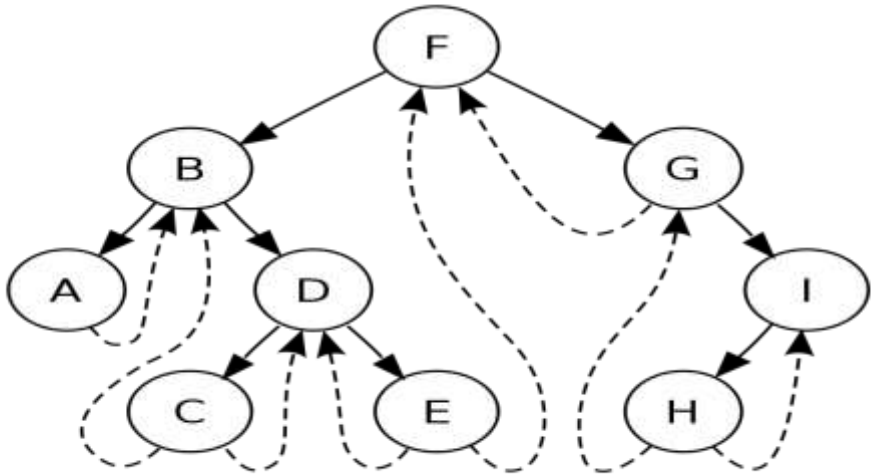
Threaded Binary Tree

Threads

- In linked representation of Binary tree,
- **Most of the nodes have NULL value in their left and right pointers.**
- It would be good to **use these pointer fields to keep some other information.**
- Useful for Traversal Operation

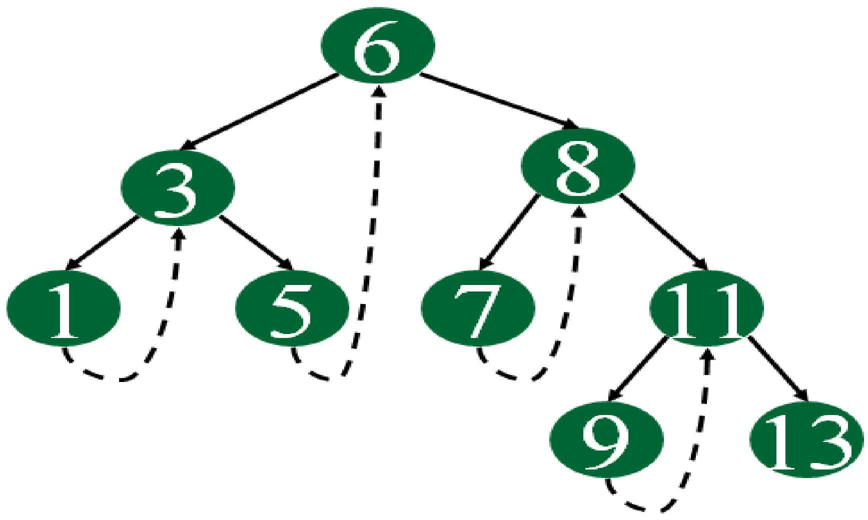
Threads

- **These NULL pointers can be used to point nodes higher in the tree.**
- Such pointers are called Threads



Threaded Binary Tree

- A binary tree which implements such pointers is called threaded Binary tree



Single Threaded Binary Tree

The dotted lines represent threads

Threads

Threads corresponding to 3 Traversals:

- **Preorder**
- **Postorder**
- **Inorder**

Inorder Threading

Two types-

- **One Way Inorder Threading/ Single Threaded**
- **Two Way Inorder Threading / Double Threaded**

Inorder Threading-

One Way Inorder Threading

- Each NULL right pointer is altered to contain a thread to point to that node's inorder successor.

Two Way Inorder Threading

- Each NULL right pointer is altered to contain a thread to point to that node's inorder successor.

+

- Each NULL left pointer is altered to contain a thread to point to that node's inorder predecessor.

Inorder Threading

Two types-

- **Right In-Threaded Binary Tree**
- **Left In-Threaded Binary Tree**
- **Fully In-Threaded Binary Tree**

Inorder Threading-

Right In-Threaded Binary Tree

- If we use right field of the node to take the thread

Left In-Threaded Binary Tree

- If we use left field of the node to take the thread

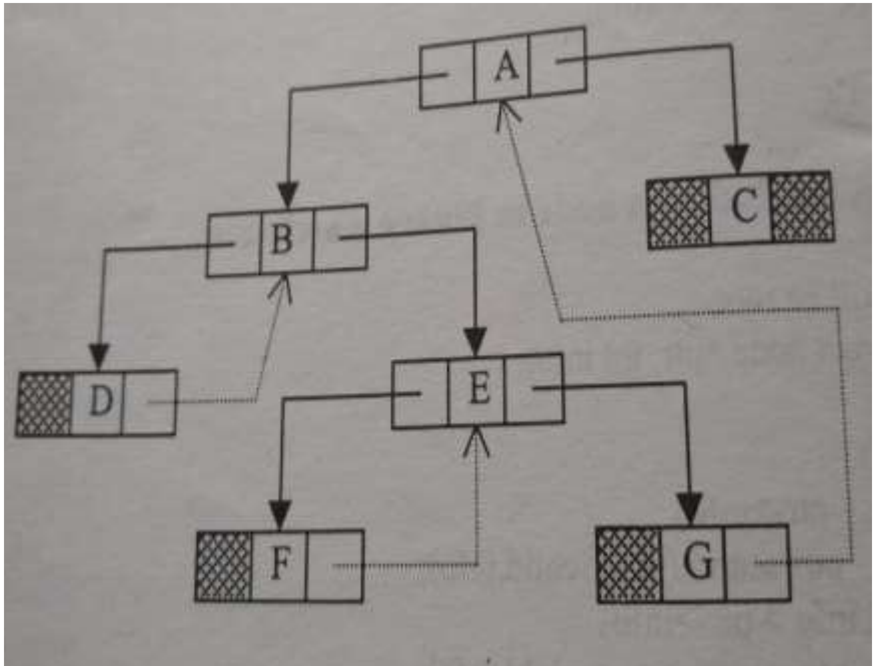
Fully In-Threaded Binary Tree/In-Threaded Tree

- If both left and right fields are used for threading

Right In-Threaded Binary Tree

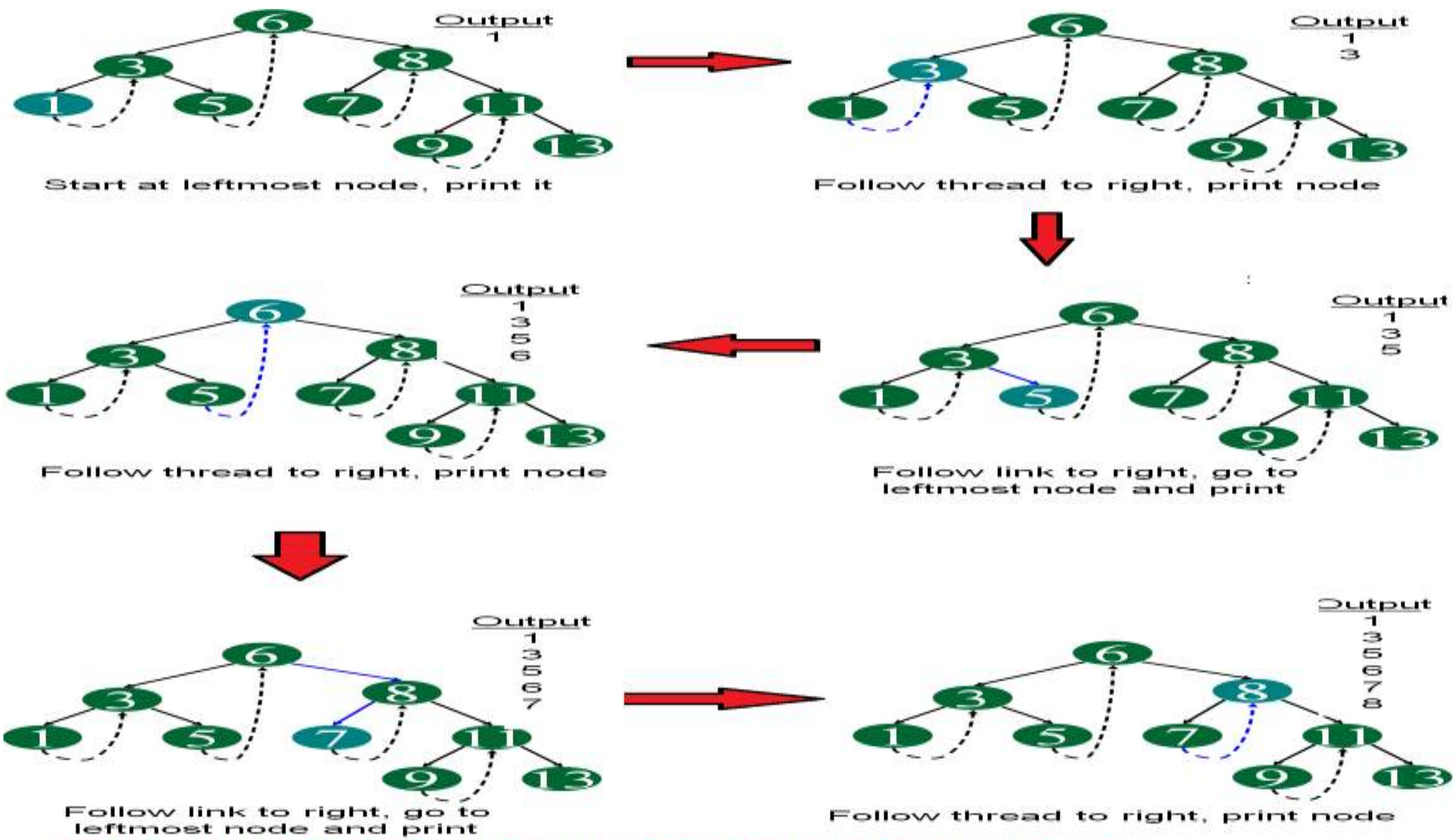
- **Each NULL right pointer is altered to contain a thread to point to that node's inorder successor.**

- For Eg-
- Thread of D points to B which is inorder successor of D



Right-in –threaded binary tree

Right In-Threaded Binary Tree



continue same way for remaining node.....

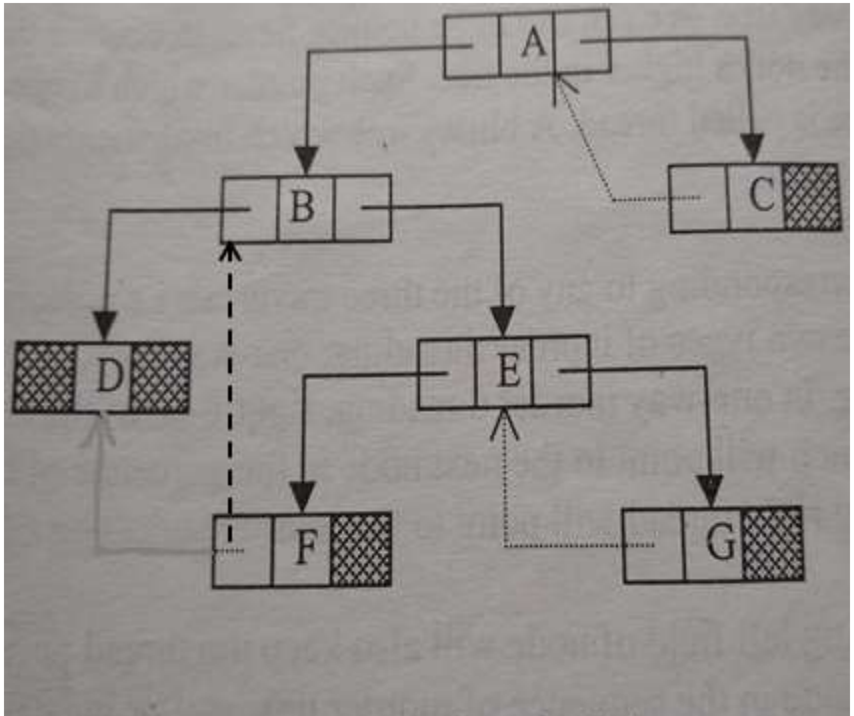
Prof. Shweta Dhawan Chachra

Courtesy:https://www.geeksforgeeks.org/threaded-binary-tree/

Left In-Threaded Binary Tree

- **Each NULL left pointer is altered to contain a thread to point to that node's inorder predecessor.**

- For Eg-
- Thread of F points to B which is inorder predecessor of F

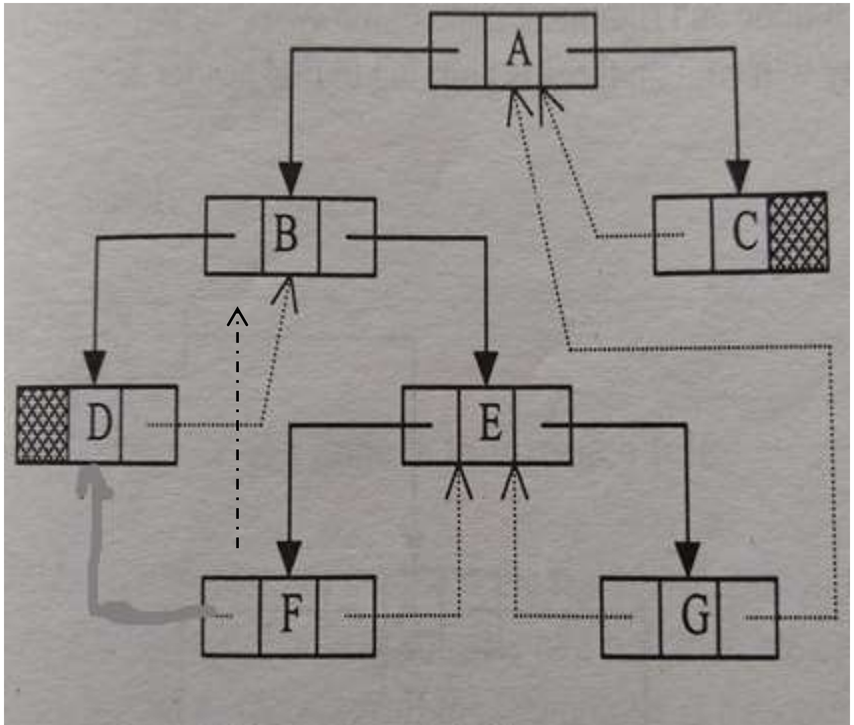


Left-in-threaded binary tree

Fully In-Threaded Binary Tree

- If both left and right fields are used for threading

- For eg-
- Left thread F points to E, inorder successor
- Right thread F points to D, inorder predeccessor



Fully in-threaded binary tree

Preorder Threading-

- Similarly , we can have Right Pre-threaded and Left Pre-threaded Tree corresponding to the Preorder Traversal.

