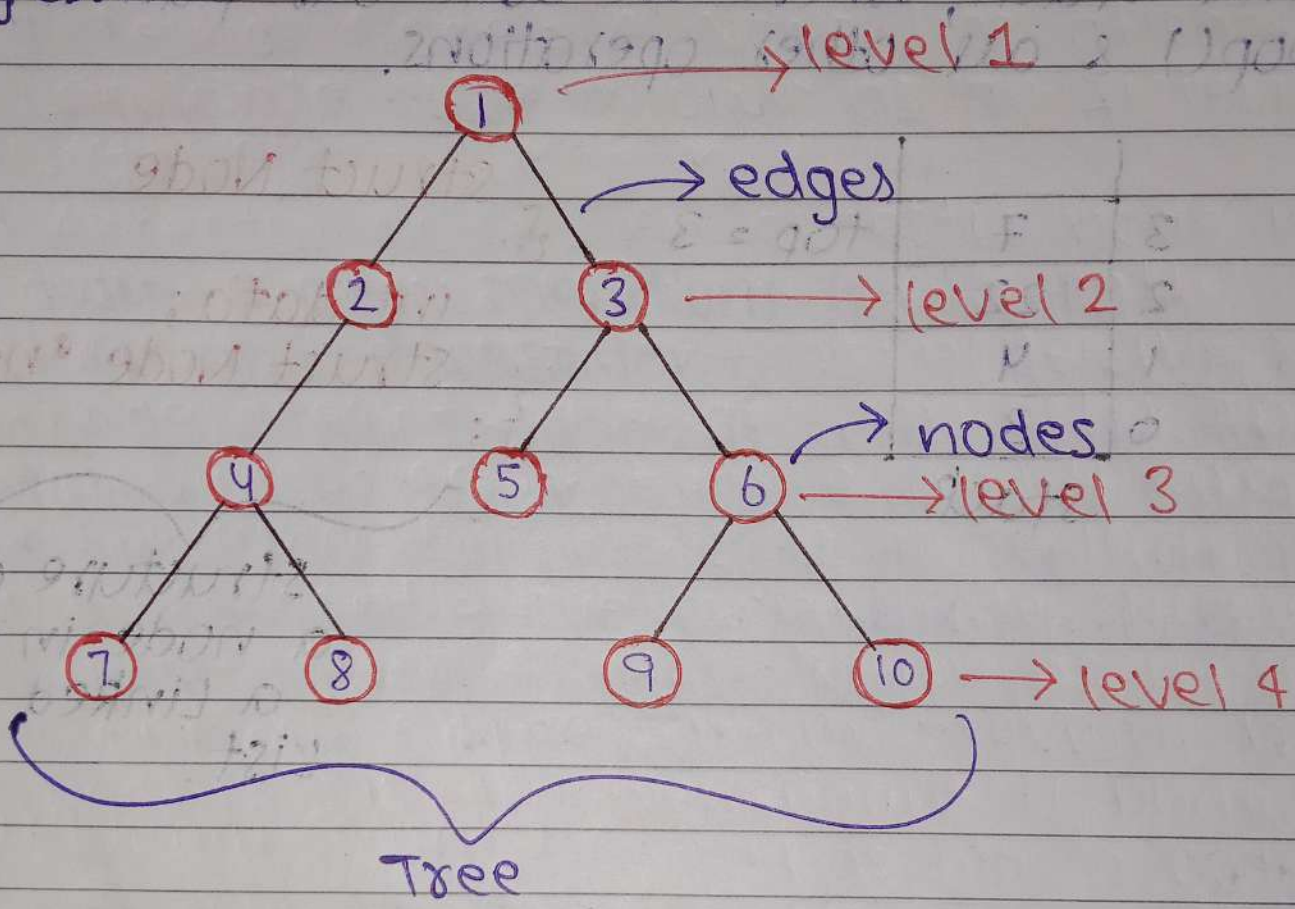
Trees

Tree represents the nodes connected by edges.



Other Data structures, like :- arrays, linked list, stack and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size, which is not acceptable in today's computational world.
Different tree data structures allow quicker & easier access to the data as it is a non-linear data structure.

* Some keypoints of the tree data structure
i) A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
ii) A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierar-chical structure as elements in a Tree are arranged in multiple levels.
iii) In the Tree Data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type.
iv) Each node contains some data & the link or reference of other nodes that can be called children.

* Terminologies in Tree Data Structure :-

i) Root = The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. If a node is directly linked to some other node, it would be called a Parent-child Relationship.
ii) Child Node = If the node is a descendant of any node, then the node is known as child node.
iii) Parent = If the node contains any sub-node, then that node is said to be the parent of that sub-node.
iv) Sibling = The nodes that have the same parent are known as sibling.

v) Leaf Node = The node of the tree which doesn't have any child node, is called a Leaf Node. A leaf node is the bottom node of the tree. There can be any number of Leaf Nodes present in a general tree. Leaf Nodes can also be called External Nodes.

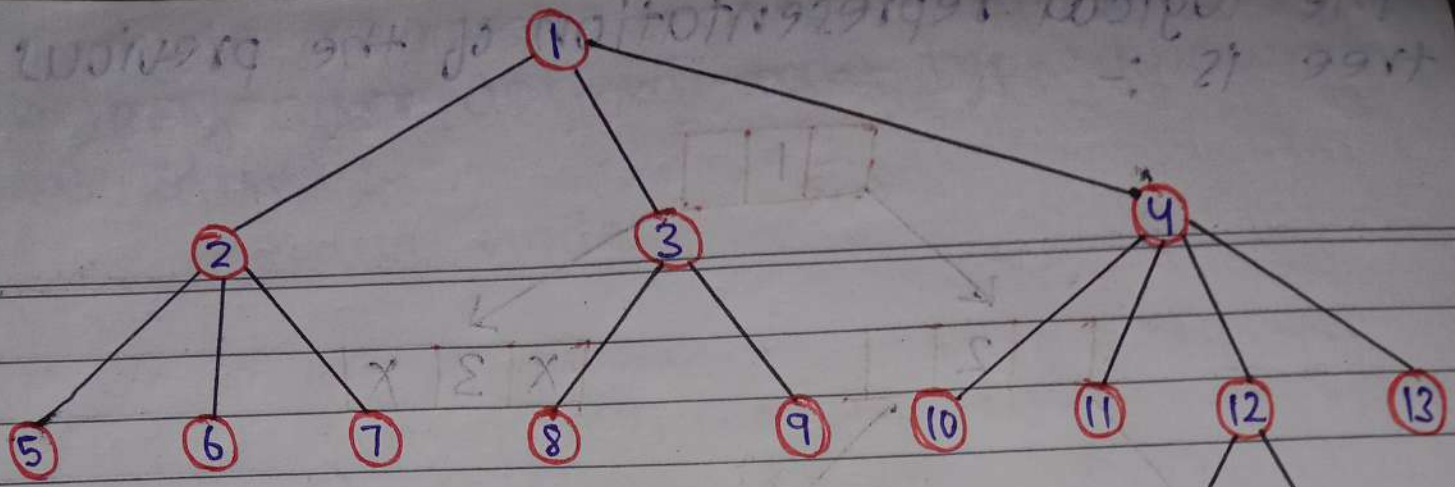
vi) Internal Node = A Node has atleast one child node known as an Internal Node.

vii) Ancestor Node = An ancestor node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestor. In our previous tree, node 1, 2 and 4 are the ancestor nodes of node 10.

viii) Descendant = The immediate successor of the given node is known as descend-ant of a node. In our previous tree, node 8 is the descendant of node 4.

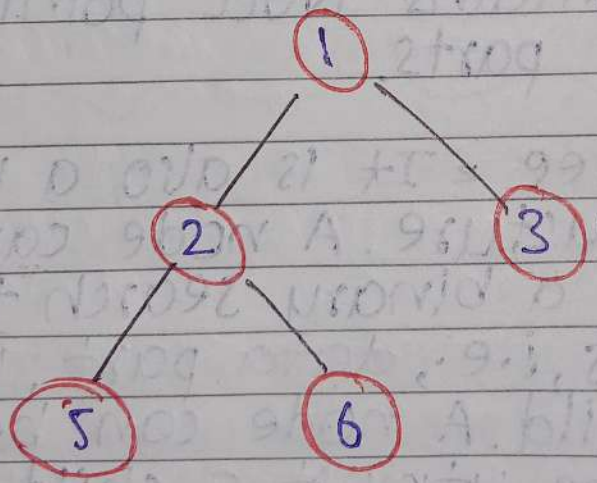
* Types of Tree Data Structures :-

i) General Tree = The General Tree is one of the types of tree data structure. In this, a node can have either 0 or maximum n number of nodes. There is no restrictions imposed on the degree of the node.



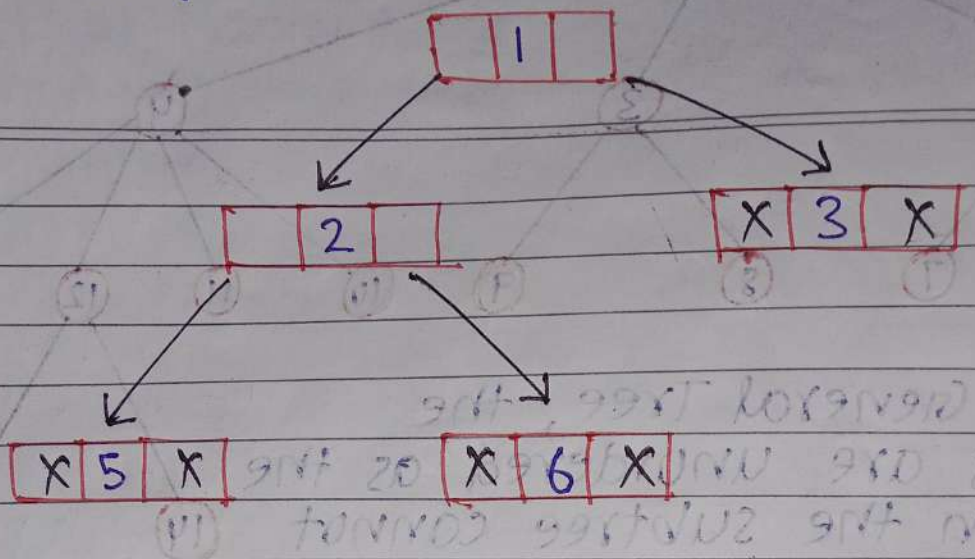
* In the General Tree, the subtrees are unordered as the nodes in the subtree cannot be ordered.

(ii) Binary Tree = The Binary Tree means that the node can have maximum 2 childrens. Each node can have either 0, 1 or 2 childrens.



This is a binary tree because each node contains the utmost 2 childrens.

The logical representation of the previous tree is :-

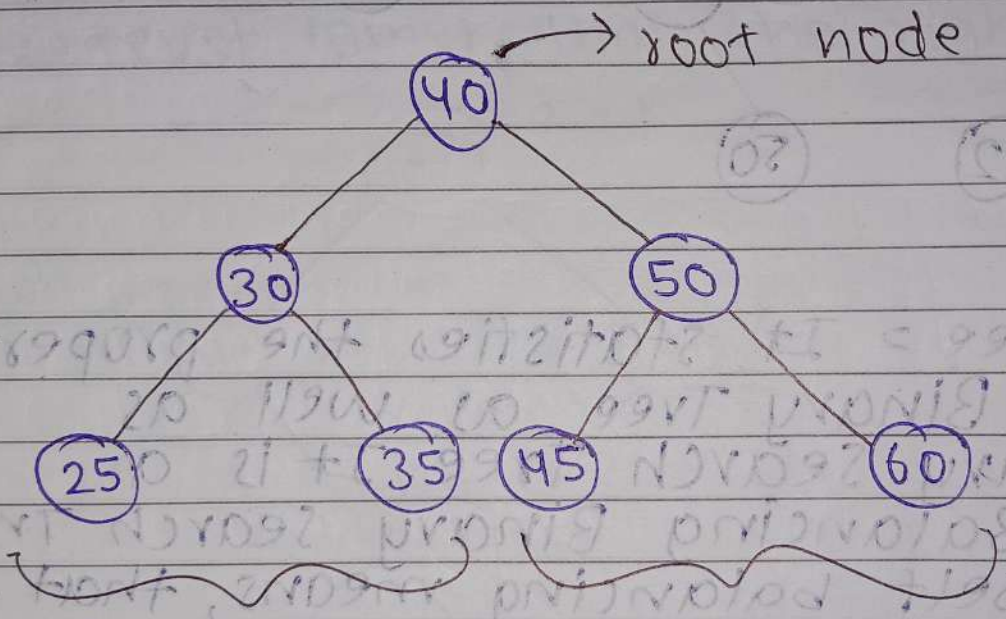


In the above tree, node 1. contains two pointers, i.e., left and a right pointer pointi -ng to the left and right node respectively. The node 2 contains both the nodes (left and right), therefore, it has two pointers. The nodes 3, 5 and 6 are the leaf nodes, so all these nodes contains NULL pointer on both left and right parts.

