp

Functionality Testing - AVL Tree

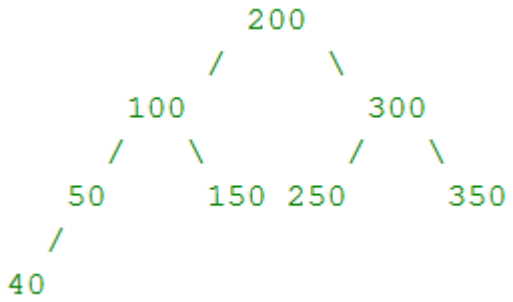
- In the Main function:

```
root = delete_node(root, 300);
cout << "AVL Tree in-order traversal   : ";
inOrder(root);
cout << "\n";
cout << "AVL Tree pre-order traversal  : ";
preOrder(root);
cout << "\n";
cout << "AVL Tree post-order traversal : ";
postOrder(root);
```

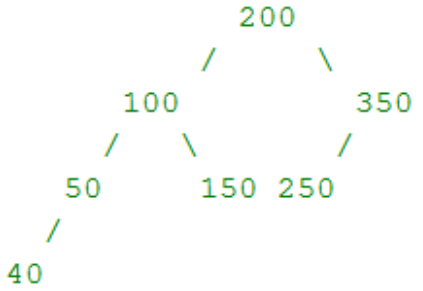
- Expected Output:

```
AVL Tree in-order   : 40 50 100 150 200 250 350
AVL Tree pre-order  : 200 100 50 40 150 350 250
AVL Tree post-order : 40 50 150 100 250 350 200
```

- Current Tree Diagram



- New Tree Diagram



AVL-Tree.cpp

Functionality Testing - AVL Tree

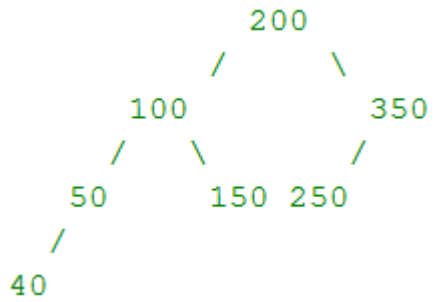
- In the Main function:

```
root = delete_node(root, 250);
cout << "AVL Tree in-order traversal   : ";
inOrder(root);
cout << "\n";
cout << "AVL Tree pre-order traversal  : ";
preOrder(root);
cout << "\n";
cout << "AVL Tree post-order traversal : ";
postOrder(root);
```

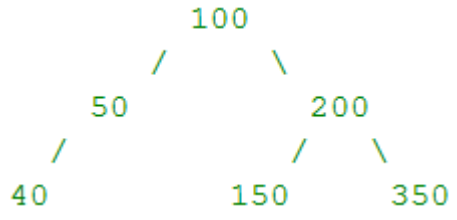
- Expected Output:

```
AVL Tree in-order   : 40 50 100 150 200 350
AVL Tree pre-order  : 100 50 40 200 150 350
AVL Tree post-order : 40 50 150 350 200 100
```

- Current Tree Diagram



- New Tree Diagram



AVL-Tree.cpp

Lecture Agenda

- ✔ Section 1: Introduction to AVL Tree
- ✔ Section 2: Rotation Operation
- ✔ Section 3: Insertion Operation
- ✔ Section 4: Deletion Operation
- ✔ Section 5: Search Operation
- ✔ Section 6: Traverse Operation

Section 7: Time Complexity & Space Complexity

Time Complexity & Space Complexity

➢ Time Analysis

	Worst Case	Average Case
• Balance Factor	Θ(1)	θ(1)
• Rotation	Θ(1)	θ(1)
• Insert	Θ(log n)	θ(log n)
• Delete	Θ(log n)	θ(log n)
• Search	Θ(log n)	θ(log n)
• Traverse	Θ(n)	θ(n)

Lecture Agenda

- ✓ Section 1: Introduction to AVL Tree
- ✓ Section 2: Rotation Operation
- ✓ Section 3: Insertion Operation
- ✓ Section 4: Deletion Operation
- ✓ Section 5: Search Operation
- ✓ Section 6: Traverse Operation
- ✓ Section 7: Time Complexity & Space Complexity

Assignment

Implement STL Set

- Sets are containers that store unique elements following a specific order.
- In a set, the value of an element also identifies it (the value is itself the key, of type T), and each value must be unique. The value of the elements in a set cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.
- Sets are a type of associative containers in which each element has to be unique, because the value of the element identifies it. The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.

More Info: cplusplus.com/reference/set/set/
More Info: geeksforgeeks.org/set-in-cpp-stl/

Implement STL Set

- Member functions:
  - (constructor) Construct vector (public member function)
  - (destructor) Vector destructor (public member function)
  - (operator=) Assign content (public member function)
- Iterators:
  - (begin) Return iterator to beginning (public member function)
  - (end) Return iterator to end (public member function)
  - (rbegin) Return reverse iterator to reverse beginning (public member function)
  - (rend) Return reverse iterator to reverse end (public member function)
  - (cbegin) Return const_iterator to beginning (public member function)
  - (cend) Return const_iterator to end (public member function)
  - (crbegin) Return const_reverse_iterator to reverse beginning (public member function)
  - (crend) Return const_reverse_iterator to reverse end (public member function)

Implement STL Set

- Capacity: (empty) Test whether vector is empty (public member function)
  (size) Return size (public member function)
  (max_size) Return maximum size (public member function)
- Modifiers: (insert) Insert element (public member function)
  (erase) Erase elements (public member function)
  (swap) Swap content (public member function)
  (clear) Clear content (public member function)
- Operations: (find) Get iterator to element (public member function)
  (count) Count elements with a specific value (public member function)
  (lower_bound) Return iterator to lower bound (public member function)
  (upper_bound) Return iterator to upper bound (public member function)
  (equal_range) Get range of equal elements (public member function)

More Info: cplusplus.com/reference/set/set/

Implement STL Map

- Maps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order.
- In a map, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this key. The types of key and mapped value may differ, and are grouped together in member type value_type, which is a pair type combining both.
- The mapped values in a map can be accessed directly by their corresponding key using the bracket operator ((operator[]).
- Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have same key values.

More Info: cplusplus.com/reference/map/map/
More Info: geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/

Implement STL Map

- Member functions: (constructor) Construct vector (public member function)
  (destructor) Vector destructor (public member function)
  (operator=) Assign content (public member function)
- Iterators: (begin) Return iterator to beginning (public member function)
  (end) Return iterator to end (public member function)
  (rbegin) Return reverse iterator to reverse beginning (public member function)
  (rend) Return reverse iterator to reverse end (public member function)
  (cbegin) Return const_iterator to beginning (public member function)
  (cend) Return const_iterator to end (public member function)
  (crbegin) Return const_reverse_iterator to reverse beginning (public member function)
  (crend) Return const_reverse_iterator to reverse end (public member function)

Implement STL Map

- Capacity: (empty) Test whether vector is empty (public member function)
  (size) Return size (public member function)
  (max_size) Return maximum size (public member function)
- Modifiers: (insert) Insert element (public member function)
  (erase) Erase elements (public member function)
  (swap) Swap content (public member function)
  (clear) Clear content (public member function)
- Operations: (find) Get iterator to element (public member function)
  (count) Count elements with a specific value (public member function)
  (lower_bound) Return iterator to lower bound (public member function)
  (upper_bound) Return iterator to upper bound (public member function)
  (equal_range) Get range of equal elements (public member function)

More Info: cplusplus.com/reference/map/map/


DO
MORE.