Structure of a node in 2way In Threaded Tree

```
typedef enum {thread,link} boolean;
struct node
        {
        struct node *left_ptr;
        boolean left;
        int info;
        struct node *right_ptr;
        boolean right;
        };
```

Structure of a node in 2way In Threaded Tree

```
typedef enum {thread,link} boolean;
struct node
        {
        struct node *left_ptr;
        boolean left;
        int info;
        struct node *right_ptr;
        boolean right;
        };
```

- Two boolean numbers
  - left
  - right
- to differentiate between a thread and link

Enumeration

- An enumerated data type
- Enumeration
- Enum
- Data type consisting of a set of named values called elements, members or enumerators of the type

Enumeration

Eg-

enum e_tag {a,b,c,d=20,e,f,g=20,h}var;

In absence of initialization , the values assigned start at Zero and increase

- a=0
- b=1
- c=2
- d=20
- e=21
- f=22
- g=20
- h=21

Enum used-

- **typedef enum {thread,link} boolean;**
- declares an enumeration datatype called boolean
- thread=0
- link=1

Structure of a node in 2way In Threaded Tree

- These members can take values thread or link
  - **left =link**, pointer left_ptr points to left child of the node
  - **left=thread,** pointer left_ptr is a thread pointing to inorder predecessor of the node
  - **right=link**, pointer right_ptr points to right child of the node
  - **right=thread,** pointer right_ptr is a thread pointing to inorder successor of the node

In-Threaded Tree

- In Inorder traversal
  - **1st Node has no predecessor**
  - **Last Node has no Successor**
  - **Left pointer of Leftmost Node/1st Node=NULL**
  - **Right Pointer of Rightmost Node/ Last Node =NULL**

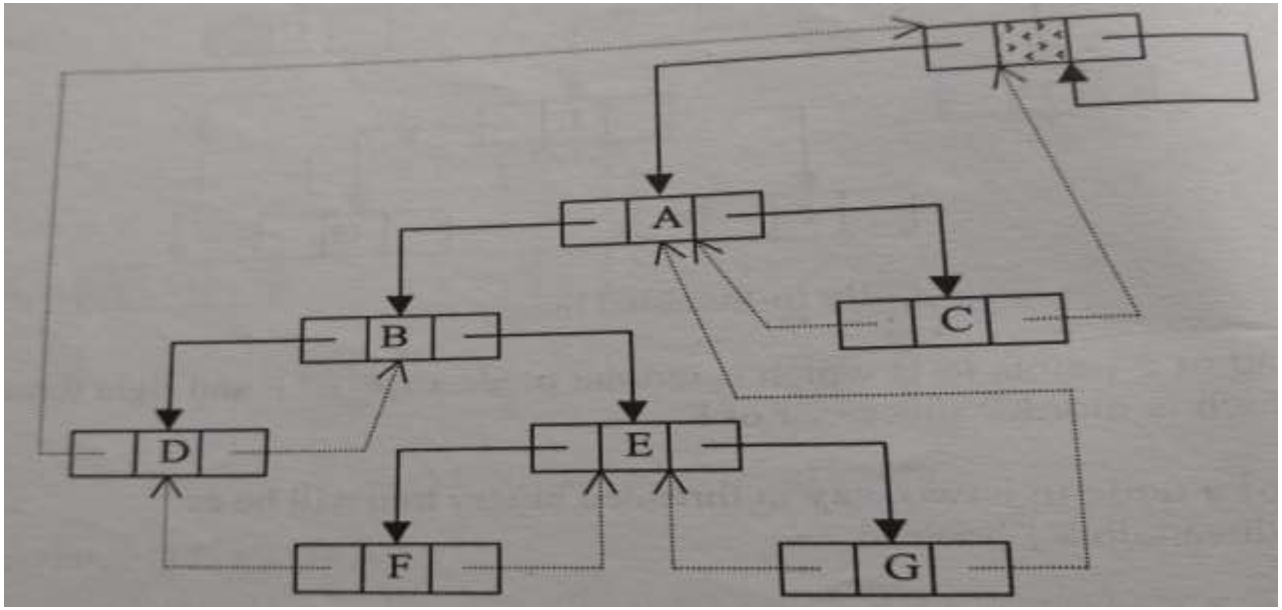
- **Still NULL values…..**
- **Solution?**

Solution= In-Threaded Tree with Header Nodes

Solution= In-Threaded Tree with Header Nodes

- A dummy node=Header Node is taken
- Our tree will be the left subtree of this Header Node.
- Left pointer of Header Node will point to the root node of our tree.

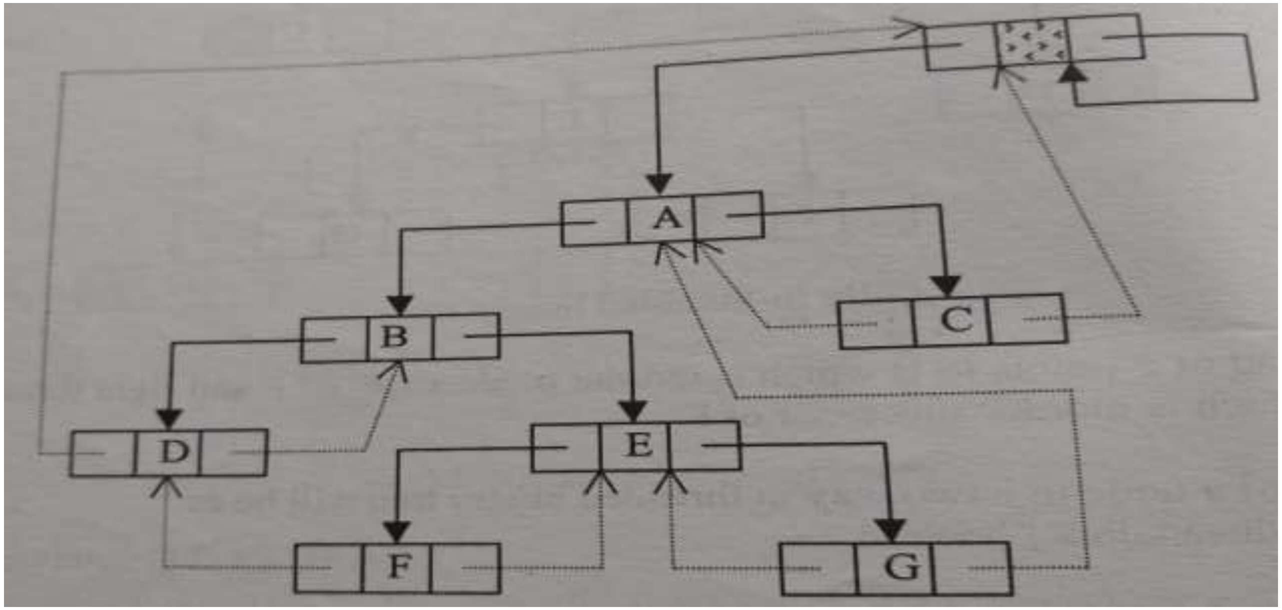
Header Node



Solution= In-Threaded Tree with Header Nodes

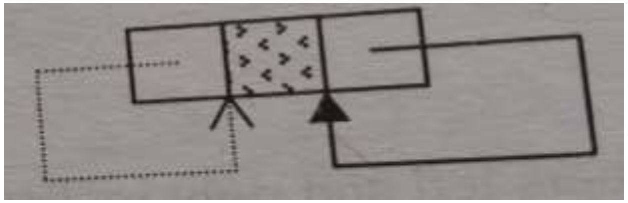
- Now, Leftmost/Rightmost Node will not contain NULL
- **Will contain threads pointing to this Header Node**

Header Node



Solution= In-Threaded Tree with Header Nodes

- When our tree will be empty then Left pointer of Header Node will be a thread pointing to itself.
- Condition for empty In-threaded Tree with Header Node-
  head->lchild=head



Header Node

Inorder Traversal in In-Threaded Binary Tree

Finding Inorder Successor of a node in In-threaded Node

The Inorder Successor of a node is the Leftmost node in the right subtree of that node

1) If the Right pointer of a node consists of a link then we can traverse the right subtree and find the inorder successor.

2) If the right pointer is a thread, then that thread will point to the inorder successor

Finding Inorder Successor of a node in In-threaded Node

```
struct node * in_succ(struct node *ptr)
{
        struct node *succ;
        if(ptr->right==thread)
                succ=ptr->right_ptr;
        else
        {
                ptr=ptr->right_ptr;
                while(ptr->left==link)
                        ptr=ptr->left_ptr;
                succ=ptr;
        }
        return succ;
}
```

```
typedef enum {thread,link} boolean;
struct node
        {
        struct node *left_ptr;
        boolean left;
        int info;
        struct node *right_ptr;
        boolean right;
        };
```

Finding Inorder Predeccessor of a node in In-threaded Node

- **The Inorder Predeccessor of a node is the Rightmost node in the left subtree of that node.**

1) If the Left pointer of a node consists of a link then we can traverse the left subtree and find the inorder predeccessor.

2) If the left pointer is a thread, then that thread will point to the inorder predecessor

Finding Inorder Predecessor of a node in In-threaded Node

```
struct node * in_pred(struct node *ptr)
{
        struct node *pred;
        if(ptr->left==thread)
                pred=ptr->left_ptr;
        else
                {
                ptr=ptr->left_ptr;
                while(ptr->right==link)
                        ptr=ptr->right_ptr;
                pred=ptr;
        }
        return pred;
}
```

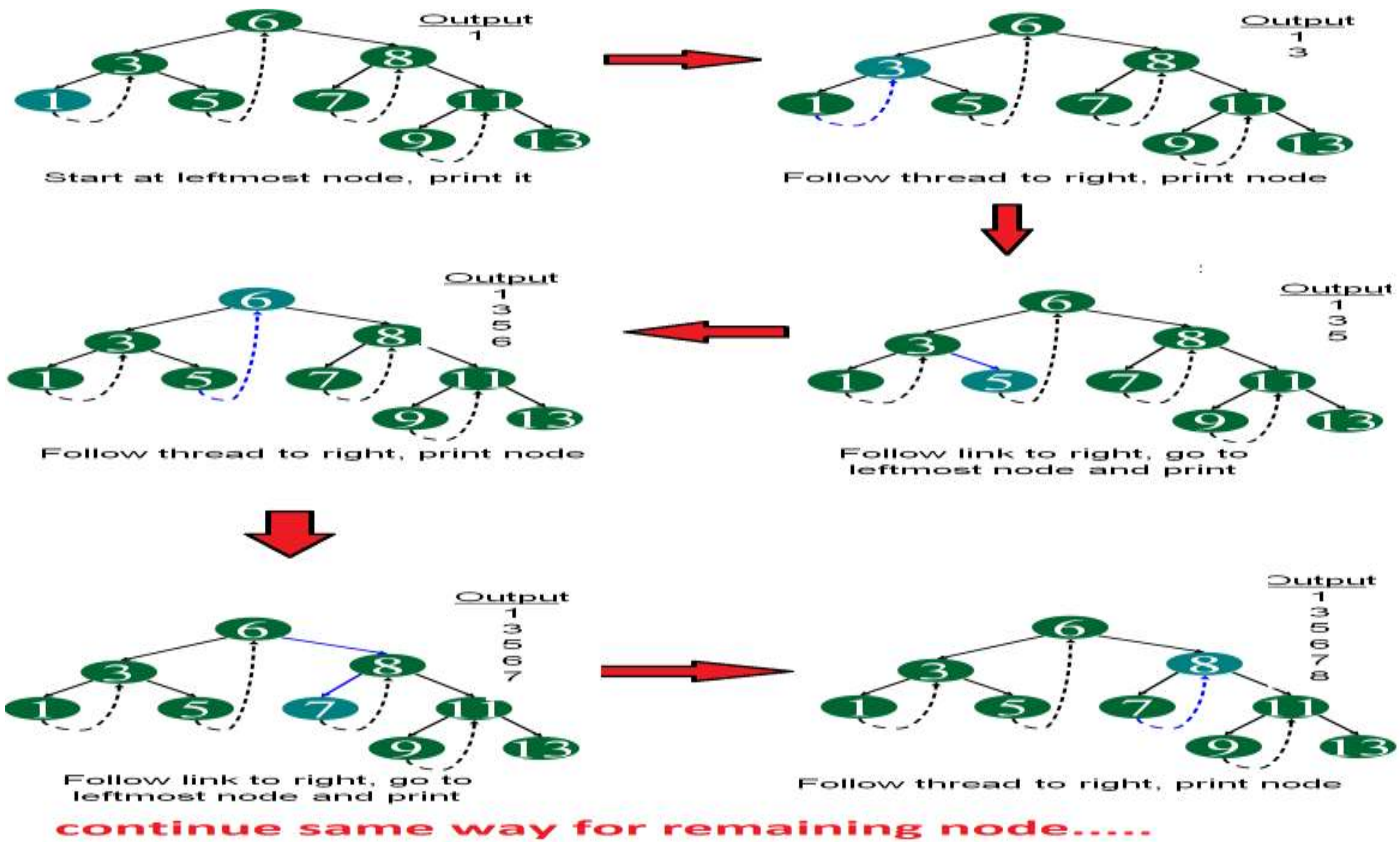
```
typedef enum {thread,link} boolean;
struct node
        {
        struct node *left_ptr;
        boolean left;
        int info;
        struct node *right_ptr;
        boolean right;
        };
```

Inorder Traversal in Right In-threaded Node

- **Traverse the leftmost node of the tree**
- With the help of in_succ() function, find the inorder successor of each node and traverse it

- **Rightmost node of the tree is the last node in the inorder traversal and**
- Its right pointer is a thread points to the header node,
- Hence we stop on reaching header node

Right In-Threaded Binary Tree



Prof. Shweta Dhawan Chachra

Courtesy:https://www.geeksforgeeks.org/threaded-binary-tree/

Inorder Traversal in In-threaded Node

```
inorder()
{
        struct node *ptr;
        if(head->left_ptr==head)
                {
                printf("Tree is empty");
                return;
                }
        ptr=head->left_ptr;
        /*Find the leftmost node and traverse it*/
        while(ptr->left==link)
                ptr=ptr->left_ptr;
        printf("%d",ptr->info);
        while(1)
        {
                ptr=in_succ(ptr);
                if(ptr==head)        /*If last node is reached*/
                        break;
                printf("%d",ptr->info);
        }/*end of while*/
}
```

- **Rightmost node of the tree is the last node in the inorder traversal and**
- **Its pointer right poiter is a thread points to the header node,**
- **Hence we stop on reaching header node**

31-10-2020

AVL Trees

AVL Trees

- Why is it called so?