(iii) Binary Search Tree = It is also a node based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left child & right child. A node can be connected to the utmost 2 child, so the node contains two pointers. Every node in the left sub-tree must contain a value less than the value of the root node & the value of each node in the right-subtree must be bigger than the value of the root node.

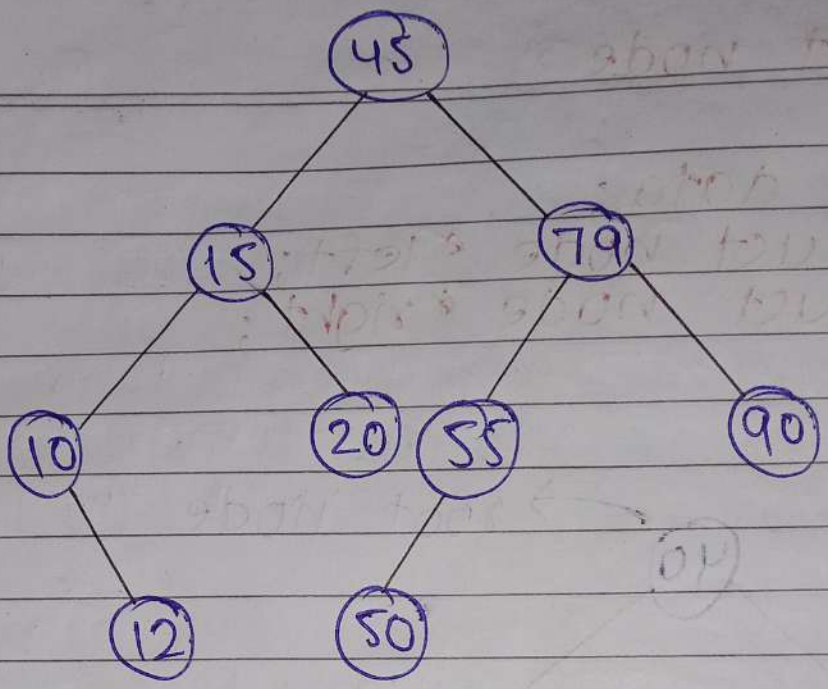
A node can be created with the help of user defined data type known as struct :-

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



* Advantages of Binary Search Tree (BST) :-
  i) Searching an element in BST is easy as we always have a hint that which sub-tree has the desired element.
  ii) As compare to Array and Linked List, insertion & deletion operations are faster in BST.

Eg:- create a Binary search Tree :-
45, 15, 79, 90, 10, 55, 12, 20, 50

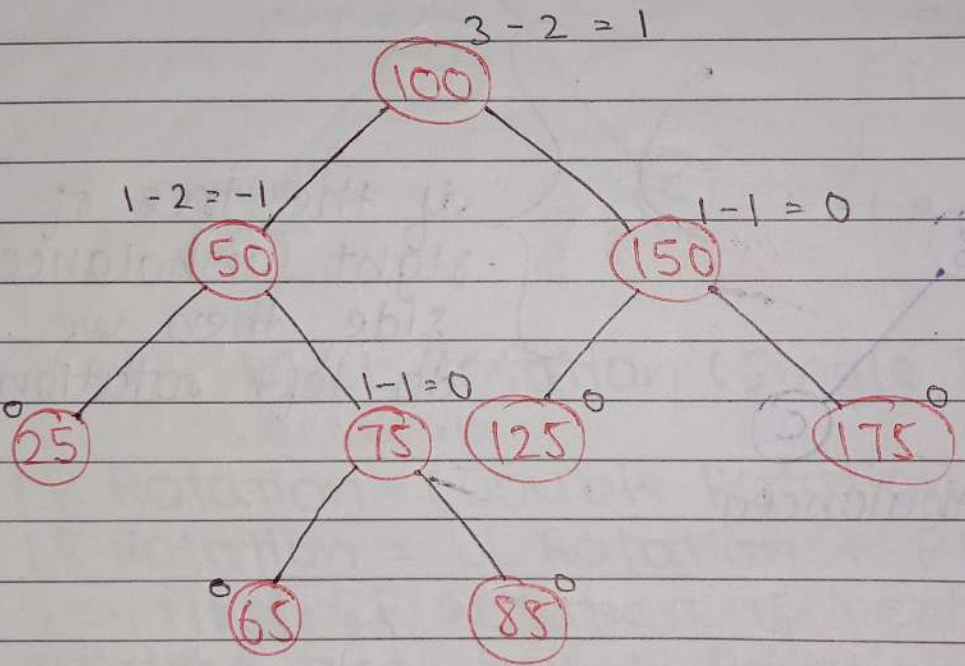


iv) AVL Tree = It statisfies the property of the Binary Tree as well as Binary search Tree. It is a self Balancing Binary search Tree. Here, self balancing means, that balancing the heights of left subtree and right subtree. This balancing is measured in terms of the Balancing Factor. The balancing factor can be defined as the difference between the height of the left sub-tree & the height of the right sub-tree. The value of balancing factor must be 0, -1 or +1, therefore, each node in the AVL tree should have the value of the balancing factor either as 0, -1 or +1.

If the balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

If the balance factor of any node is 0, it means that the left sub-tree & right sub-tree contains equal height.

If the balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.



* Rotations in AVL Tree :-
We perform rotation in AVL tree only in case if Balance Factor is other than -1, 0, 1.
There are basically 4 types of rotations in AVL trees :-

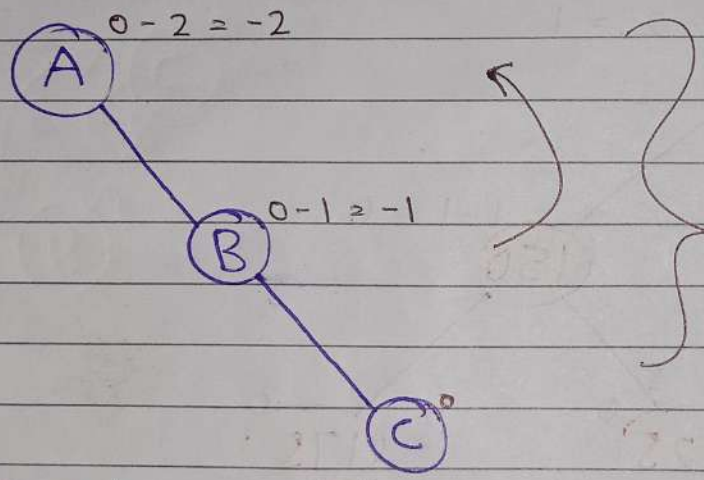
i) LL Rotation (Left Left) = Inserted node is in the left subtree of left subtree of A.

ii) RR Rotation = Inserted node is in the right sub-tree of right sub-tree of A.

iii) LR Rotation = Inserted node is in the right sub-tree of left subtree of A.

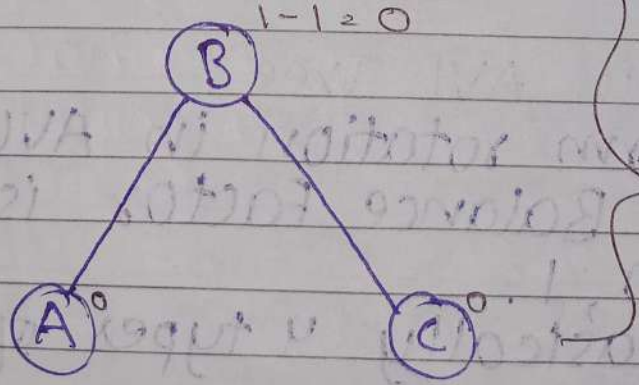
iv) RL Rotation = Inserted node is in the left sub-tree of right subtree of A.

Note:- Node A is the node whose balancing factor is other than -1, 0, 1.



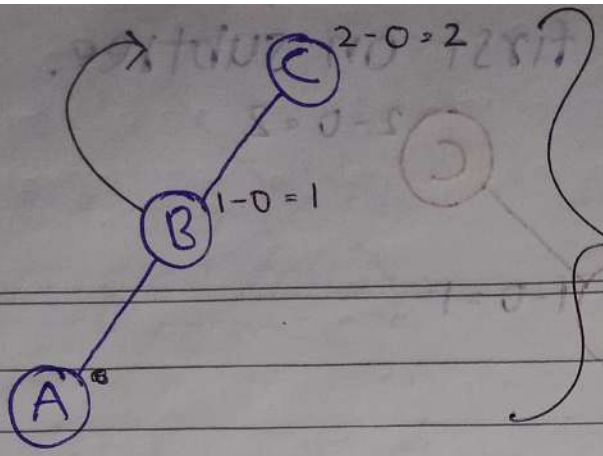
if the tree of right unbalanced side, then we do left rotation.

* Right Unbalanced Tree



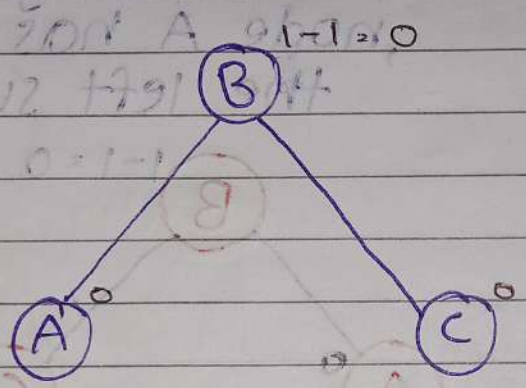
Now, the tree is Balanced.

a) RR Rotation (Single Rotation)



if the tree is left unbalanced, then we do right rotation.

* Left Unbalanced Tree

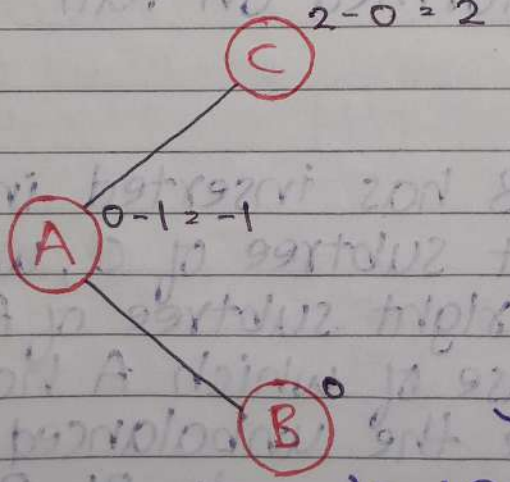


Now, the tree is Balanced Tree.

b) LL Rotation (Single Rotation)

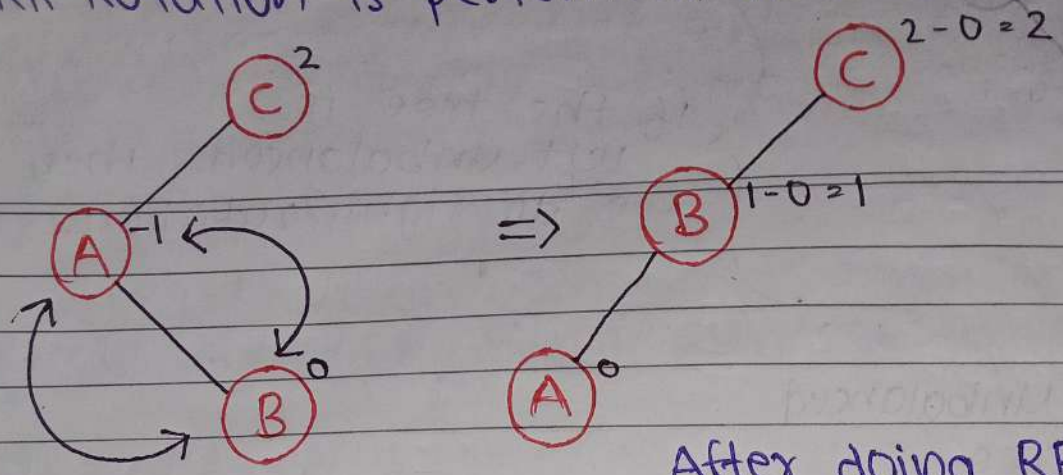
c) LR Rotation = (Double Rotation)

LR Rotation = LL Rotation + RR Rotation, i.e., first RR Rotation is performed on subtree and then LL Rotation is perform-ed on full tree.