AVL Trees

- **Why is it called so?**
- Russian Mathematician **G.M. Adelson Velskii and E.M . Landis** came with a new technique for **balancing binary search tree**
- Called AVL trees on their names.

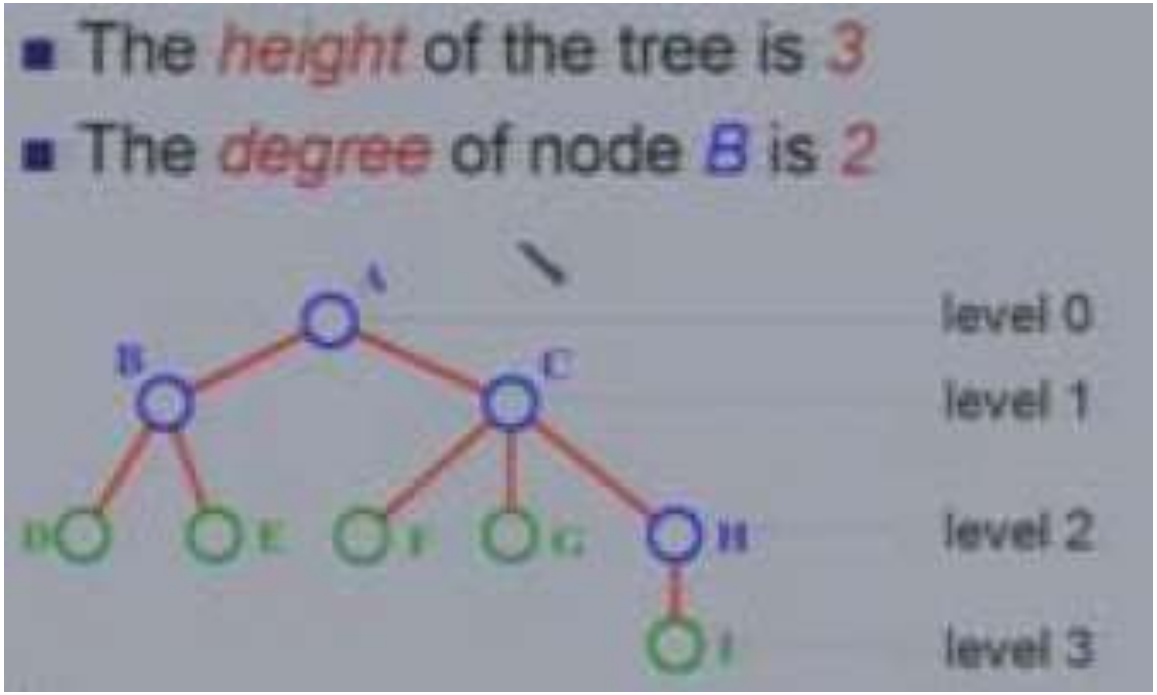
AVL Trees

- **Balanced binary search tree**
- A binary search tree where height of left and right subtree of any node will be with maximum difference 1
- Each node has a balance factor.
- <u>**Balance Factor=Height of Left subtree-Height of Right Subtree**</u>

Tree

Basic Terminology-

- Height -
  - Maximum level of any leaf in the tree.



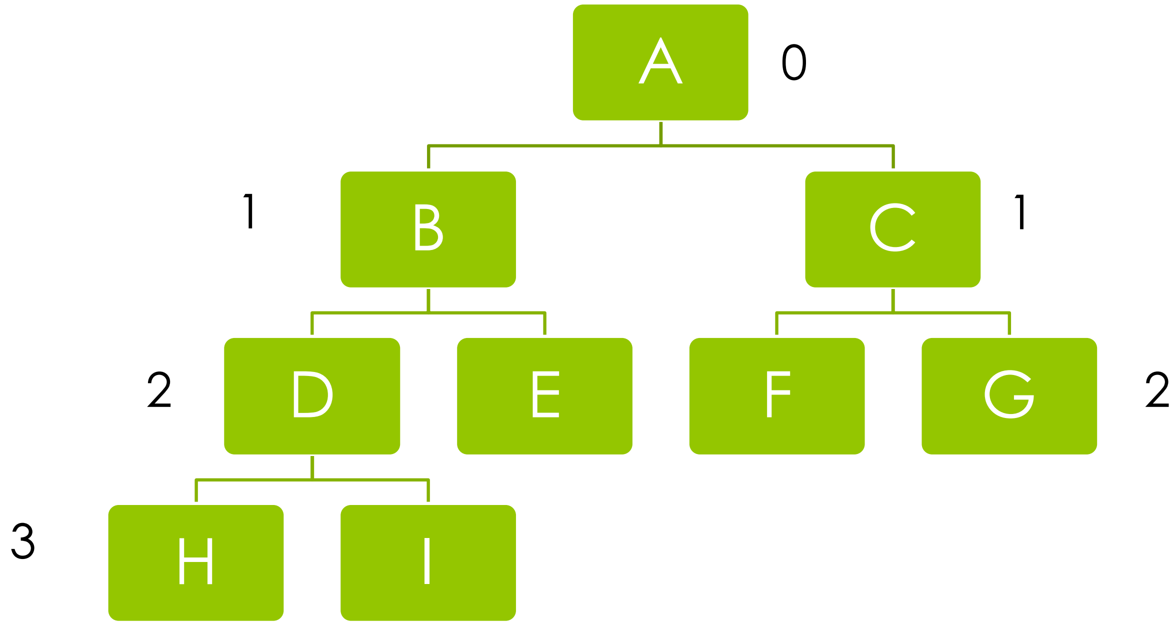
AVL Trees

- **Right Heavy Node/Right High**
  - If Height of right subtree is one more than height of its left subtree.
- **Left Heavy Node/Left High**
  - If Height of its left subtree is one more than height of its right subtree
- **Balanced Node**
  - If height of left subtree=Height of right subtree.

Balance factor –

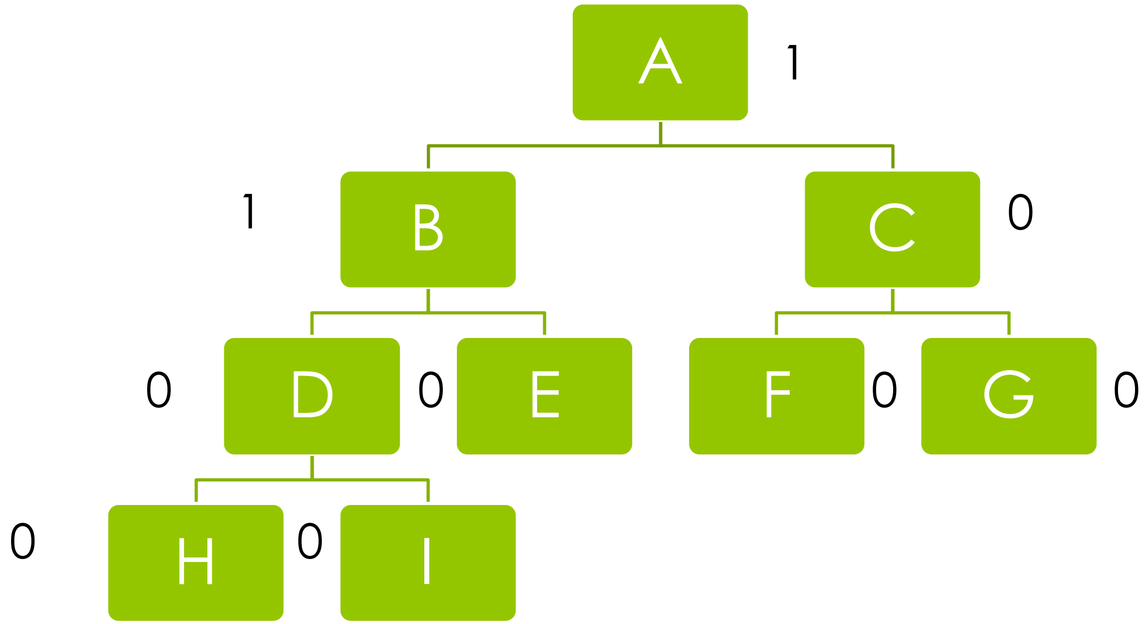
- Left High=1
- Right High=-1
- Balanced node=0

AVL trees- Node with Levels



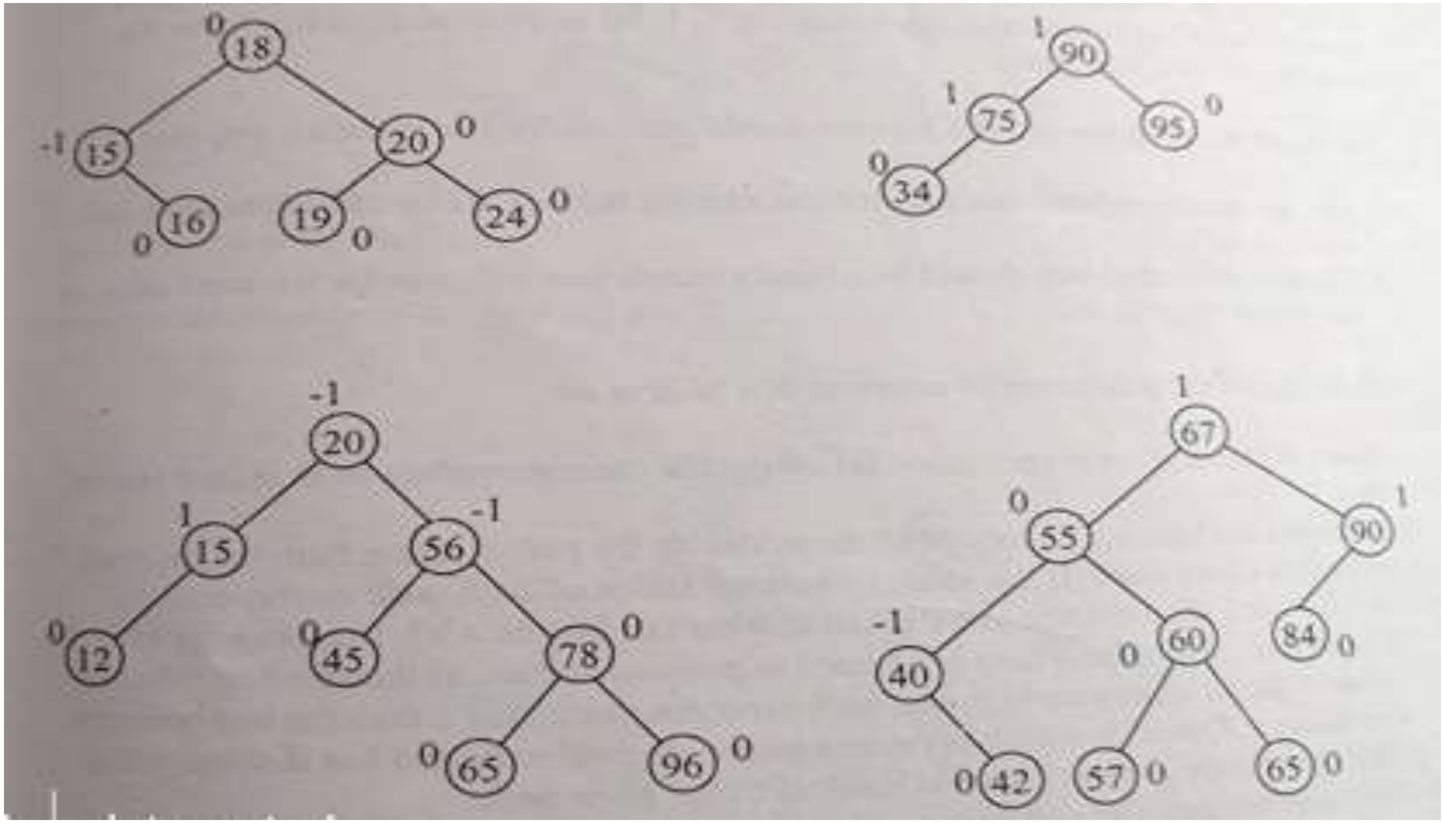
- For Node A=Left Heavy
- Balance Factor=Height of Left Subtree-Height of Right Subtree
- Balance Factor=3-2=1

AVL tree with Balance Factor

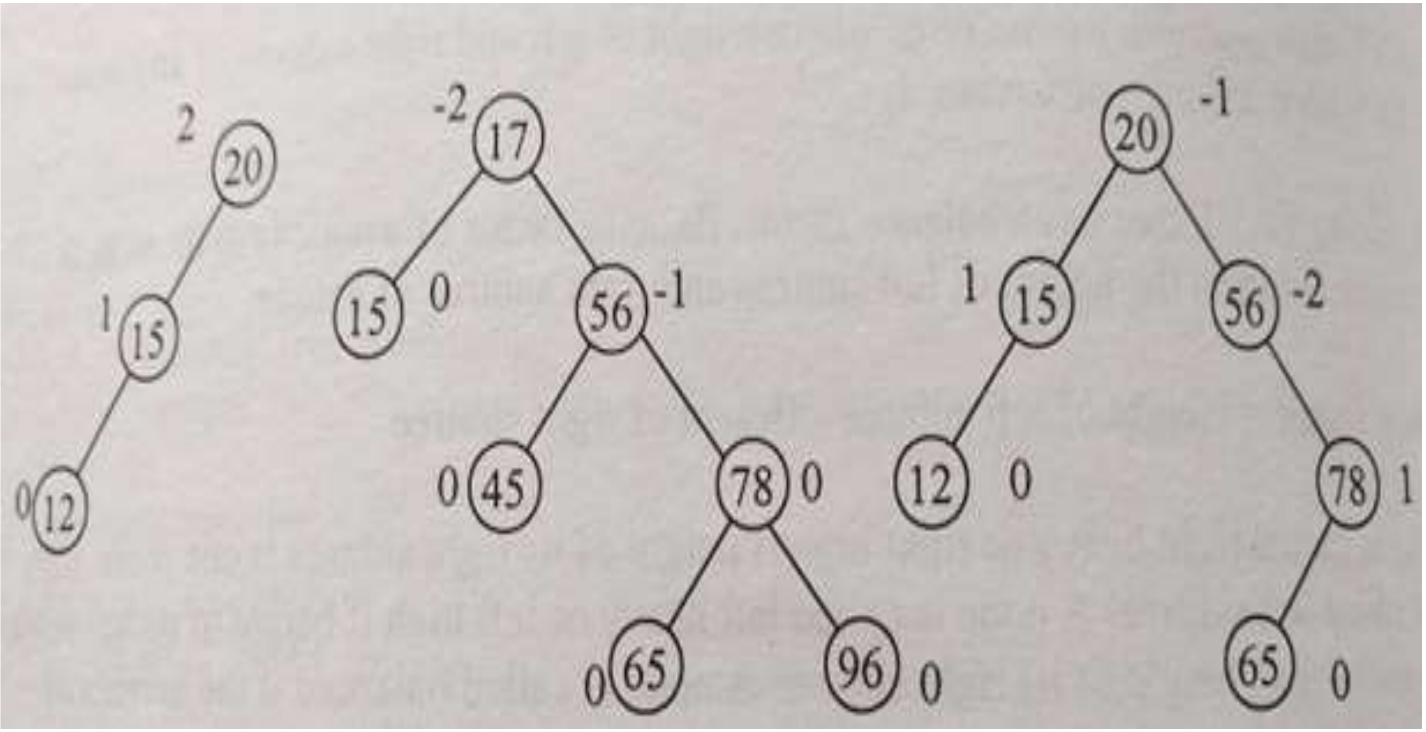


- For Node A=Left Heavy
- Balance Factor=Height of Left Subtree-Height of Right Subtree
- Balance Factor=3-2=1