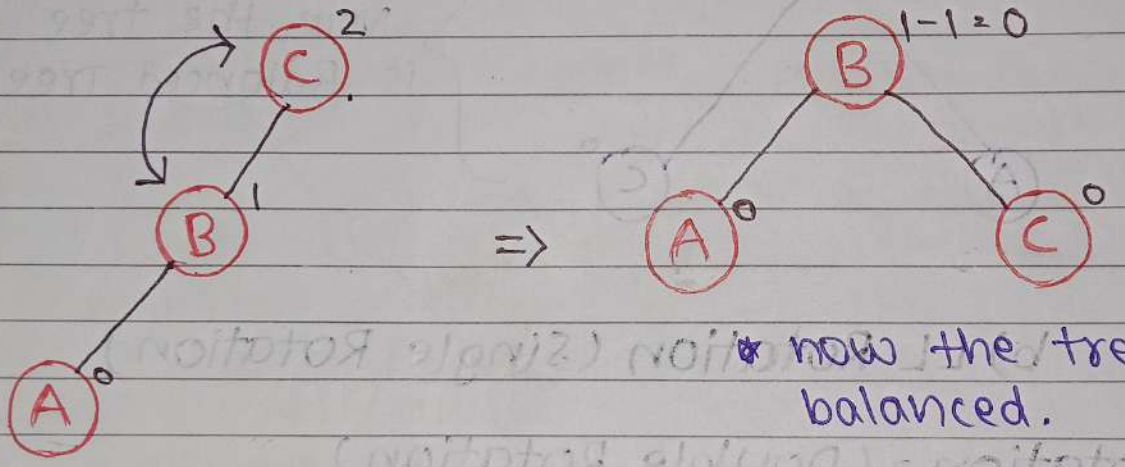


Node B has inserted into the right subtree of A which is a right subtree of C, because of which C has become the unbalanced node. Now, we do LR Rotation.

RR Rotation is performed first on subtree.



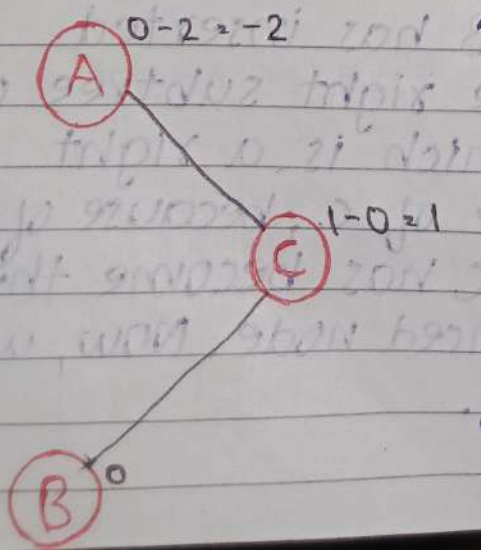
After doing RR Rotation, node A has become the left subtree of B.



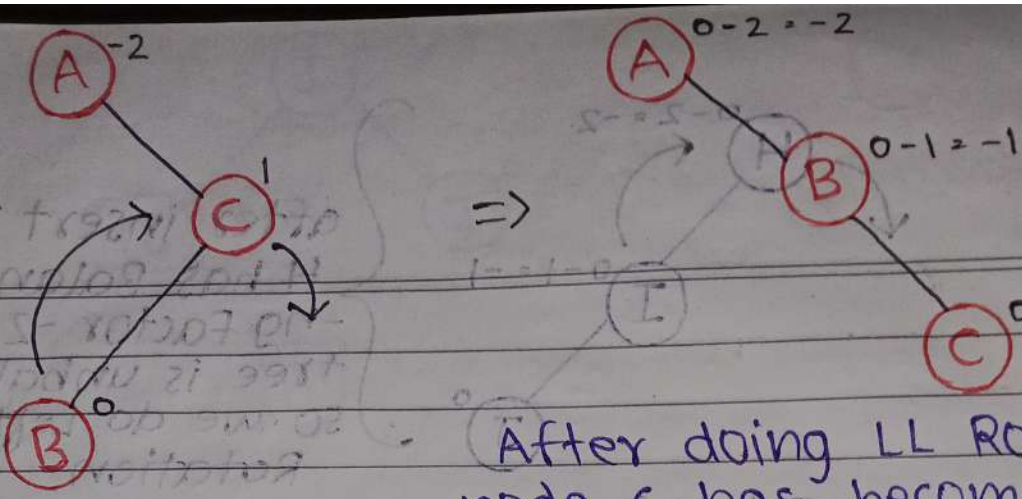
* now the tree is balanced.

d) RL Rotation = (Double Rotation)

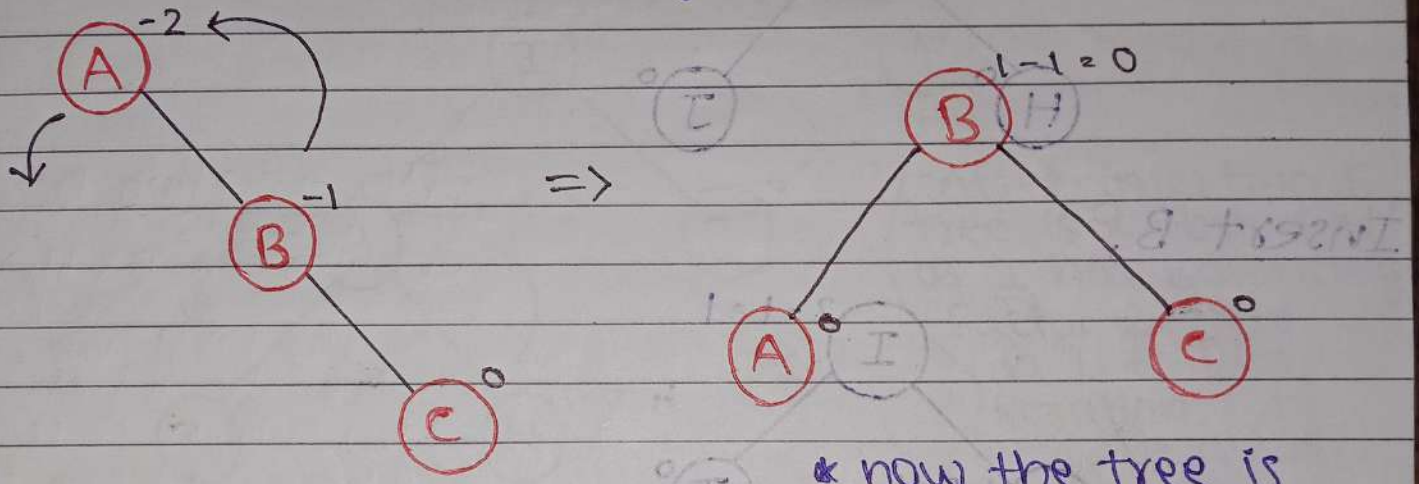
RL Rotation = RR Rotation + LL Rotation, first LL Rotation is performed on subtree then RR Rotation is performed on full tree.



Node B has inserted into the left subtree of C, which is a right subtree of A, because of which A has become the unbalanced node. Now, we do RL Rotation.



After doing LL Rotation, node C has become the right subtree of B.

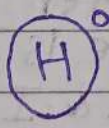


* now the tree is balanced.

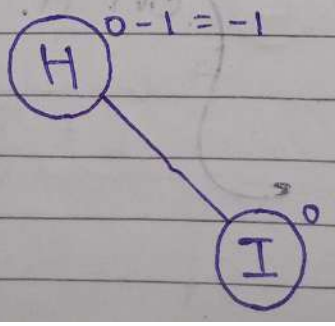
X X X X X

Eg:- construct AVL Tree :-
H, I, J, B, A, E, C, F, D, G, K, L

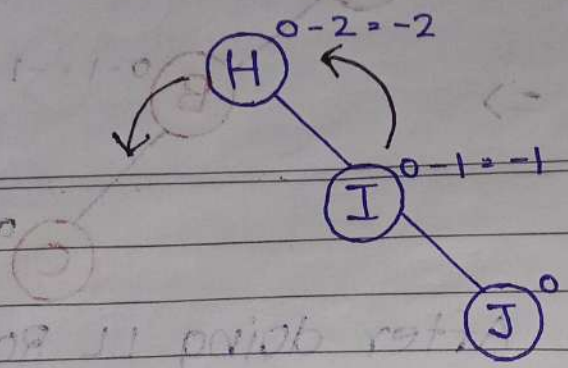
Insert H,



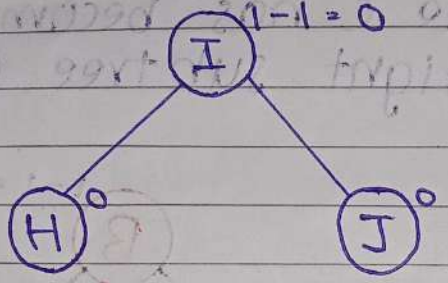
Insert I,



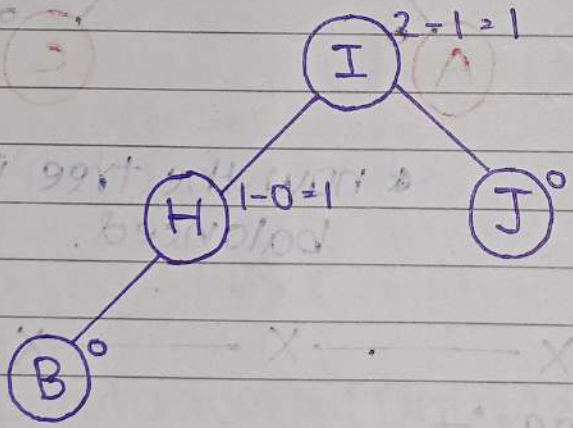
Insert J,



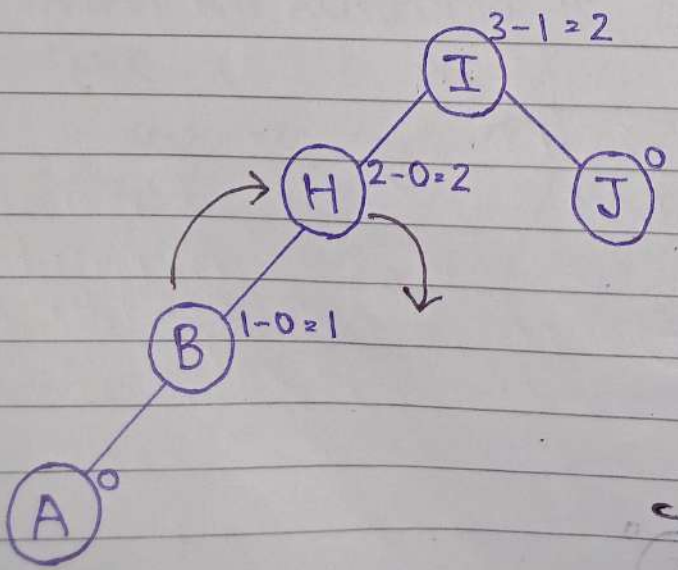
after insert J, H has Balanci -ng Factor -2, tree is unbalance so we do Left Rotation.



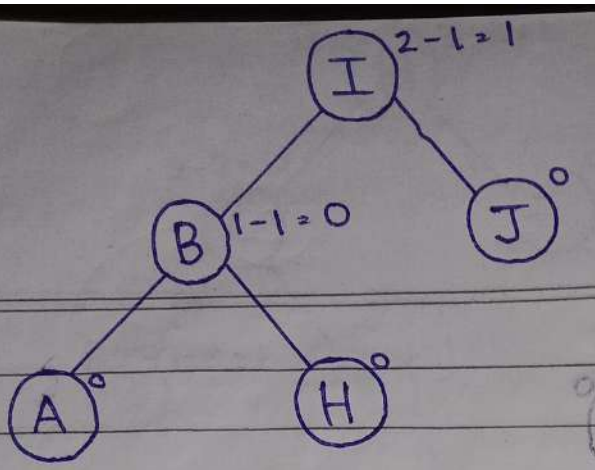
Insert B,



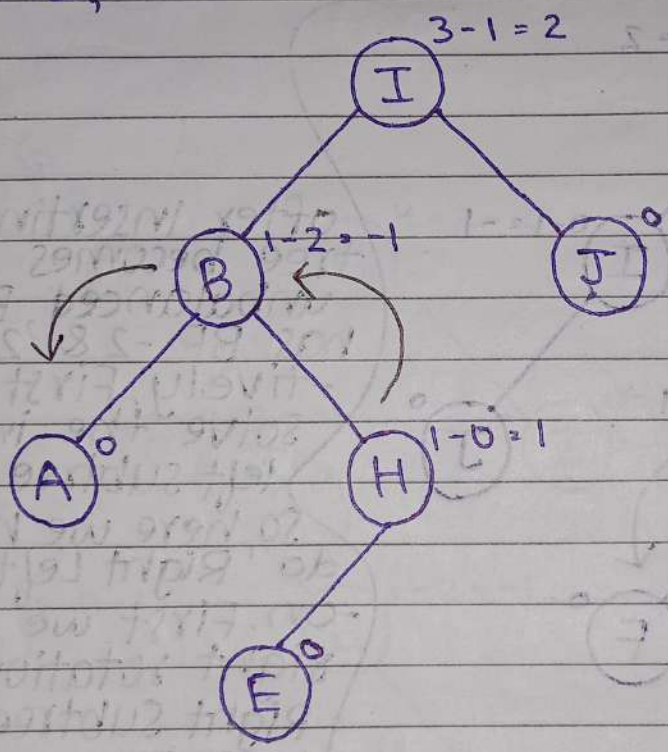
Insert A,



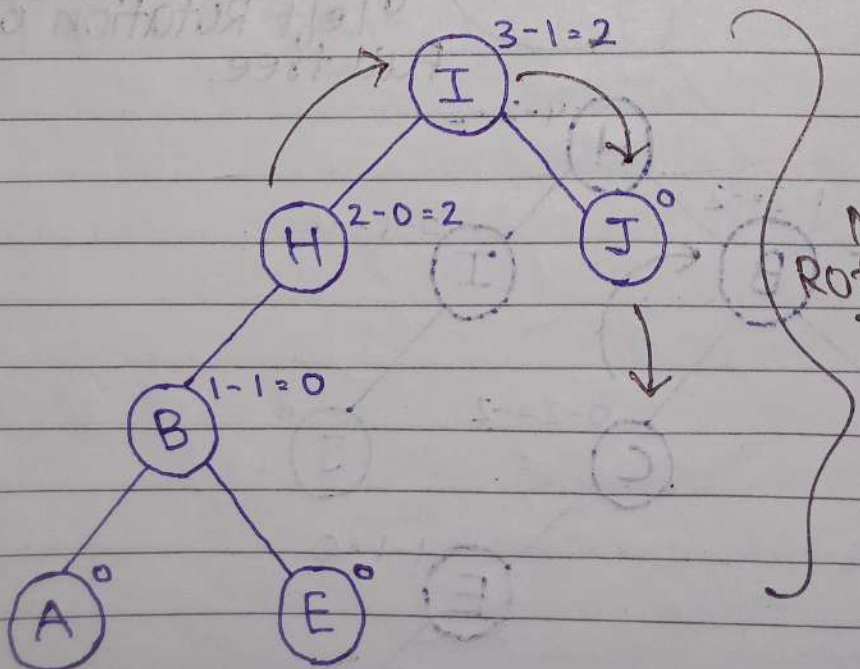
after inserting A, tree is unbalanced as H & I has Balancing Factor 2. we do right rotation on H.



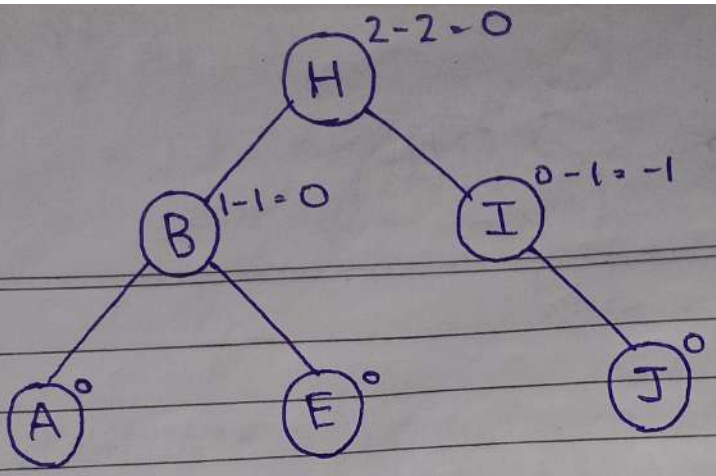
Insert E,



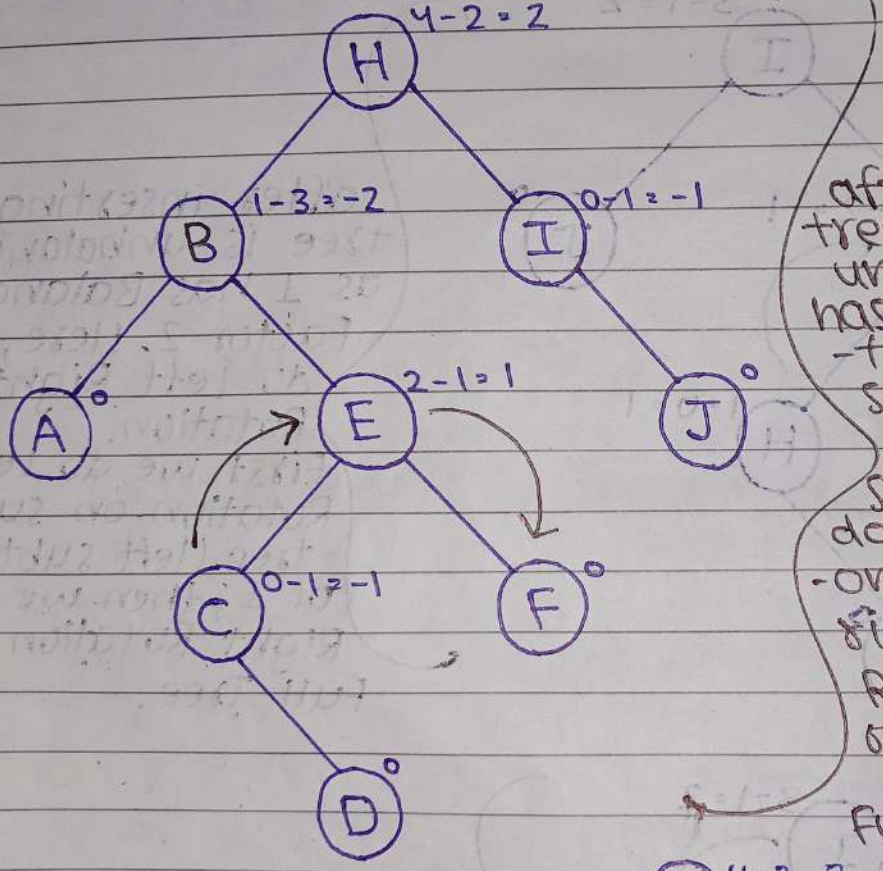
after inserting E, tree is unbalanced as I has Balancing Factor 2. Here, we do Left Right Rotation. First we do Left Rotation on sub-tree (left subtree) of I, then we do Right Rotation on Full Tree.



Now we do Right Rotation on Full tree.