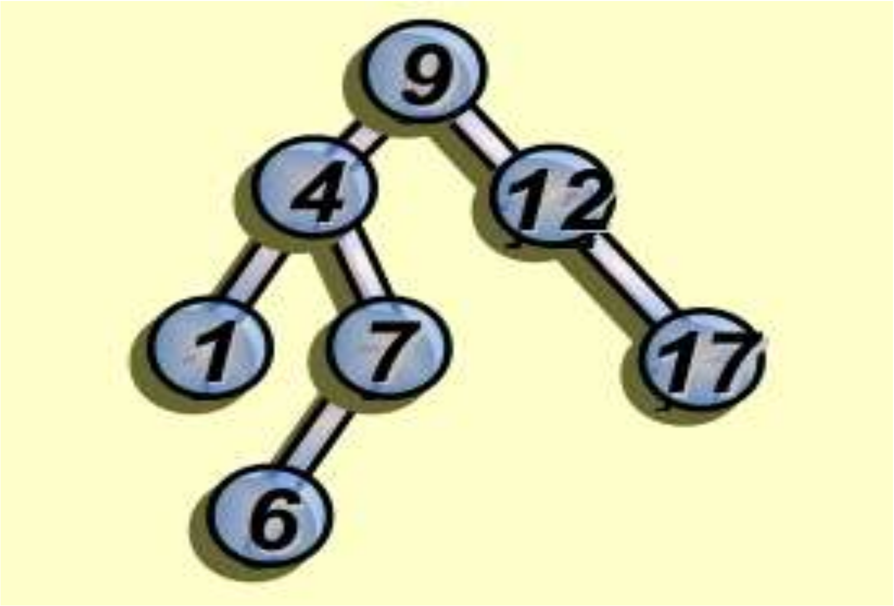
Binary Search Trees which are AVL Trees



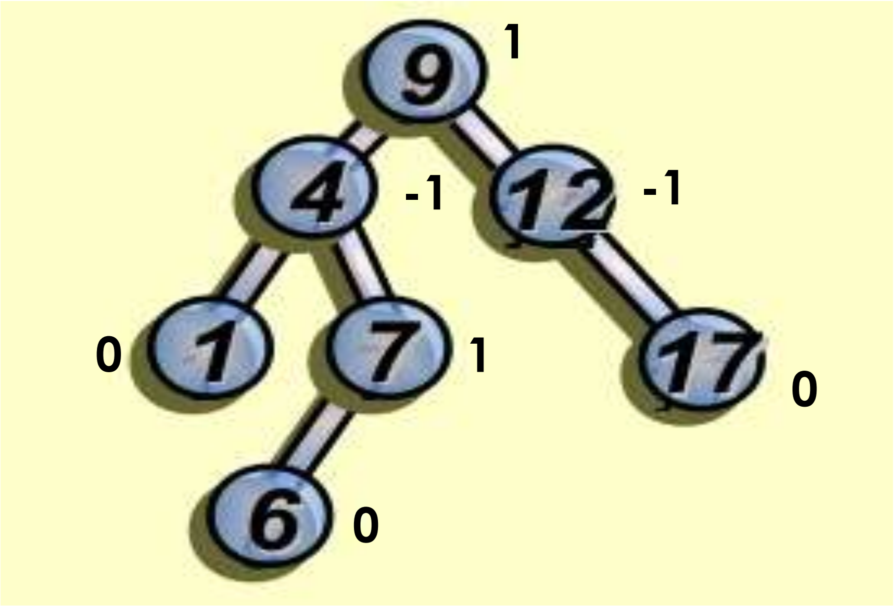
Binary Search Trees which are not AVL Trees



Calculate the Balance factor



Calculate the Balance factor



Insertion in AVL Trees–

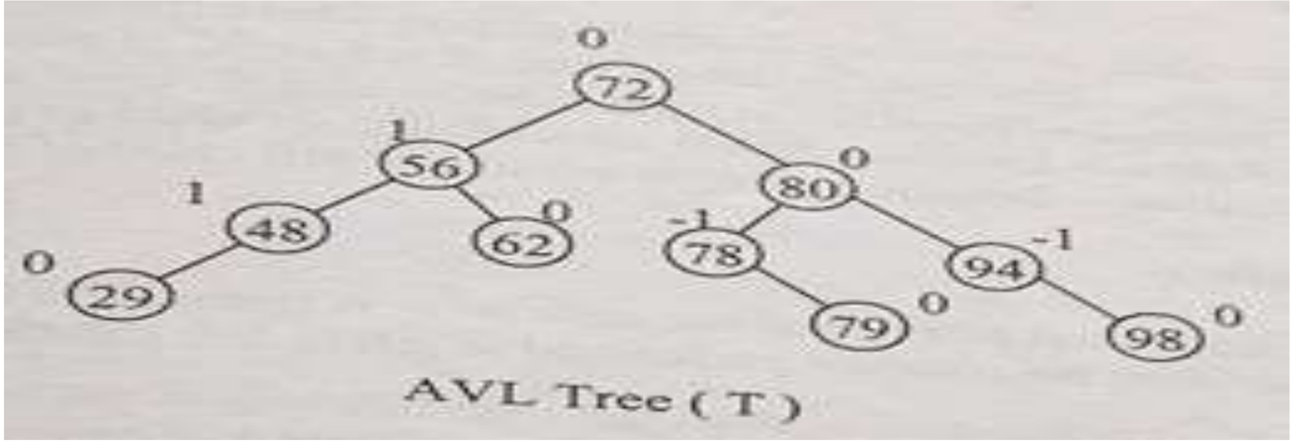
- Similar to Insertion in Binary Search Tree
- Steps-

1) Insert the node at the proper place using same procedure as in BST

2) Calculate the balance factors of all the nodes on the path starting from the inserted node to the root node.
   a) **If the tree is balanced then there is no need to proceed further.**
   b) **If absolute value of balanced factor of any node in this path >1 then the tree becomes unbalanced.**
   c) **The node which is nearest to the inserted node & has absolute value of balance factor >1 is marked as Pivot node**

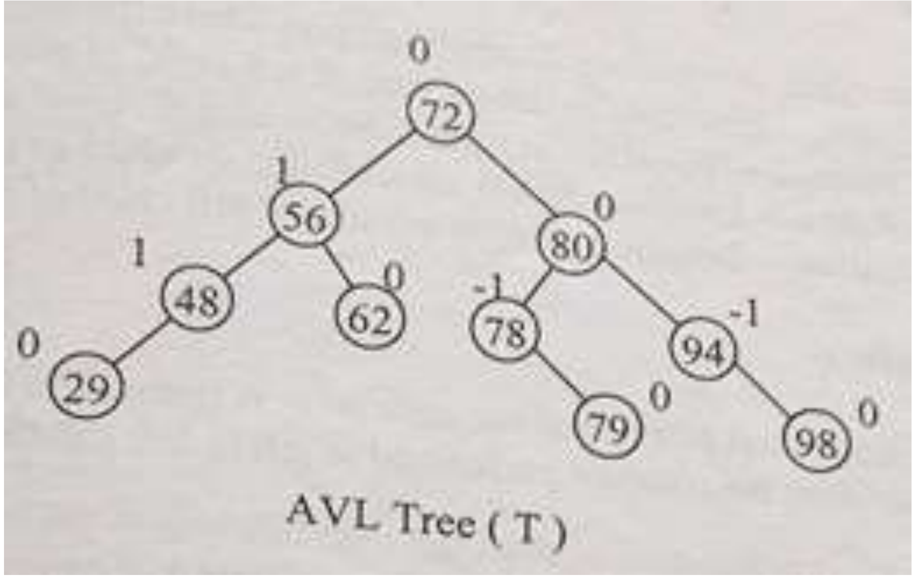
3) We perform **rotations about the pivot node**

Insertion in AVL Trees–



- **Insertion of 62 in the given AVL tree**

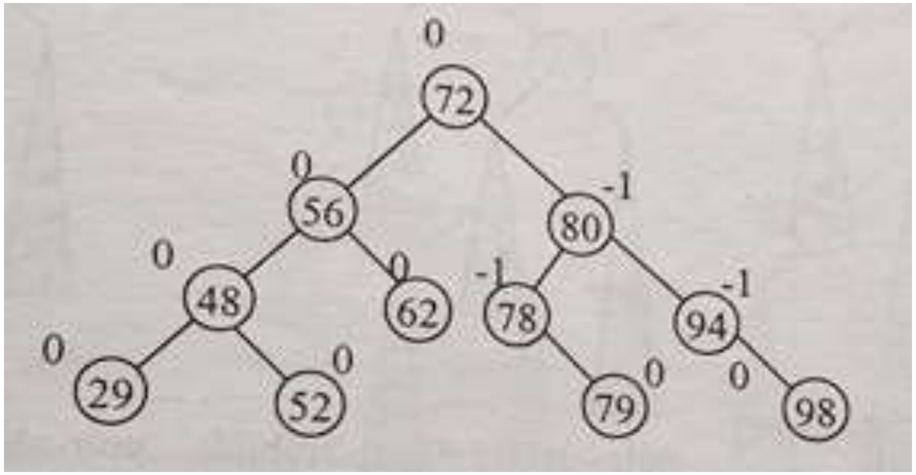
Insertion in AVL Trees–

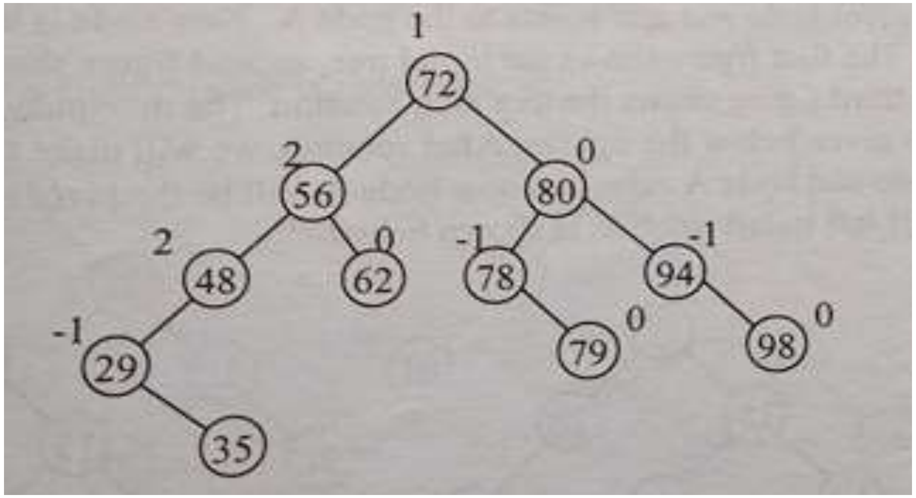


- After inserting 62,
  - the balance factors of some nodes changed
  - but tree is still balanced

- Now Insert 35

Insertion in AVL Trees–





- After inserting 35,
  - Tree is unbalanced
  - Node 48 is unbalanced and
  - Is nearest to the inserted node, so it's the Pivot Node
  - Rotations will be needed now

AVL Rotations–

4 types of Rotation:

- **Left to Left Rotation**
- **Right to Right Rotation**
- **Right to Left Rotation**
- **Left to Right Rotation**

AVL Rotations–

4 types of Rotation:

- **Left to Left Rotation**
- **Right to Right Rotation**
- **Right to Left Rotation**
- **Left to Right Rotation**

AVL Rotations–

4 types of Rotation depending upon where the new node is inserted:

Child to Subtree Relationship(Of Pivot Node)

- **Left to Left Rotation**
  - Insertion in Left subtree of left child of Pivot Node
- **Right to Right Rotation**
  - Insertion in Right subtree of right child of Pivot Node
- **Right to Left Rotation**
  - Insertion in Left subtree of right child of Pivot Node
- **Left to Right Rotation**
  - Insertion in right subtree of left child of Pivot Node

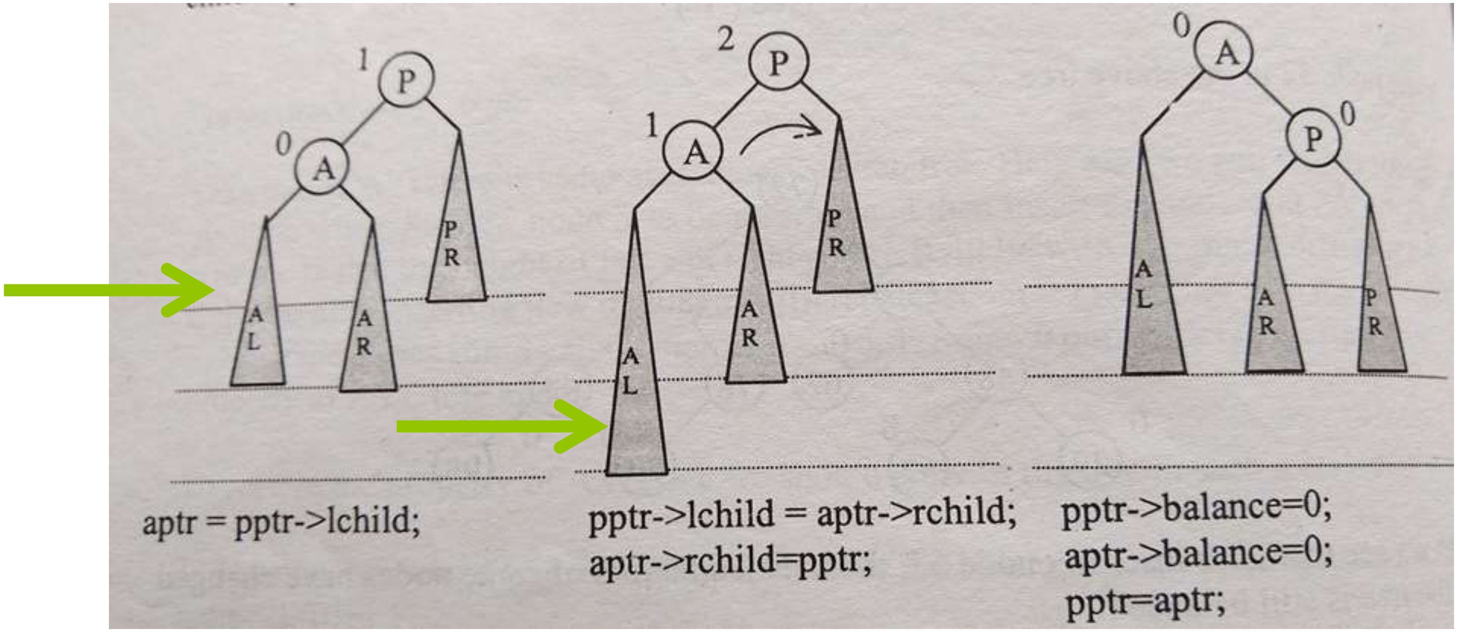
AVL Rotations–

Child to Subtree Relationship(Of Pivot Node)