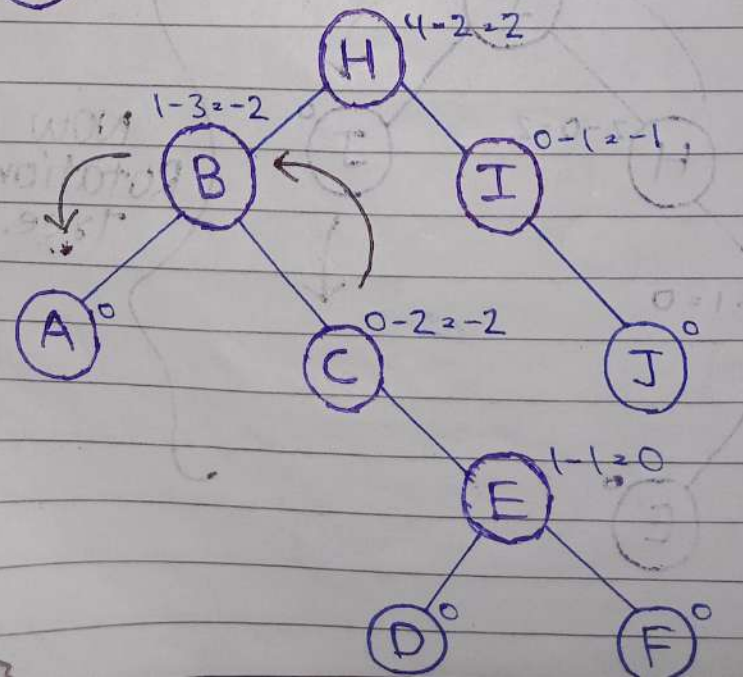


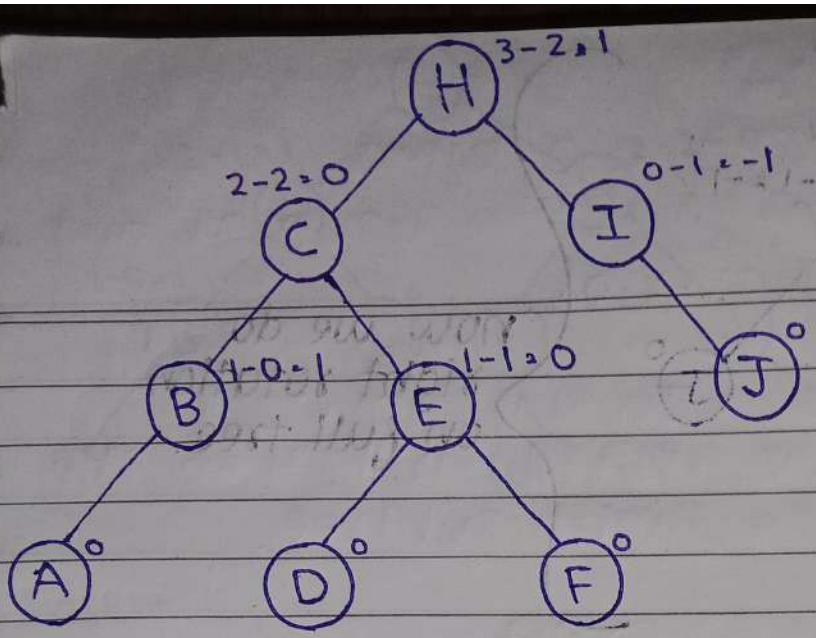
Insert C, F, D,



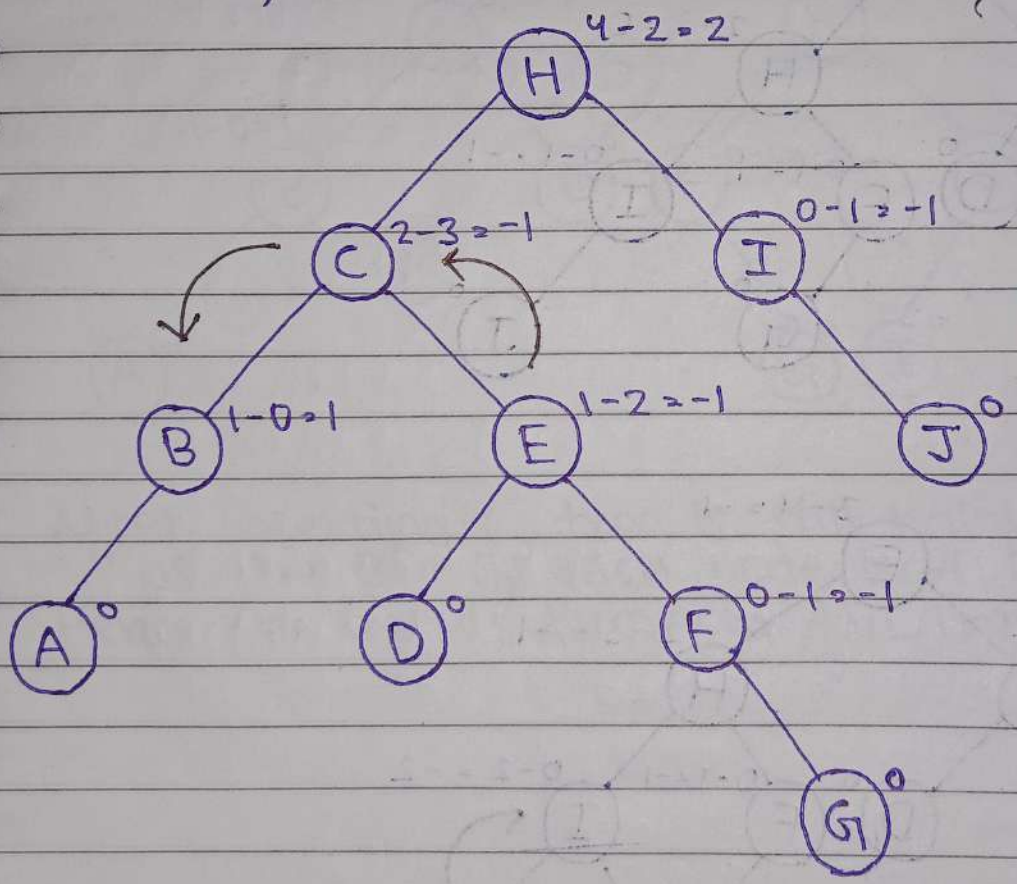
after inserting D, tree becomes unbalanced, B & H has BF -2 & 2 respec -tively. First, we solve the inner left subtree of H. So, here we have to do Right Left Rotati -on. First we do right rotation of Right Subtree of B. Then we do Left Rotation on full tree.

now we do full left rotation on left subtree of H.

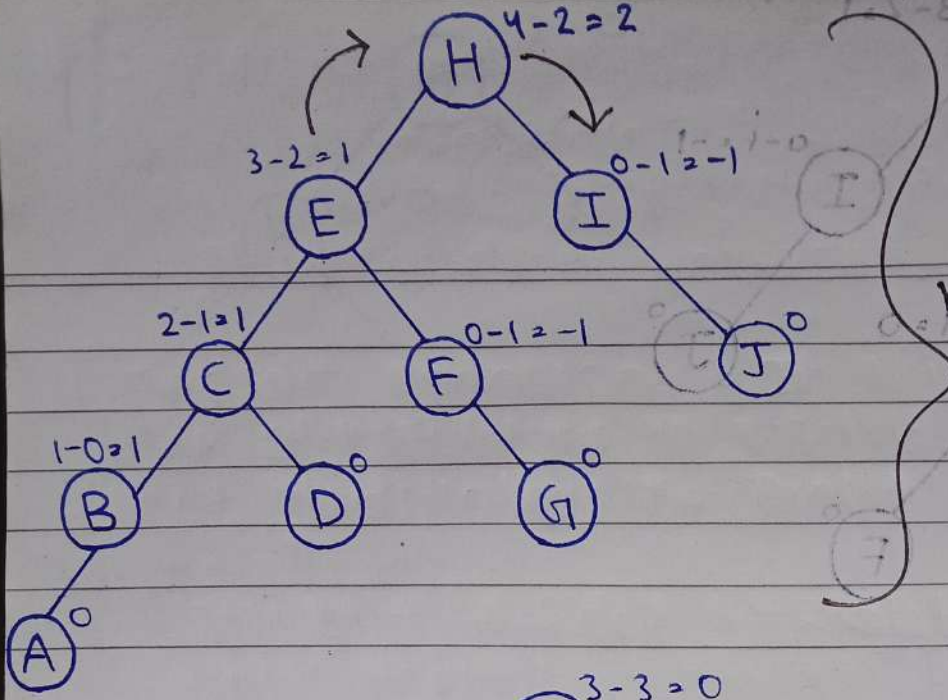




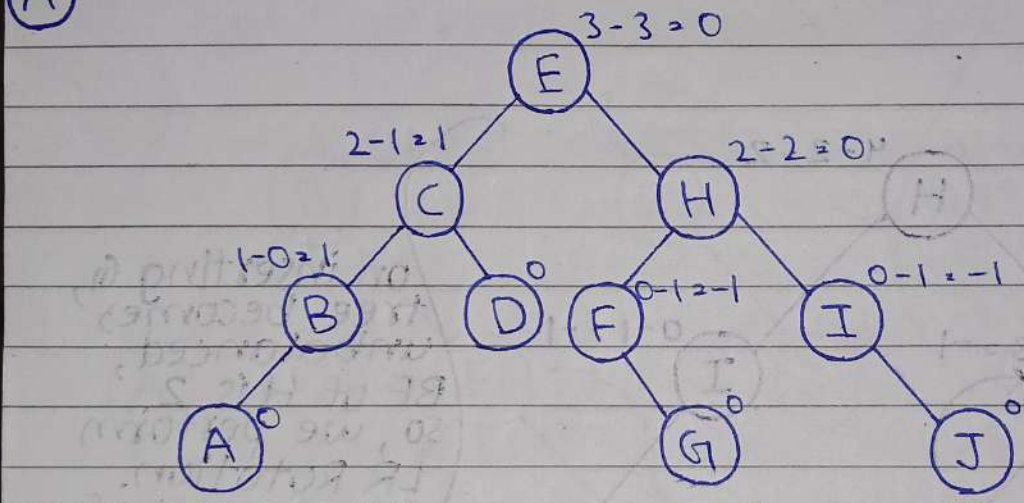
Insert G,



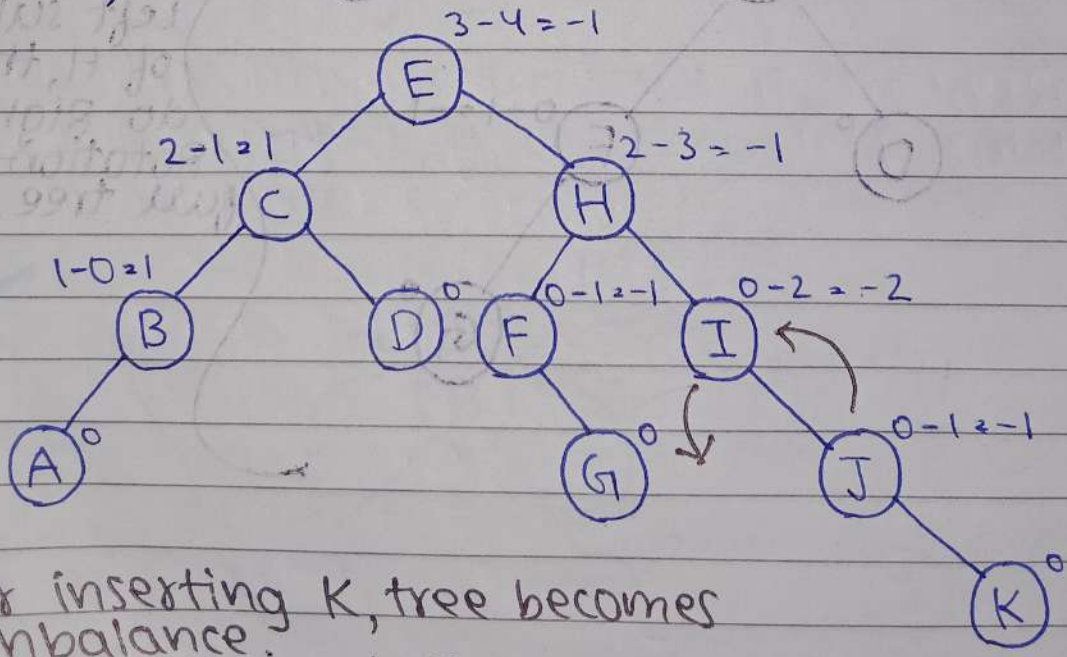
on inserting G, tree becomes unbalanced, BF of H is 2. So, we perform LR Rotation. First we perform Left Rotation on Left subtree of H, then we do Right Rotation on full tree.



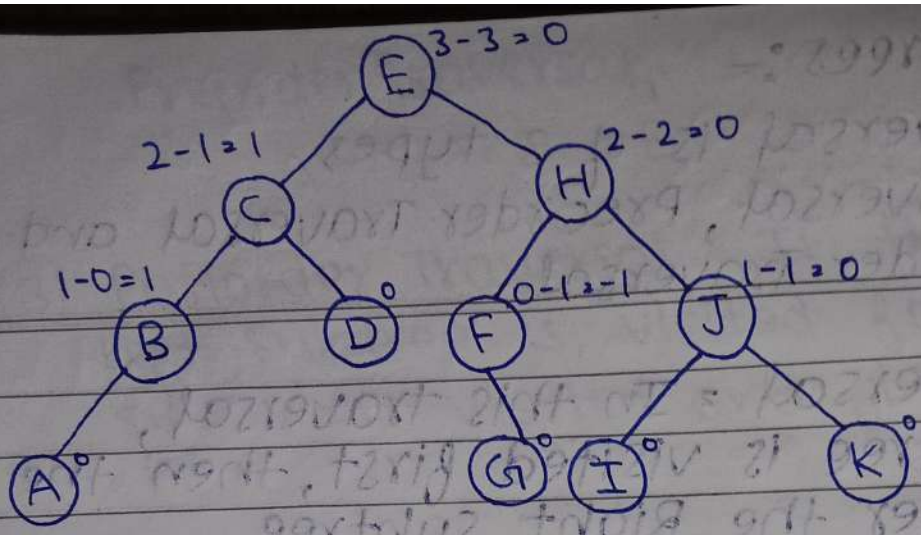
now we do right rotation on full tree.



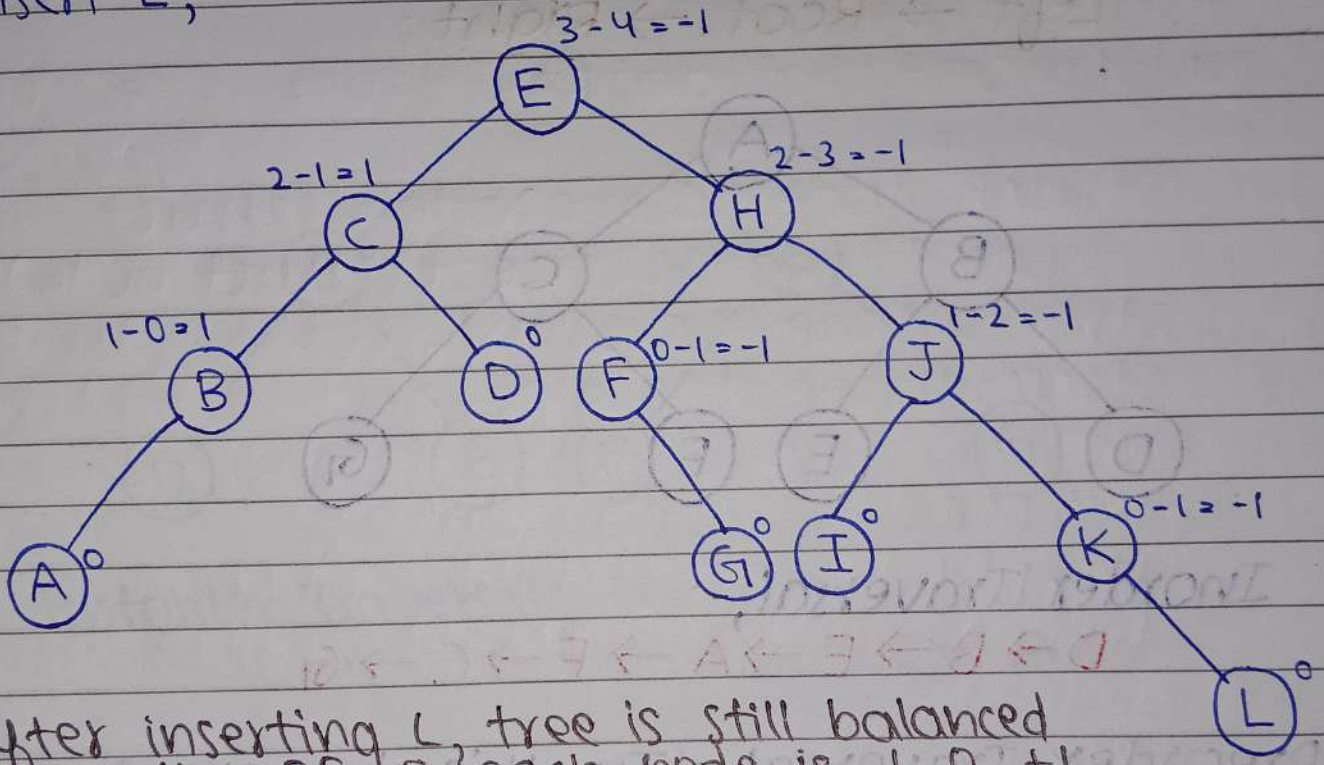
Insert K,



after inserting K, tree becomes unbalance.
so, we do left rotation on right subtree of I.



Insert L,



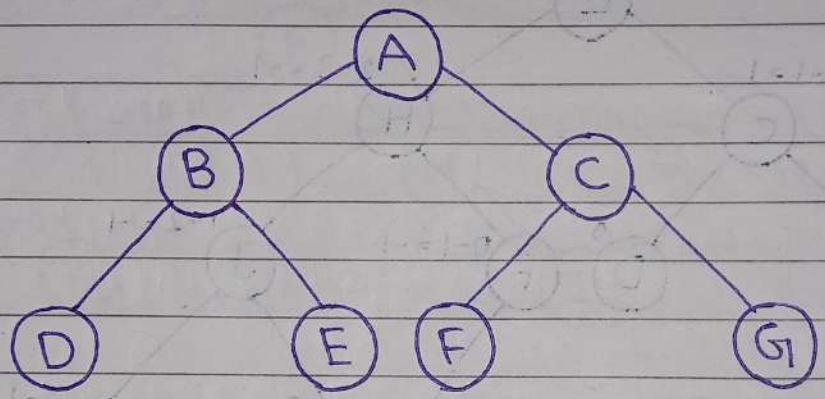
After inserting L, tree is still balanced as the BF of each node is -1, 0, +1.
Hence, the tree is Balanced AVL Tree.

Traversal in Trees :-

In Trees, traversal is of 3 types Inorder Traversal, Preorder Traversal and Postorder Traversal.

(1) Inorder Traversal = In this traversal, the Left subtree is visited first, then the Root and later the Right subtree.

Left → Root → Right

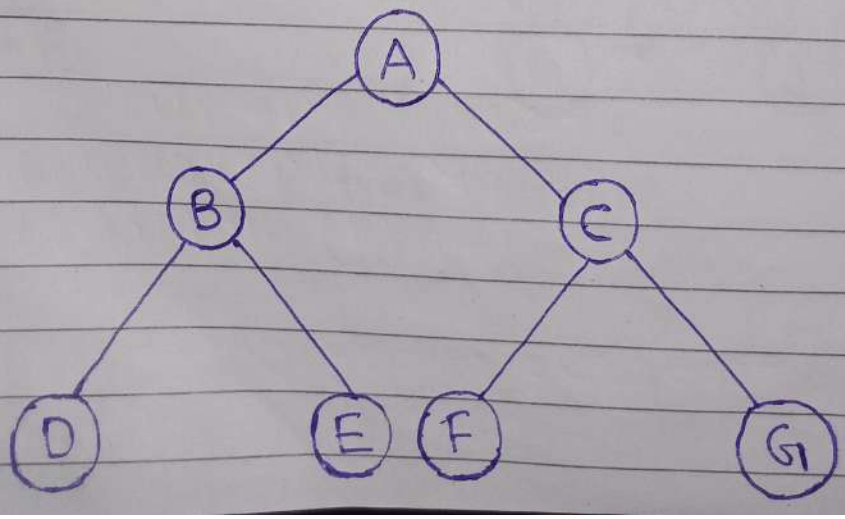


Inorder Traversal,

D → B → E → A → F → C → G

(2) Preorder Traversal = In this traversal, Root is visited first, then left subtree & finally right subtree.

Root → Left → Right

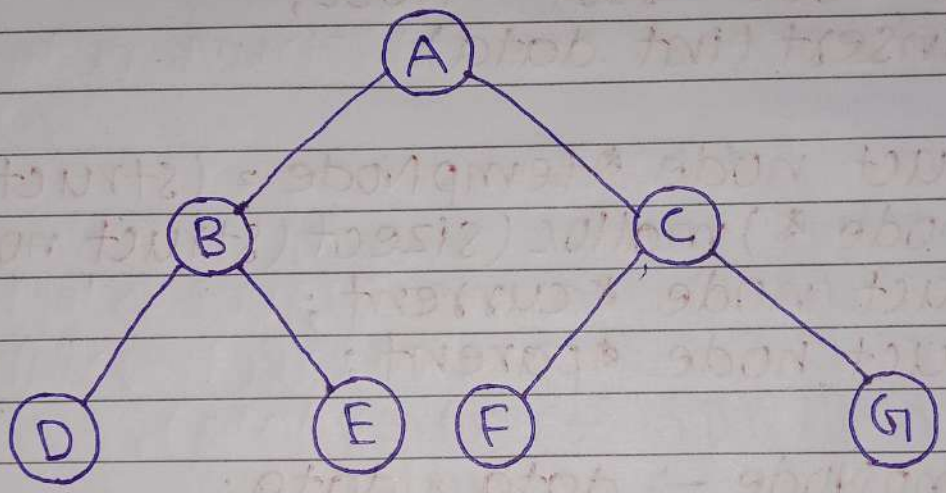


Preorder Traversal,

A → B → D → E → C → F → G

③ Postorder Traversal = In this traversal, Left subtree is visited first, then right subtree is visited and finally the ROOT node

Left → Right → Root



Postorder Traversal,

D → E → B → F → G → C → A

Inorder_Preorder_Postorder Program.

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *leftChild;
    struct node *rightChild;
};
struct node *root = NULL;
void insert(int data)
{
    struct node *tempNode = (struct
      node *)malloc(sizeof(struct node));
    struct node *current;
    struct node *parent;

    tempNode -> data = data;
    tempNode -> leftChild = NULL;
    tempNode -> rightChild = NULL;

    //if tree is empty.
    if(root == NULL)
    {
        root = tempNode;
    }
    else
    {
        current = root;
        parent = NULL;
        while(1)
        {
            parent = current;
```

```
            //go to the left of the tree.
            if(data < parent->data)
            {
                current = current->leftchild;
                //insert to the left.
                if(current == NULL)
                {
                    parent->leftchild = tempNode
                    return;
                }
            }
            else
            {
                current = current->rightchild;
                //insert to the right.
                if(current == NULL);
                {
                    parent->rightchild = tempNode;
                    return;
                }
            }
        }
    }
}
void pre_order_traversal(struct node *root)
{
    if(root != NULL)
    {
        printf("%d", root->data);
        pre_order_traversal(root->leftchild);
        pre_order_traversal(root->rightchild);
    }
}
```

```
void in_order_traversal(struct node
                        *root)
{
    if(root != NULL)
    {
        in_order_traversal(root -> leftchild);
        printf("%d ", root -> data);
        inorder_traversal(root -> rightchild);
    }
}

void post_order_traversal(struct node *root)
{
    if(root != NULL)
    {
        post_order_traversal(root -> leftchild);
        post_order_traversal(root -> rightchild);
        printf("%d ", root -> data);
    }
}

int main()
{
    int i;
    int array[7] = {27, 14, 35, 10, 19, 31, 42};
    for(i = 0; i < 7; i++)
    {
        insert(array[i]);
    }

    printf("Preorder : ");
    pre_order_traversal(root);

    printf("Inorder : ");
    in_order_traversal(root);
```

```
    printf("Postorder: ");
    post_order_traversal(root);
    printf("\n");
    return 0;
}
```

* Convert expression from Infix to Postfix using stack.

```
#include <stdio.h>
#include <ctype.h>
char stack[100];
int top = -1;
void push(char x)
{
    stack[++top] = x;
}
char pop()
{
    if(top == -1)
    {
        return -1;
    }
    else
    {
        return stack[top--];
    }
}
int priority(char x)
{
    if(x == '(')
    {
        return 0;
    }
```

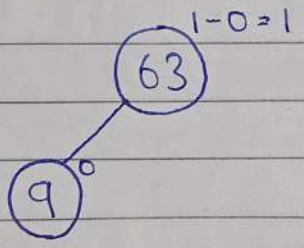
```
    if(x == '+' || x == '-')
    {
        return 1;
    }
    if(x == '*' || x == '/')
    {
        return 2;
    }
    return 0;
}
int main()
{
    char exp[100];
    char *e, x;
    printf("enter the expression: ");
    scanf("%s",exp);
    e = exp;
    while(*e != '\0')
    {
        if(isalnum(*e)){
            printf("%c ",*e);
        }
        elseif(*e == '('){
            push(*e);
        }
        elseif(*e == ')'){
            while((x = pop()) != '('){
                printf("%c",x);
            }
        }
        else{
            while(priority(stack[top]) >=
                                priority(*e)){
                printf("%c",pop());
            }
```