- **Left to Left Rotation**
  - Insertion in Left subtree of left child of Pivot Node
- **Right to Right Rotation**
  - Insertion in Right subtree of right child of Pivot Node
- **Right to Left Rotation**
  - Insertion in Left subtree of right child of Pivot Node
- **Left to Right Rotation**
  - Insertion in right subtree of left child of Pivot Node

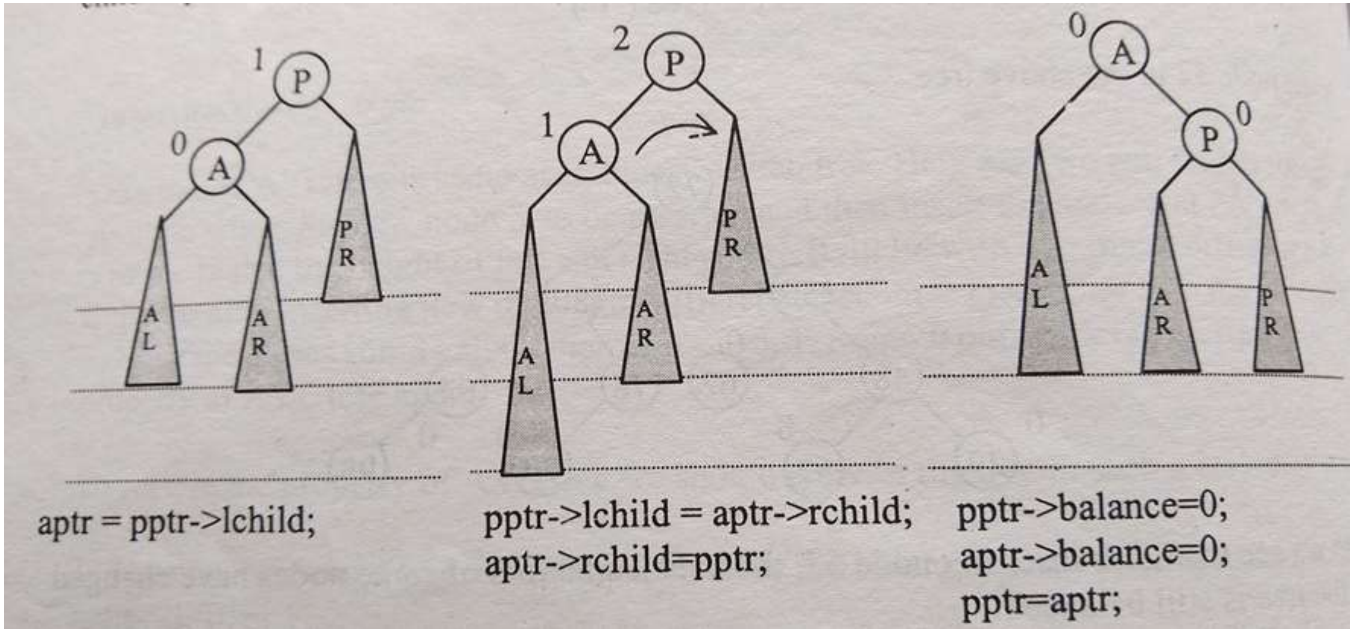
AVL Rotations–

- **Left to Left Rotation**
  - When the Pivot node is left heavy and
  - the new node is inserted in left subtree of the left child of pivot node then
  - the rotation performed is left to left rotation



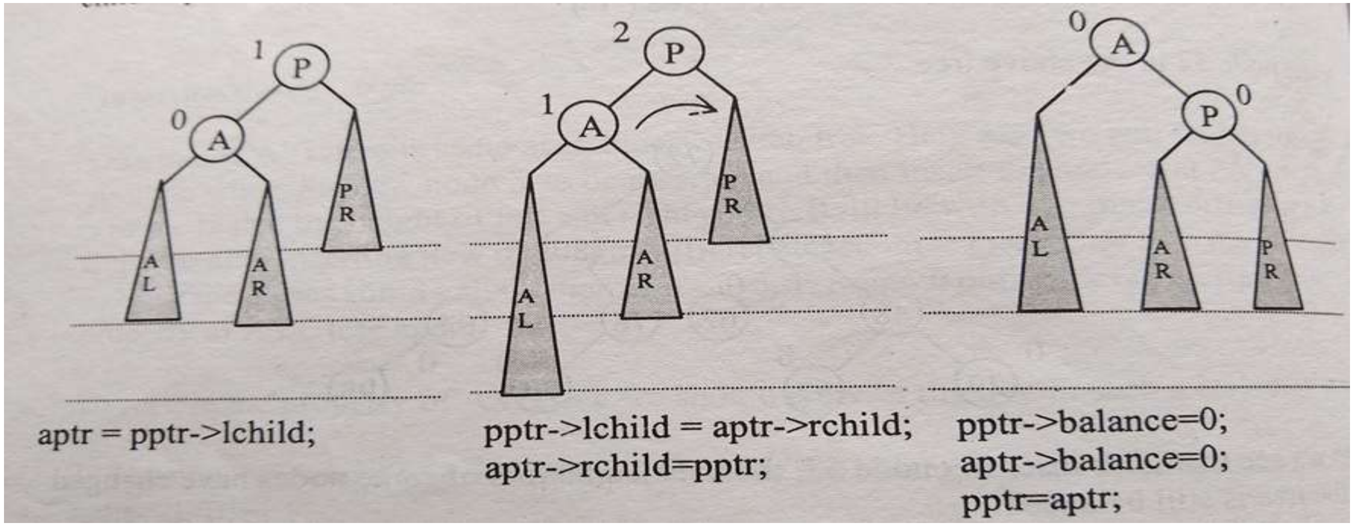
AVL Rotations–

- **Left to Left Rotation**
  - P =Pivot node
  - A =Left Child of the Pivot Node
  - AL,AR=Left and Right Subtrees of Node A
  - PR=Right Subtree of Node P



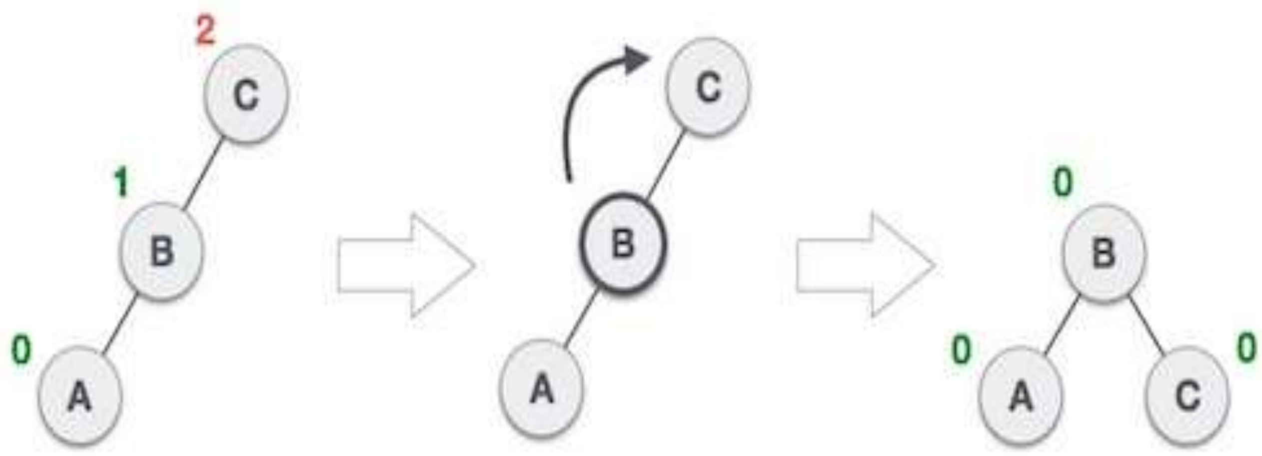
AVL Rotations–

- **Left to Left Rotation(Clockwise about Pivot)**
  - pointer pptr points to Pivot node
  - Pointer aptr points to A node
  - New node inserted in left Subtree of A
  - **Clockwise Rotation about Pivot P is performed , Now A will be the pivot node**



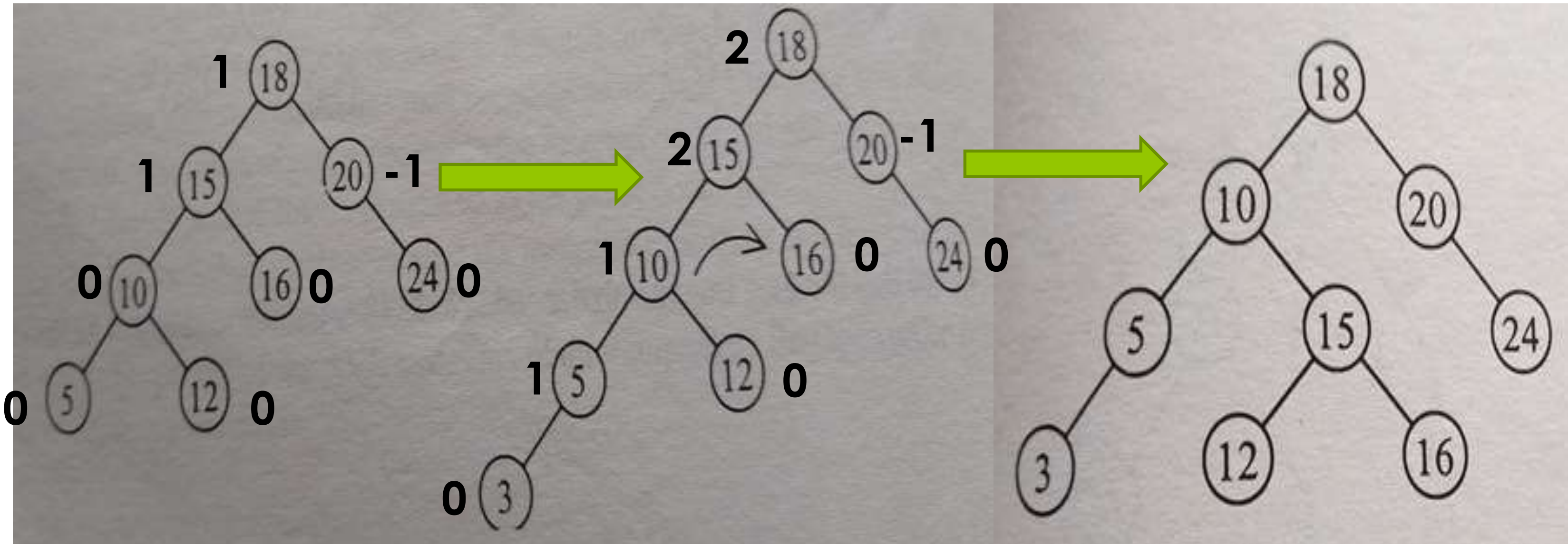
AVL Rotations–

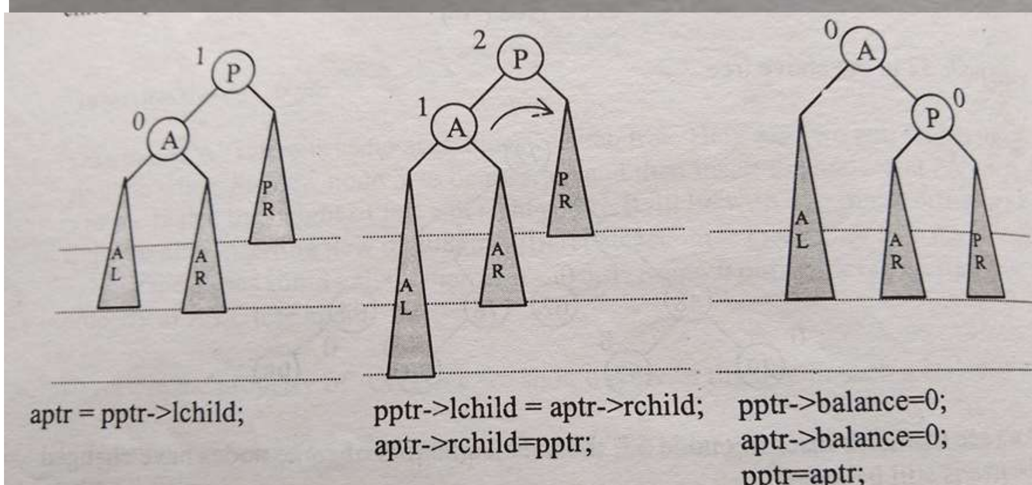
- **Left to Left Rotation-**Example
- **Left to Left Rotation(Clockwise about Pivot)**



AVL Rotations–

- **Left to Left Rotation-**Example

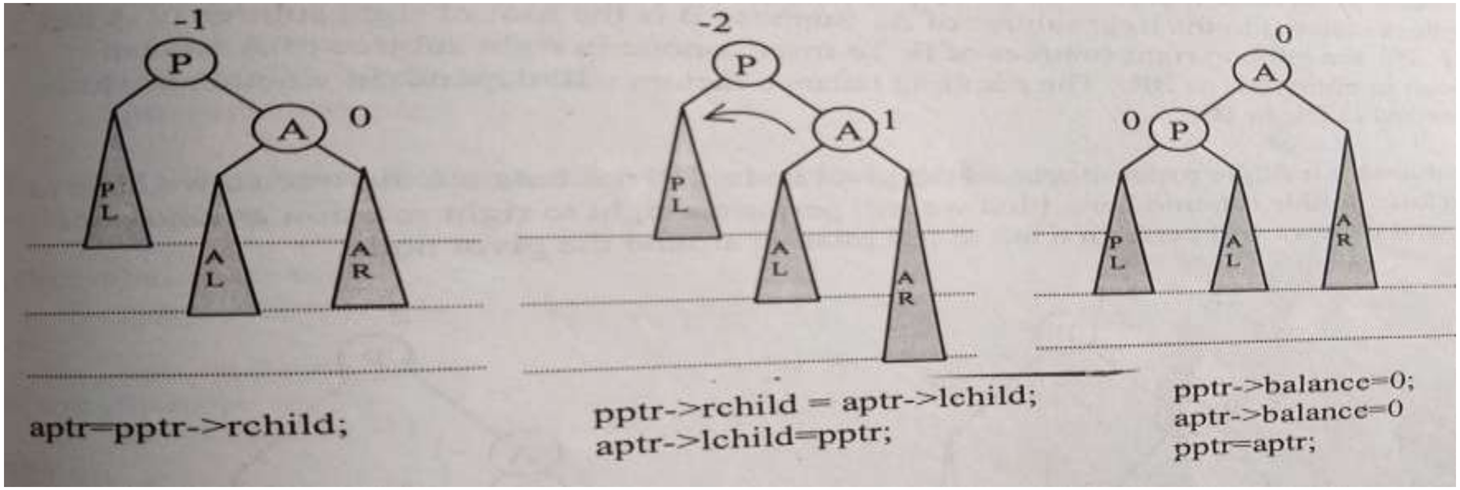




Prof. Shweta Dhawan Chachra

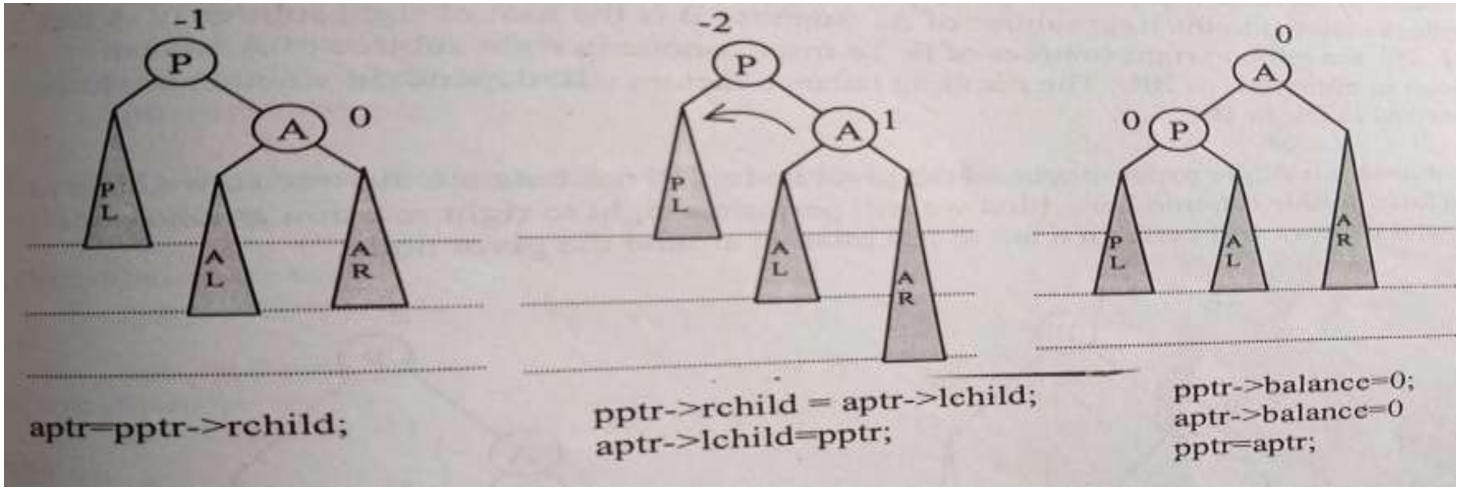
AVL Rotations–

- **Right to Right Rotation(Anti –Clockwise about Pivot)**
  - When the Pivot node is right heavy and
  - the new node is inserted in right subtree of the right child of pivot node then
  - the rotation performed is right to right rotation
  - Mirror image of Left to Left rotation



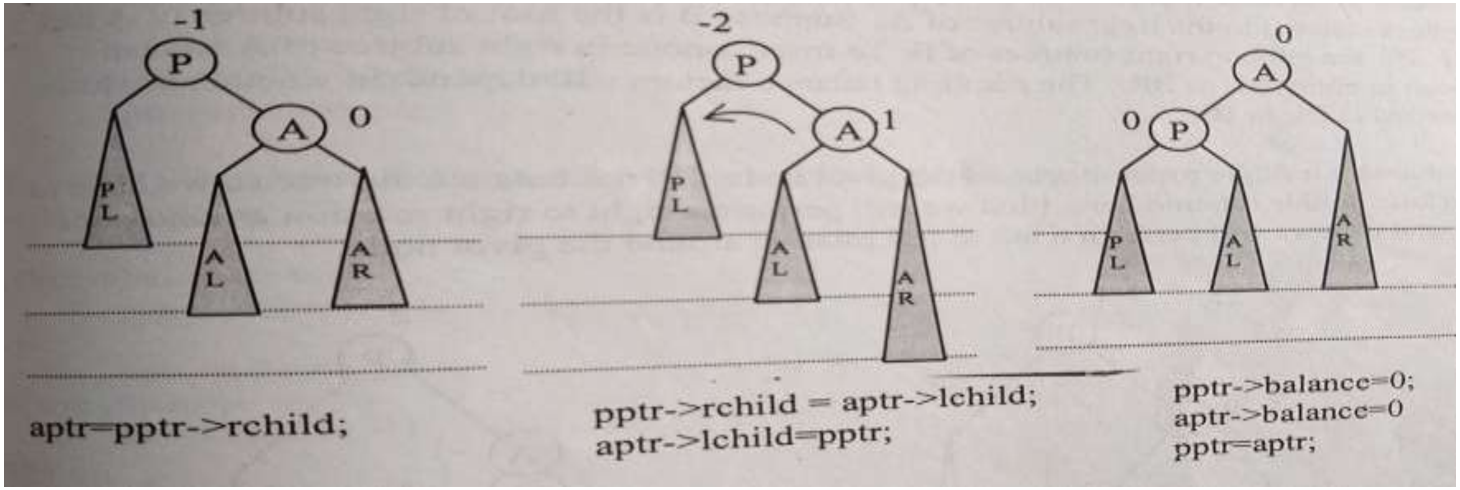
AVL Rotations–

- **Right to Right Rotation**
  - P =Pivot node
  - A=Right Child of the Pivot Node
  - AL,AR=Left and Right Subtrees of Node A
  - PL=Left Subtree of Node P



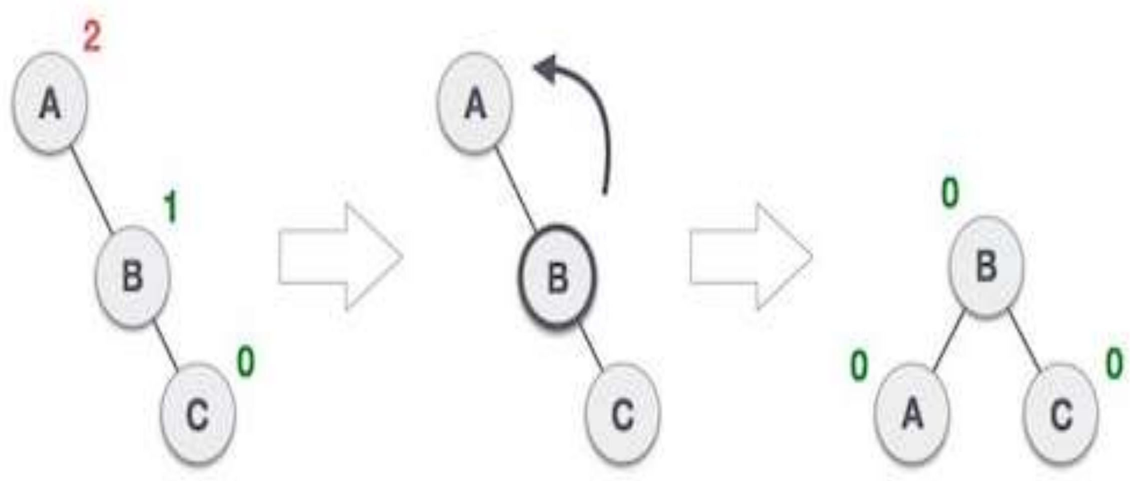
AVL Rotations–

- **Right to Right Rotation**
  - pointer pptr points to Pivot node
  - Pointer aptr points to A node
  - New node inserted in right Subtree of A
  - **Anticlockwise Rotation about Pivot P is performed, Now A will be the pivot node**



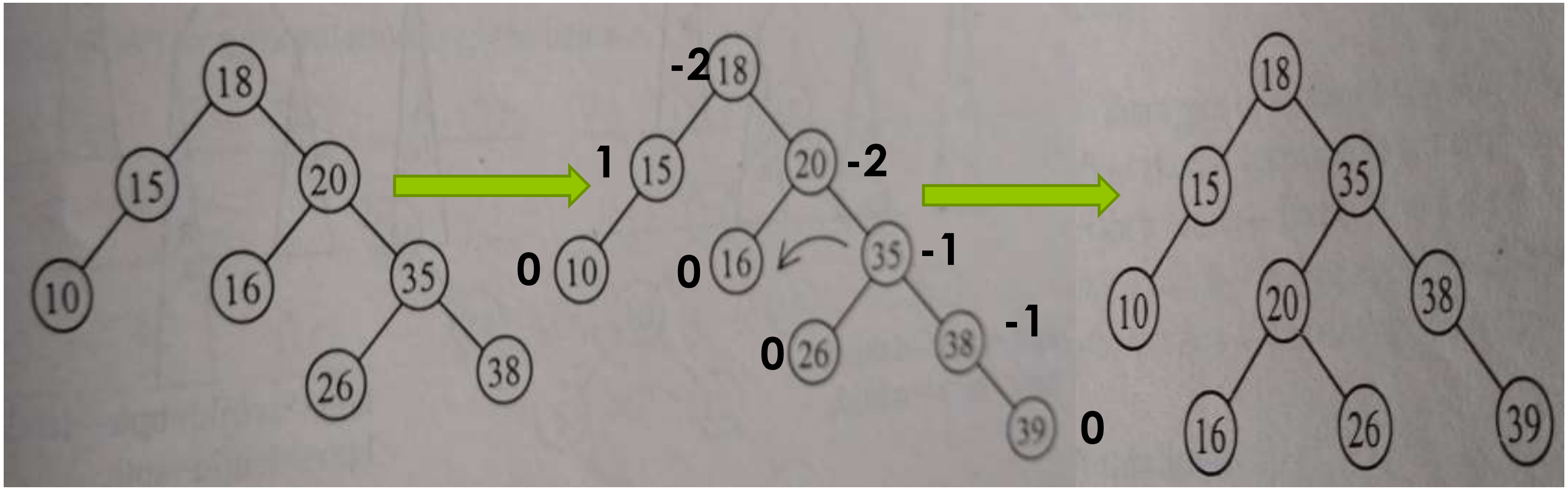
AVL Rotations–

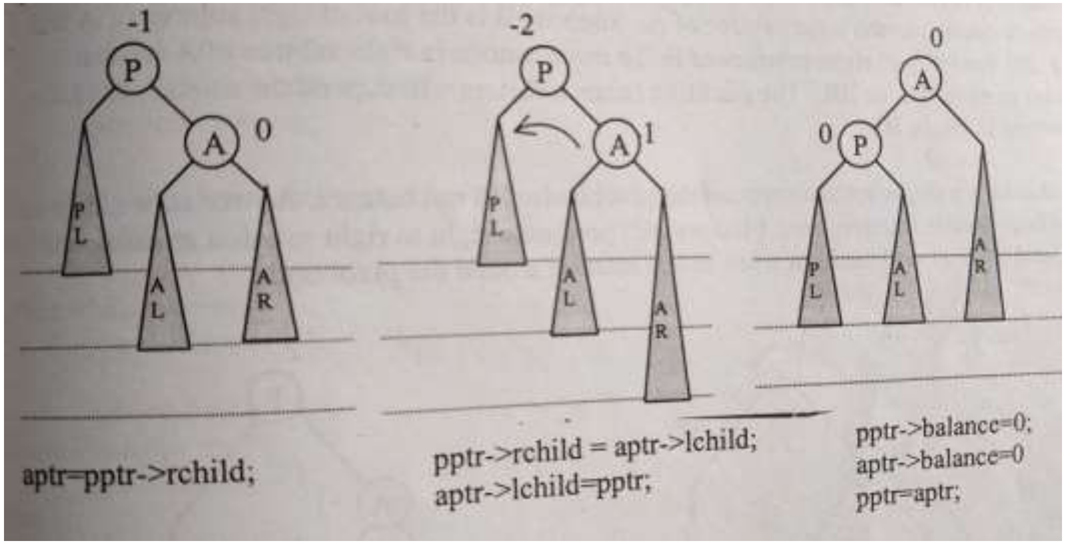
- **Right to Right Rotation-**Example
- **Anti –Clockwise about Pivot**



AVL Rotations–

- **Right to Right Rotation-**Example

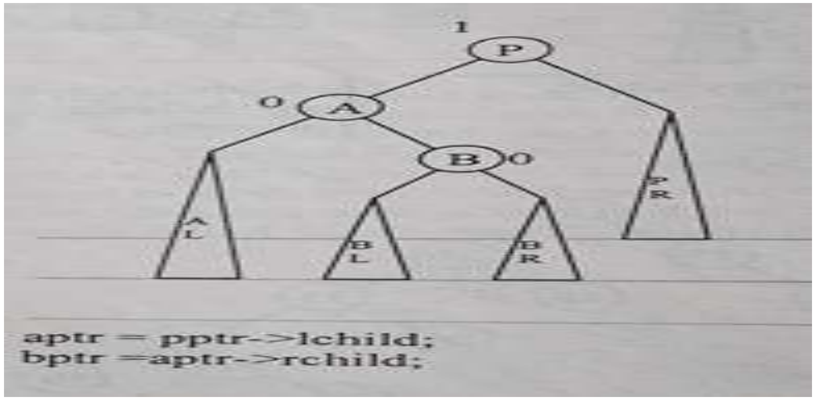




AVL Rotations–

Child to Subtree Relationship(Of Pivot Node)