```
            push(*e);
        }
        e++;
    }
    while(top != -1)
    {
        printf("%c", pop());
    }
    return 0;
}
```

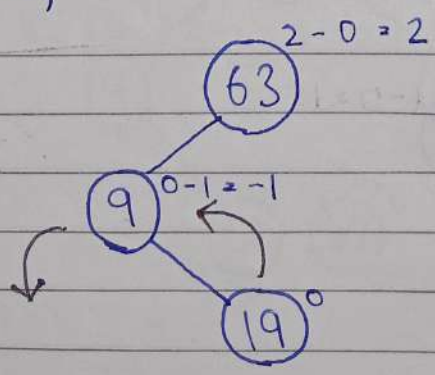
Eg :- Construct AVL Tree :-
63, 9, 19, 27, 18, 108, 99, 81

Insert 63,

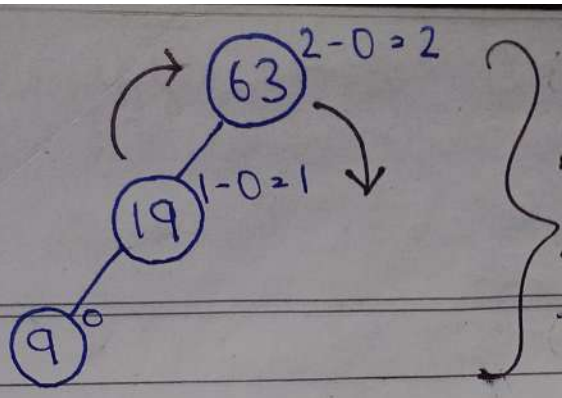
Insert 9,



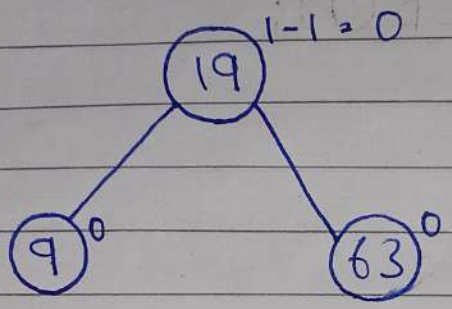
Insert 19,



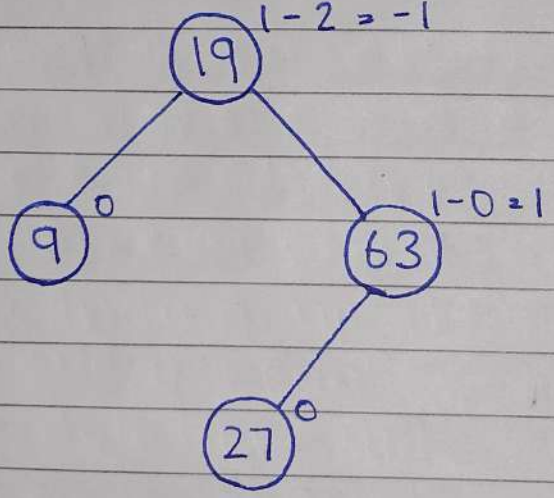
after inserting 19, the BF of 63 is 2, 19 is place at Left Right side of 63, so we do LR Rotation.
First we do Left Rotation on subtree the Right Rotation on full tree.



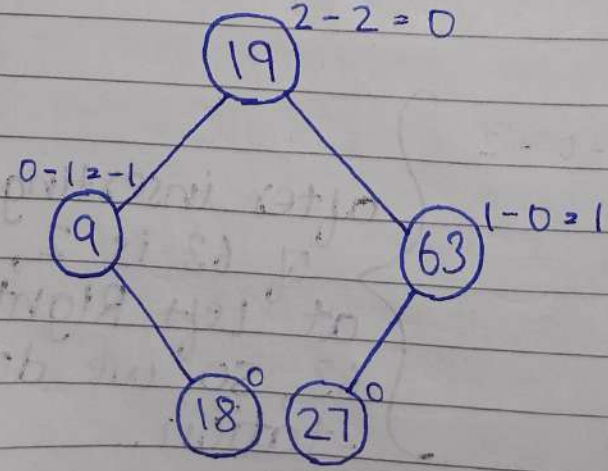
we have done left rotation on sub-tree, now we do right rotation on full tree.



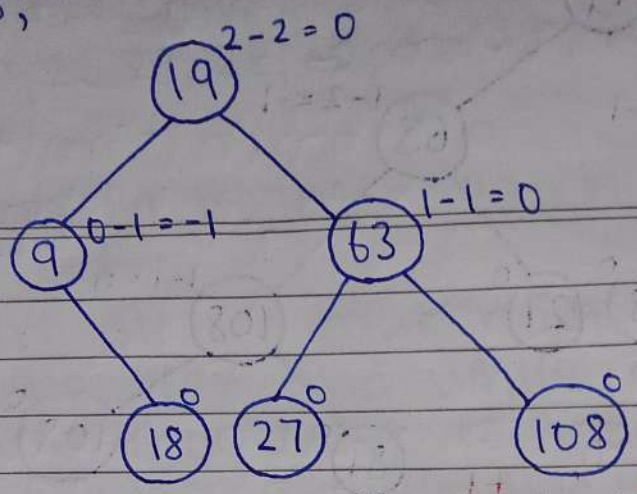
Insert 27,



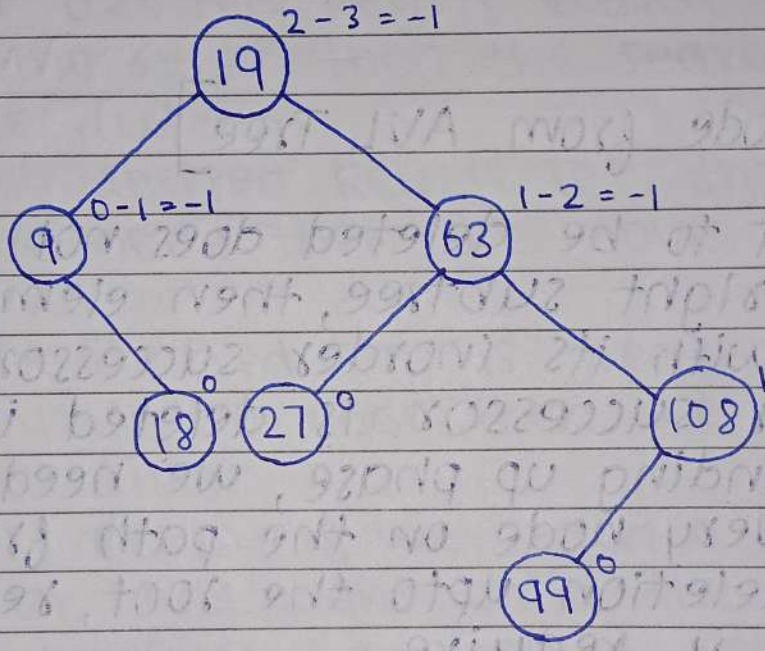
Insert 18,



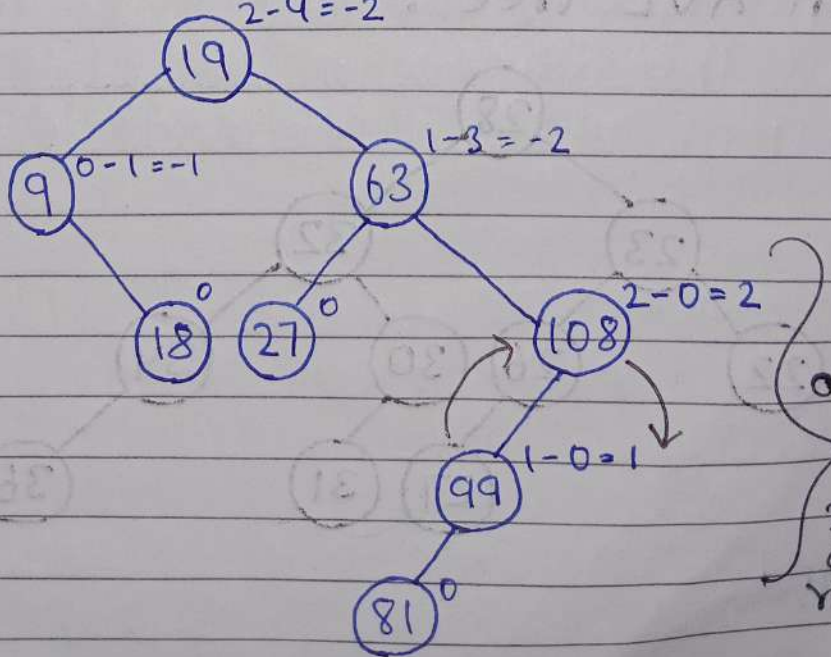
Insert 108,

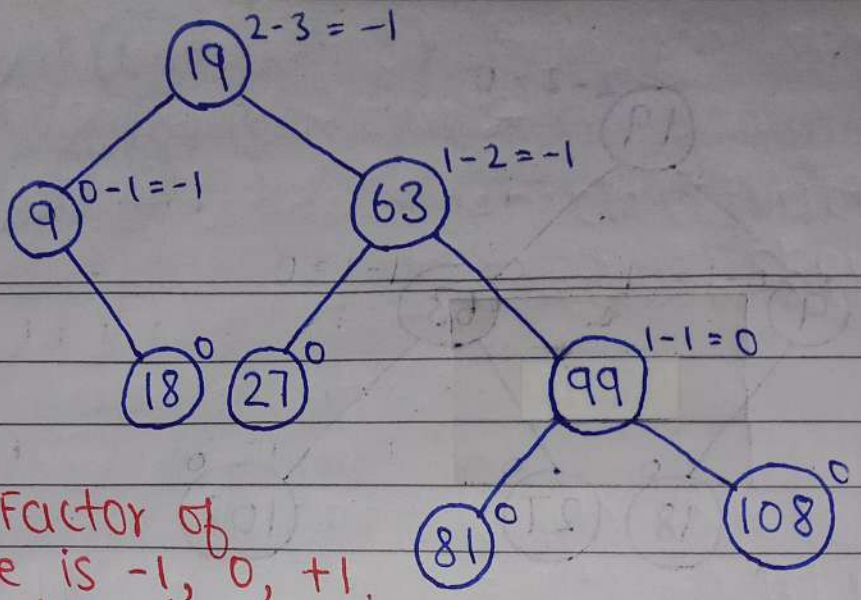


Insert 99,



Insert 81,



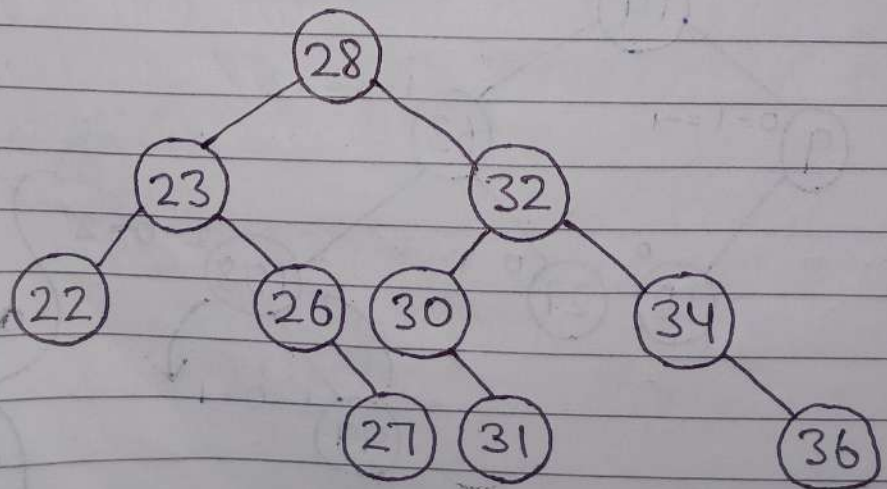


* Balancing Factor of each node is -1, 0, +1. Hence, the tree is Balanced AVL Tree.

[Deleting Node from AVL Tree]

* If element to be deleted does not have empty right subtree, then element is replaced with its inorder successor & its inorder successor is deleted instead.
* During winding up phase, we need to revisit every node on the path from the point of deletion upto the root, rebalance the tree if require.

Eg:- Given AVL Tree :-



Inorder Traversal,
22, 23, 26, 27, 28, 30, 31, 32, 34, 36

* Searching in Binary Search Tree and AVL Tree :-

We start at the root node and move down the tree and while descending when we encounter a node, we compare the desired key with the key of that node and take appropriate action.

i) If the desired key is equal to the key in the node then the search is successful.

ii) If the desired key is less than the key of the node then we move to left child.

iii) If the desired key is greater than the key of the node then we move to right child.

In the process, if we reach a NULL left child or NULL right child then the search is unsuccessful, i.e., desired element is not present in the tree.

The non-recursive function for searching a node is given below :-

```
struct node *search(struct node
                    *ptr, int skey)
{
  while(ptr != NULL)
  {
    if(skey < ptr->info){
      ptr = ptr->lchild;
    }
    else if(skey > ptr->info){
      ptr = ptr->rchild;
    }
    else{
      return ptr;
    }
  }
  return NULL;
}
```

move to left child ←

move to right child ←

if skey found return its address. ←

↓ if skey not found return null.

The above function is passed the address of the root of the tree & the key to be search. It returns the address of the node having the desired key or NULL, if the node is not found.

The recursive function for searching a node is given below :-

```
struct node *search(struct node
                    *ptr, int skey)
{
  if(ptr == NULL){
      printf("not found");
      return NULL;
  }
  else if(skey < ptr -> info){
      return search(ptr -> lchild,
                                skey);
  }
  else if(skey > ptr -> info){
      return search(ptr -> rchild,
                                skey);
  }
  else{
      return ptr;
  }
}
```

* Finding minimum node :- The last node in the leftmost path starting from the root is the minimum element. To find this node we start with the root & move along the leftmost path until we get a node with no left child.