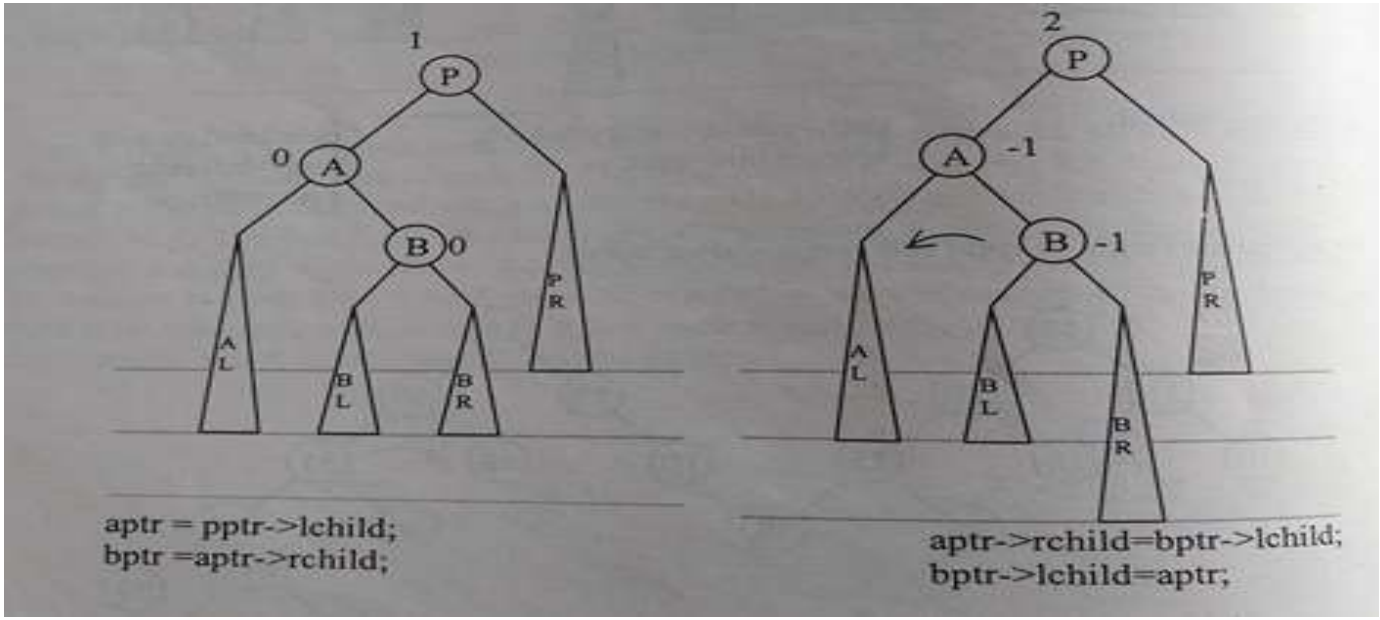
- **Left to Right Rotation**
  - New node is inserted in the right subtree of A.
  - B is the root of right subtree of A
  - BL,BR are left and right subtrees of B.
  - To insert a node in right subtree of A, **we can insert in either BL or BR.**
  - **The resulting balance factors depends on whether we have inserted in BL or BR**



AVL Rotations–

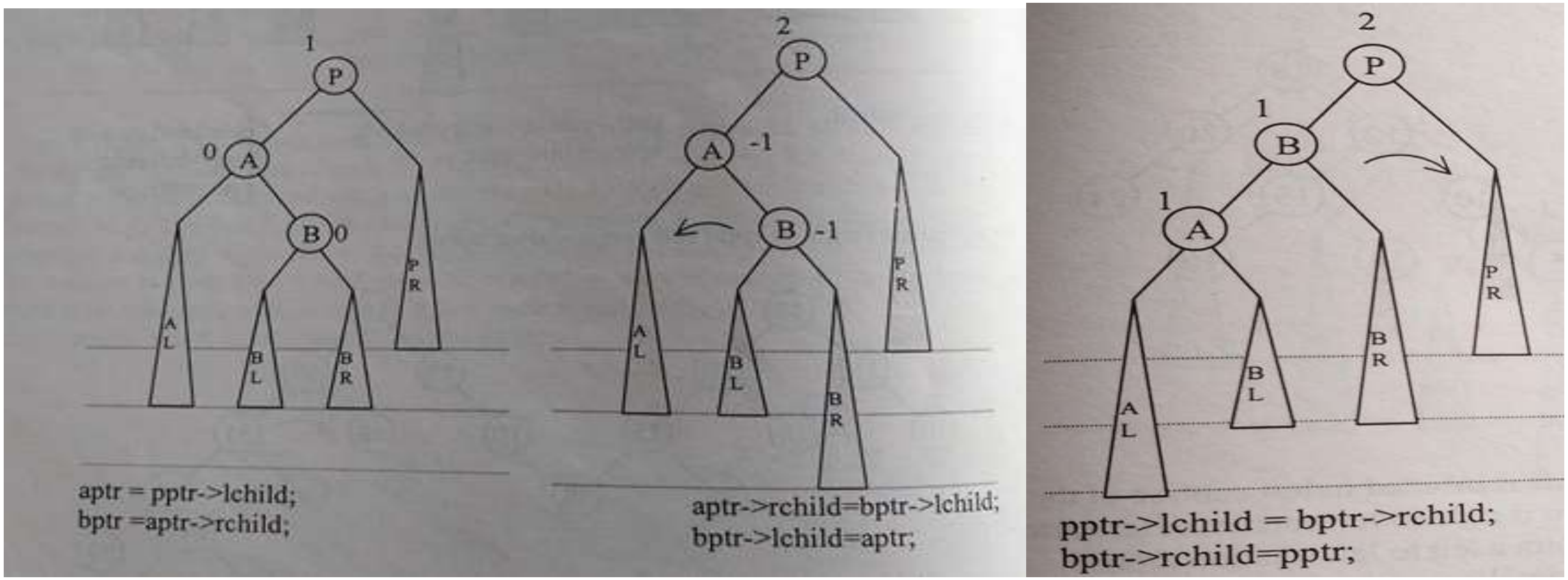
Child to Subtree Relationship(Of Pivot Node)

- **Left to Right Rotation**
  - When the Pivot node is left heavy and
  - the new node is inserted in **right subtree of the left child of pivot node then**
  - the rotation performed is left to right rotation



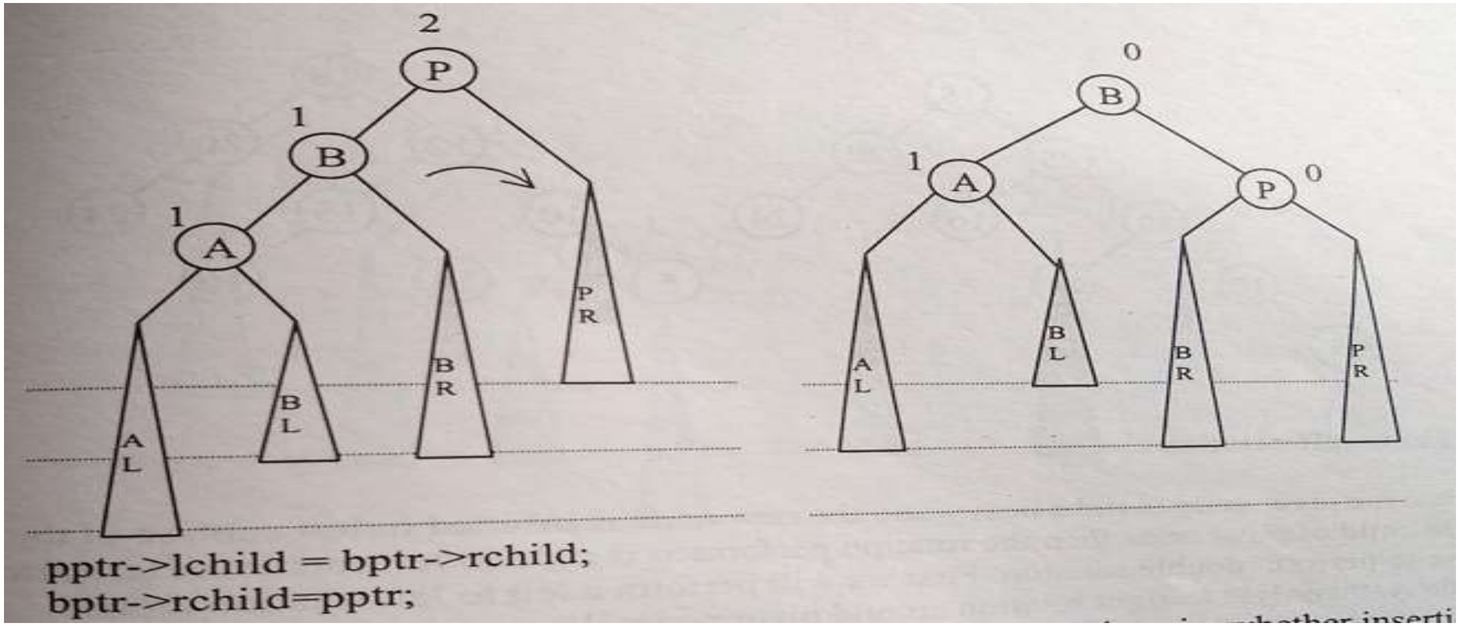
AVL Rotations–

- **Left to Right Rotation**
  - Single rotation will not balance the tree
  - Double Rotation needed here
  - **First Perform a Right to Right Rotation around node A**



AVL Rotations–

- **Left to Right Rotation**
  - Single rotation will not balance the tree
  - Double Rotation needed here
  - **Then Perform a Left to Left rotation around the pivot node**

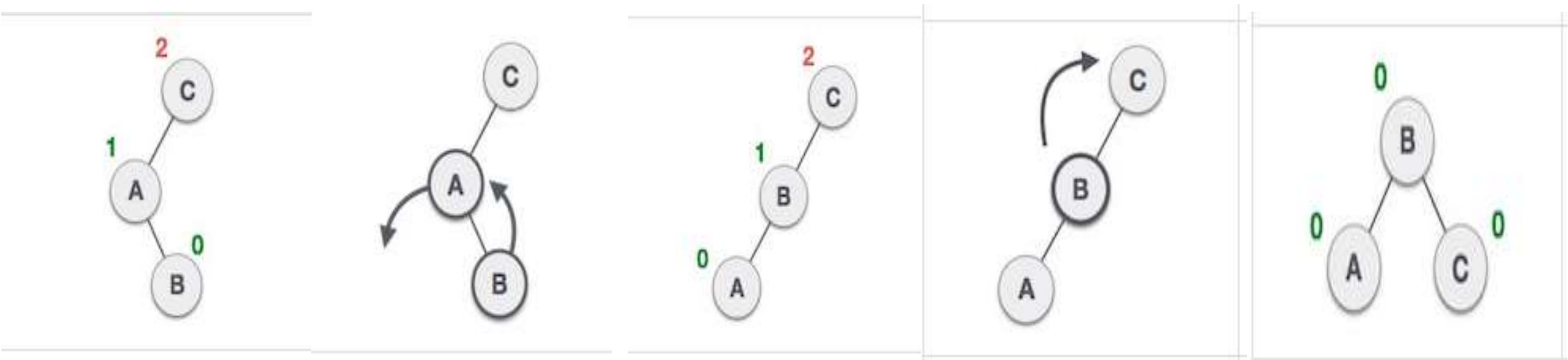


AVL Rotations–

- **Left to Right Rotation**
  - **First Perform a Right to Right Rotation around node A=>Anti –Clockwise**
  - **Then Perform a Left to Left rotation around the pivot node =>Clockwise**

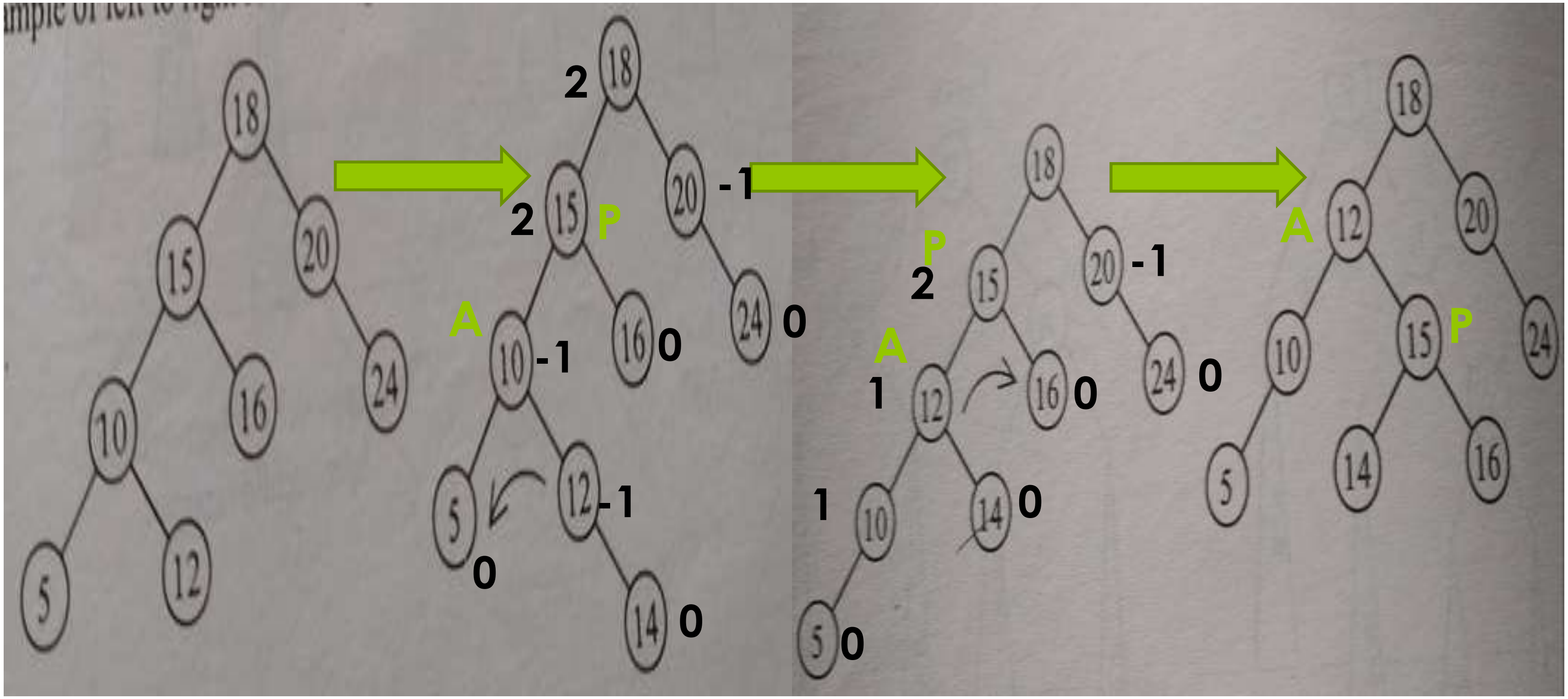
AVL Rotations–

- **Left to Right Rotation-**Example
  - **First Perform a Right to Right Rotation around node A=>Anti –Clockwise**
  - **Then Perform a Left to Left rotation around the pivot node =>Clockwise**



AVL Rotations–

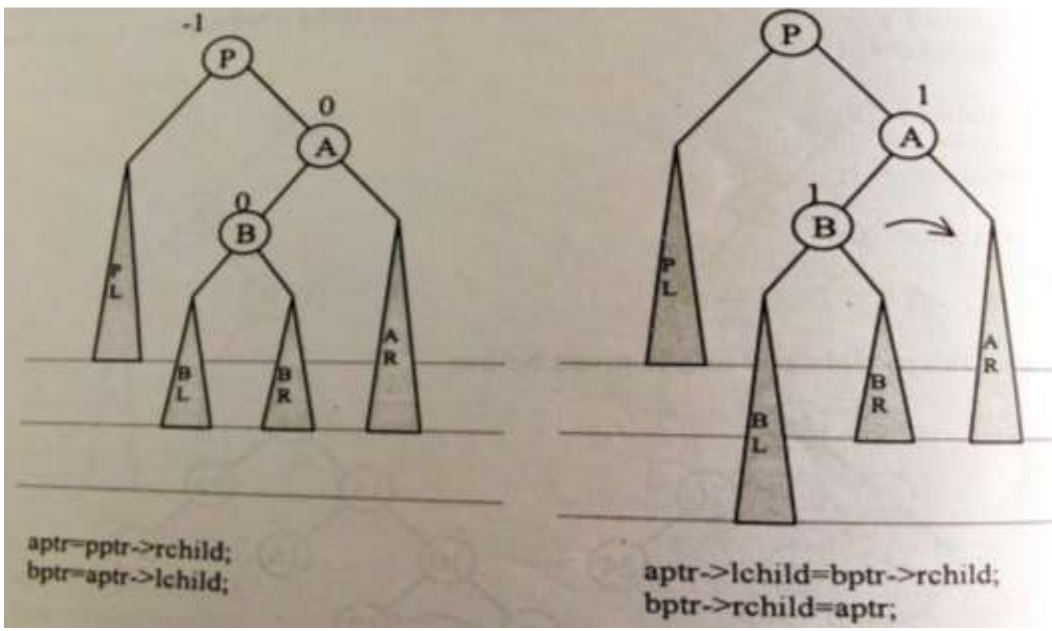
- **Left to Right Rotation-**Example
- **First Perform a Right to Right Rotation around node A=>Anti –Clockwise**
- **Then Perform a Left to Left rotation around the pivot node =>Clockwise**



AVL Rotations–

Child to Subtree Relationship(Of Pivot Node)

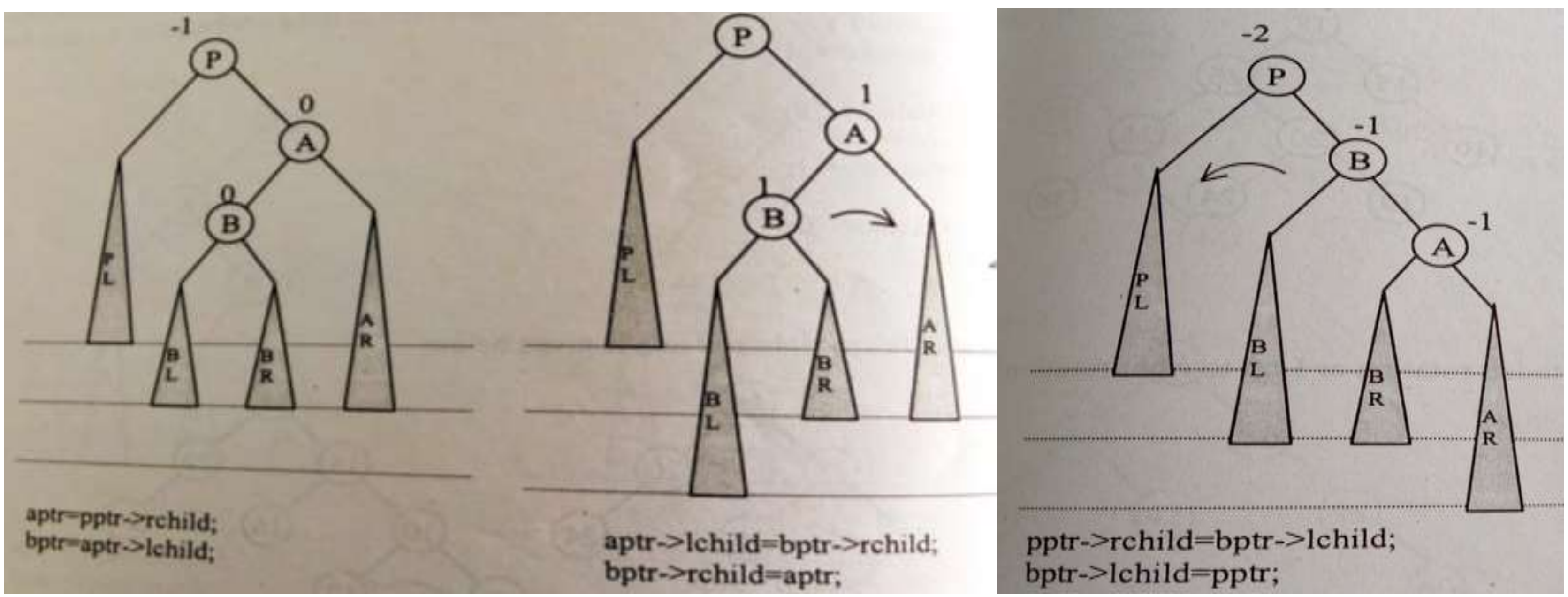
- **Right to Left Rotation**
  - When the Pivot node is right heavy and
  - the new node is inserted in left subtree of the right child of pivot node then
  - the rotation performed is right to left rotation



AVL Rotations–

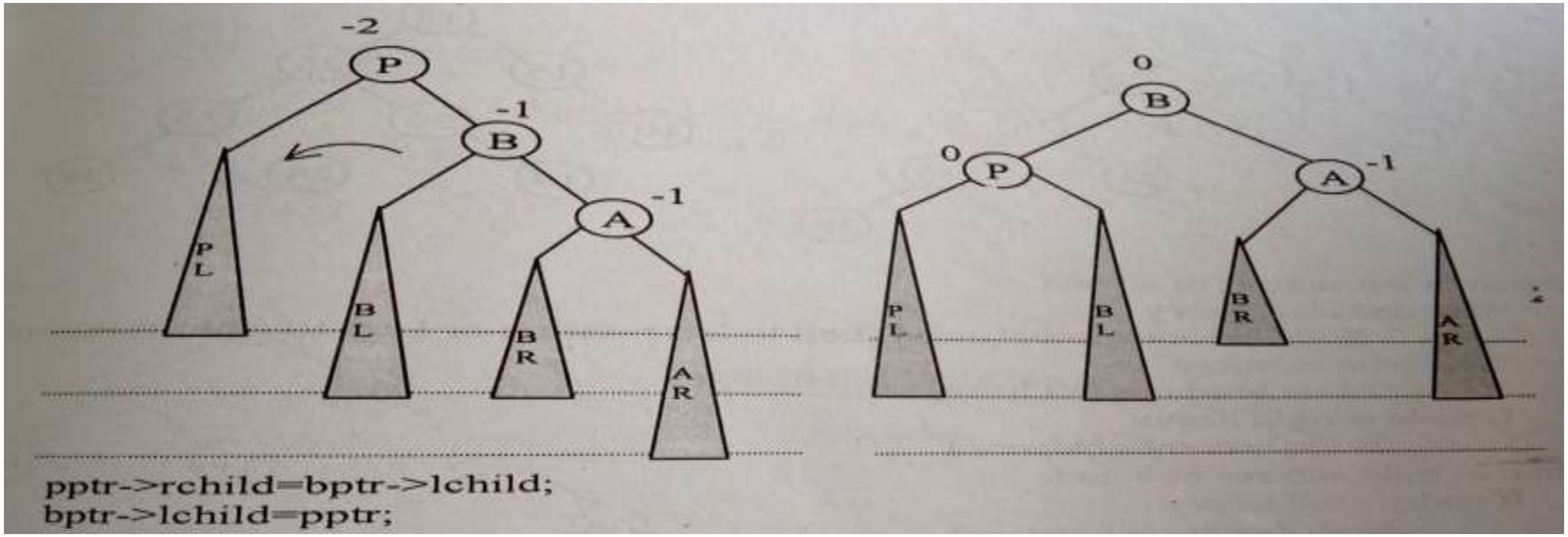
Child to Subtree Relationship(Of Pivot Node)

- **Right to Left Rotation**
  - Double Rotation needed
  - **First Perform Left to Left Rotation around node A**



AVL Rotations–

- **Right to Left Rotation**
  - **Then Right to Right rotation around pivot node P**
  - Mirror image of Left to Right Rotation

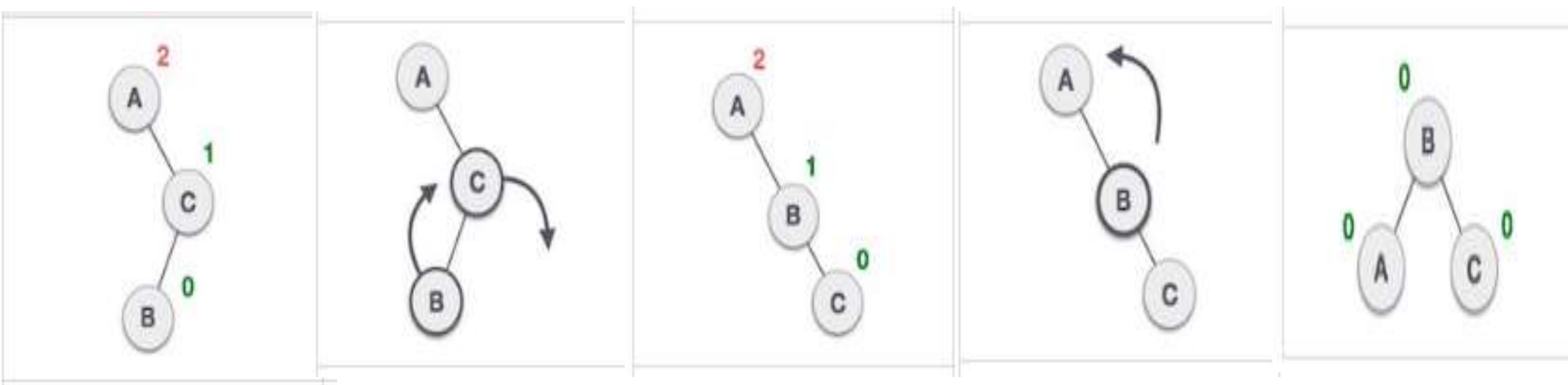


AVL Rotations–

- **Right to Left Rotation**
  - **First Perform Left to Left Rotation around node A=>Clockwise**
  - **Then Right to Right rotation around pivot node P=>Anti-Clockwise**

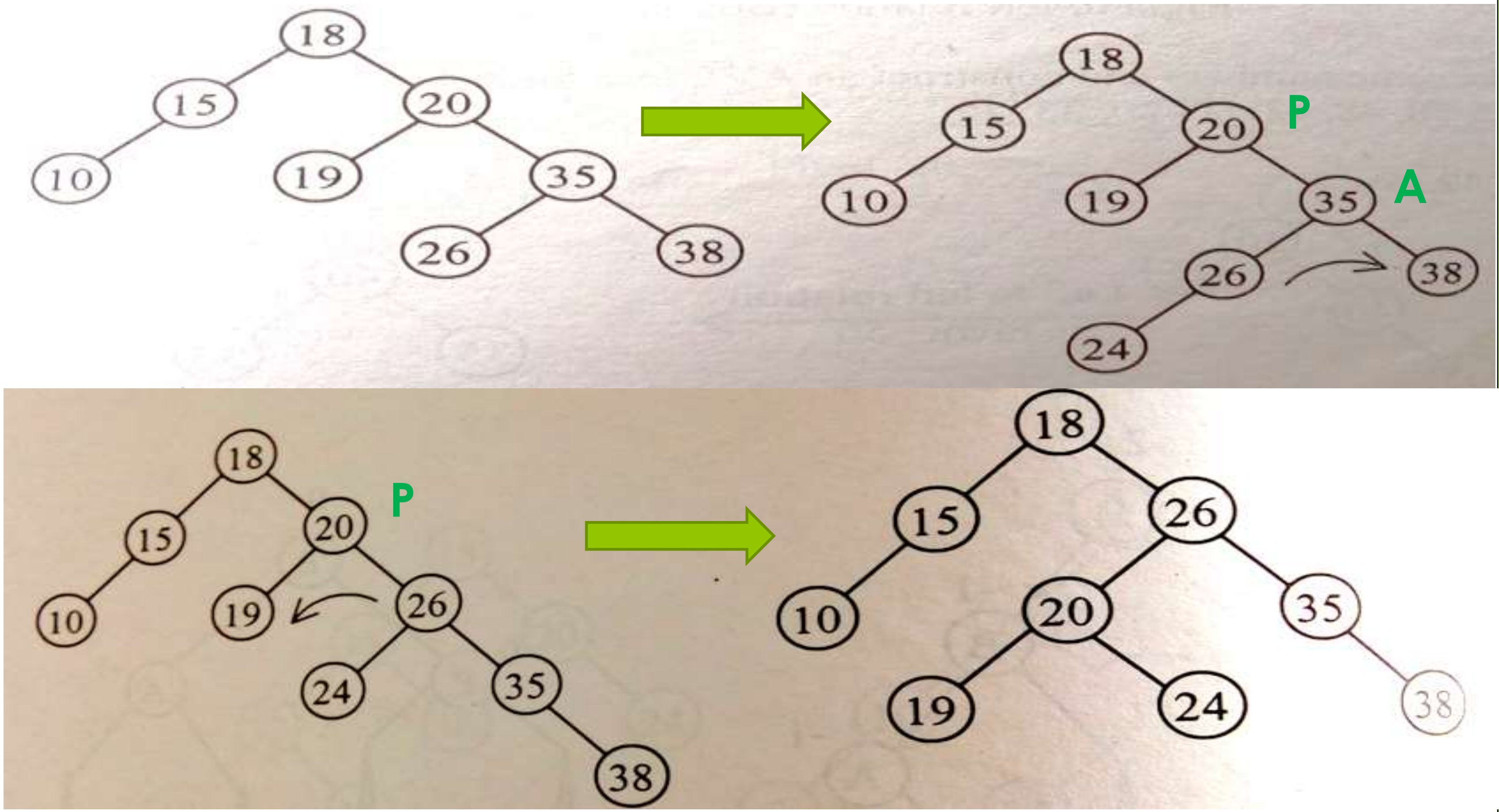
AVL Rotations–

- **Right to Left Rotation-Example**
  - First Perform Left to Left Rotation around node A=>Clockwise
  - Then Right to Right rotation around pivot node P=>Anti-Clockwise



AVL Rotations–

- **Right to Left Rotation**
  - Example



First Perform Left to Left Rotation around node A=>Clockwise
Then Right to Right rotation around pivot node P=>Anti-Clockwise