```
struct node *min(struct node *ptr)
{
    if(ptr != NULL)
    {
        while(ptr->lchild != NULL)
        {
            ptr = ptr->lchild;
        }
    }
    return ptr;
}
```

* Finding Maximum Node :- The last node in the rightmost path starting from root is the maximum element. To find this node we start at the root & move along the rightmost path until we get a node with no right child.

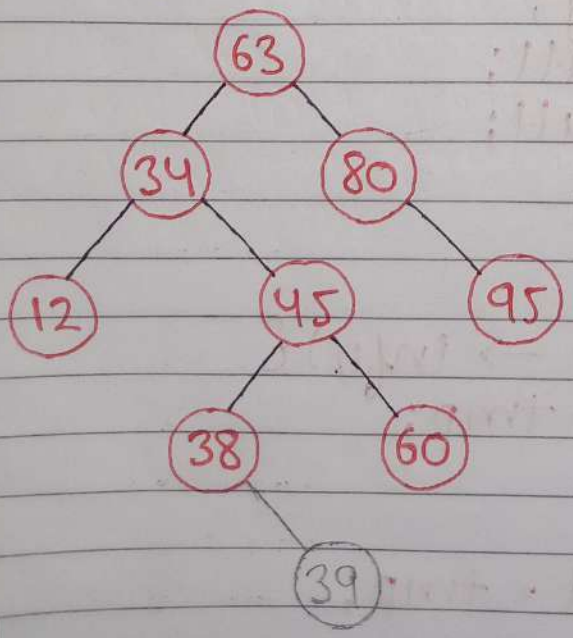
```
struct node *max(struct node *ptr)
{
    if(ptr != NULL)
    {
        while(ptr->rchild != NULL)
        {
            ptr = ptr->rchild;
        }
    }
    return ptr;
}
```

* Insertion in Binary Search Tree :-

For inserting a new node, first the key is searched & if it is not found in the tree then it is inserted at the place where search terminates.
we start at the root node and move down the tree & while descending when we encoun-ter a node we compare the key to be insert-ed with the key of that node & take appropriate action.

i) if the key to be inserted is equal to the key in the node then there is nothing to be done as duplicate keys are not allowed.
ii) If the key to be inserted is less than the key of the node then we move to left child.
iii) If the key to be inserted is greater than the key of the node then we move to right child.

* we insert the new key when we reach a NULL left or right child.



Insert 39,

39 < 63, move to left child
39 > 34, move to right child
39 < 45, move to left child.
39 > 38, move to right child
NULL Right child.
Insert 39 as Right child of 38.

The non-recursive function for inserting a node is given below :-

```
struct node *insert(struct node *root,
                    int ikey)
{
  struct node *tmp, *par, *ptr;
  ptr = root;
  par = null;
  while (ptr != null){
     par = ptr;
     if (ikey < ptr -> info){
        ptr = ptr -> lchild;
     }
     else if( ikey > ptr -> info){
        ptr = ptr -> rchild;
     }
     else{
        printf("Duplicate Key");
        return root;
     }
  }
  tmp = (struct node *)malloc(sizeof
                    (struct node));
  tmp -> info = ikey;
  tmp -> lchild = null;
  tmp -> rchild = null;
  if (par == null){
     root = tmp;
  }
  else if (ikey < par -> info){
     par -> lchild = tmp;
  }
  else{
     par -> rchild = tmp;
  }
  return root; }
```

The recursive function for inserting a node is given below :-

```
struct node *insert(struct node *ptr, int ikey)
{
    if(ptr == null){
    ptr = (struct node *) malloc
          (sizeof(struct node));
    ptr -> info = ikey;
    ptr -> lchild = null;
    ptr -> rchild = null;
    }
    else if(ikey < ptr -> info){
        ptr -> lchild = insert(ptr ->
                        lchild, ikey);
    }
    else if(ikey > ptr -> info){
        ptr -> rchild = insert(ptr ->
                        rchild, ikey);
    }
    else{
        printf("Duplicate Key");
    }
    return ptr;
}
```

To insert in empty tree, root is made to point to the new node. As we move down the tree, we keep track of the parent of the node because this is required for the insertion of new node.
The pointer "ptr" walks down the path and the parent of the "ptr" is maintained through pointer "par". when search terminates unsucc -essfully, "ptr" is NULL and "par" points to the node whose link should be changed to inse -rt the new node. The new node "tmp" is made the left or right child of the parent "par".

* Insertion and Deletion Operations in AVL Trees :-

```
#include<stdio.h>
#include<stdlib.h>
# define FALSE 0
# define TRUE 1
struct node {
    struct node *lchild;
    int info;
    struct node *rchild;
    int balance;
};
struct node *RotateLeft (struct node *pptr);
struct node *RotateRight (struct node *pptr);
struct node *insert (struct node *pptr,
                     int ikey);
struct node *insert_left_check (struct node
                *pptr, int *ptaller);
struct node *insert_right_check (struct
        node *pptr, int *ptaller);
struct node *insert_LeftBalance (struct
                node *pptr);
struct node *insert_RightBalance (struct
                node *pptr);
struct node *del (struct node *pptr, int
                dkey);
struct node *del_left_check (struct node
                *pptr, int *pshorter);
struct node *del_right_check (struct node
                *pptr, int *pshorter);
struct node *del_LeftBalance (struct node
                *pptr, int *pshorter);
struct node *del_RightBalance (struct node
                *pptr, int *pshorter);
```

```
main(){
    int choice, key;
    struct node *root = NULL;
    while (1) {
        printf("\n");
        printf("1. insertion\n");
        printf("2. Deletion\n");
        printf("3. Inorder Traversal");
        printf("4. Quit");
        printf("Enter choice = ");
        scanf("%d", &choice);
        switch(choice){
            case 1:
                printf("enter key to be
                    inserted = ");
                scanf("%d", &key);
                root = insert(root, key);
                break;
            case 2:
                printf("enter key to be
                    deleted = ");
                scanf("%d", &key);
                root = del(root, key);
                break;
            case 3:
                inorder(root);
                break;
            case 4:
                exit(1);
            default:
                printf("wrong choice");
        } (* end of switch *)
    } (* end of while *)
} (* end of main *)
```

```
struct node *RotateLeft(struct node *pptr)
{
    struct node *aptr;
    aptr = pptr -> rchild;
    pptr -> rchild = aptr -> lchild;
    aptr -> lchild = pptr;
    return aptr;
}

struct node *RotateRight(struct node *pptr)
{
    struct node *aptr;
    aptr = pptr -> lchild;
    pptr -> lchild = aptr -> rchild;
    aptr -> rchild = pptr;
    return aptr;
}

struct node *insert(struct node *pptr,
                    int ikey)
{
    static int taller;
    if (pptr == NULL) {
        pptr = (struct node *) malloc
               (sizeof (struct node));
        pptr -> info = ikey;
        pptr -> lchild = NULL;
        pptr -> rchild = NULL;
        pptr -> balance = 0;
        taller = TRUE;
    }
    else if (ikey < pptr -> info) {
        pptr -> lchild = insert(pptr -> lchild,
                                ikey);
```

```
        if (taller == TRUE) {
            pptr = insert_left_check (pptr,
                                      &taller);
        }
    }
    else if (ikey > pptr -> info) {
        pptr -> rchild = insert (pptr ->
                                 rchild, ikey);
        if (taller == TRUE) {
            pptr = insert_right_check
                (pptr, &taller);
        }
    }
    else {
        printf("Dublicate Key ");
        taller = FALSE;
    }
    return pptr;
}

struct node *insert_left_check (struct node
                                *pptr, int *ptaller)
{
    switch (pptr -> balance) {
        case 0:
            pptr -> balance = 1;
            break;
        case -1:
            pptr -> balance = 0;
            *ptaller = FALSE;
            break;
        case 1:
            pptr = insert_LeftBalance (pptr);
            *ptaller = FALSE;
    } return pptr;
}
```

```
struct node *insert_right_check (struct
                node *pptr, int *ptaller)
{
    switch (pptr -> balance) {
        case 0:
            pptr -> balance = -1;
            break;
        case 1:
            pptr -> balance = 0;
            *ptaller = FALSE;
            break;
        case -1:
            pptr = insert_RightBalance(pptr);
            *ptaller = FALSE;
    }
    return pptr;
}

struct node *insert_LeftBalance (struct node
                                 *pptr)
{
    struct node *aptr, *bptr;
    aptr = pptr -> lchild;
    if (aptr -> balance == 1) {
        pptr -> balance = 0;
        aptr -> balance = 0;
        pptr = RotateRight (pptr);
    }
    else {
        bptr = aptr -> rchild;
        switch (bptr -> balance) {
            case -1:
                pptr -> balance = 0;
                aptr -> balance = 1;
                break;
```

```
            case 1:
                pptr -> balance = -1;
                aptr -> balance = 0;
                break;
            case 0:
                pptr -> balance = 0;
                aptr -> balance = 0;
        }
        bptr -> balance = 0;
        pptr -> lchild = RotateLeft (aptr);
        pptr = RotateRight (pptr);
    }
    return pptr;
}

struct node * insert_RightBalance (struct
                                   node *pptr)
{
    struct node *aptr, *bptr;
    aptr = pptr -> rchild;
    if (aptr -> balance == -1) {
        pptr -> balance = 0;
        aptr -> balance = 0;
        pptr = RotateLeft (pptr);
    }
    else {
        bptr = aptr -> lchild;
        switch (bptr -> balance) {
            case -1:
                pptr -> balance = 1;
                aptr -> balance = 0;
                break;
            case 1:
                pptr -> balance = 0;
                aptr -> balance = -1;
```

```
            break;
        case 0:
            pptr -> balance = 0;
            aptr -> balance = 0;
        }
        bptr -> balance = 0;
        pptr -> rchild = RotateRight (aptr);
        pptr = RotateLeft (pptr);
    }
    return pptr;
}

struct node *del (struct node *pptr, int
                                    dkey)
{
    struct node *tmp, *succ;
    static int shorter;
    if (pptr == NULL){
        printf("not present");
        shorter = FALSE;
        return (pptr);
    }
    if ( dkey < pptr -> info) {
        pptr -> lchild = del (pptr -> lchild, dkey);
        if (shorter == TRUE) {
            pptr = del_left_check (pptr,
                                    &shorter);
        }
    }
    else if (dkey -> pptr -> info) {
        pptr -> rchild = del (pptr -> rchild,
                                    dkey);
        if (shorter == TRUE) {
            pptr = del_right_check (pptr,
                                    &shorter);
}   }
```

```
else {
    if (pptr -> lchild != NULL && pptr ->
                rchild != NULL)
    {
        succ = pptr -> rchild;
        while (succ -> lchild) {
            succ = succ -> lchild;
        }
        pptr -> info = succ -> info;
        pptr -> rchild = del (pptr ->
                    rchild, succ -> info);
        if (shorter == TRUE)
        {
            pptr = del_right_check
                    (pptr, &shorter);
        }
    }
    else {
        tmp = pptr;
        if (pptr -> lchild != NULL) {
            pptr = pptr -> lchild;
        }
        else if (pptr -> rchild != NULL) {
            pptr = pptr -> rchild;
        }
        else {
            pptr = NULL;
        }
        free (tmp);
        shorter = TRUE;
    }
}
return pptr;
}
```

```
struct node *del_left_check(struct
                node *pptr, int *pshor
                                -ter)
{
  switch(pptr->balance)
  {
     case 0:
        pptr->balance = -1;
        *pshorter = FALSE;
        break;
     case 1:
        pptr->balance = 0;
        break;
     case -1:
        pptr = del_RightBalance(pptr,
                          pshorter);
  }
  return pptr;
}

struct node *del_right_check(struct
        node *pptr, int *pshorter)
{
   switch(pptr->balance)
   {
      case 0:
         pptr->balance = 1;
         *pshorter = FALSE;
         break;
      case -1:
         pptr->balance = 0;
         break;
      case 1:
         pptr = del_LeftBalance(pptr,
                          pshorter);
   }
```

```
    return pptr;
}
struct node *del_LeftBalance(struct
                node *pptr, int
                *pshorter)
{
    struct node *aptr, *bptr;
    aptr = pptr->lchild;
    if(aptr->balance == 0){
        pptr->balance = 1;
        aptr->balance = -1;
        *pshorter = FALSE;
        pptr = RotateRight(pptr);
    }
    else if(aptr->balance == 1){
        pptr->balance = 0;
        aptr->balance = 0;
        pptr = RotateRight(pptr);
    }
    else{
        bptr = aptr->rchild;
        switch(bptr->balance){
            case 0:
                pptr->balance = 0;
                aptr->balance = 0;
                break;
            case 1:
                pptr->balance = -1;
                aptr->balance = 0;
                break;
            case -1:
                pptr->balance = 0;
                aptr->balance = 1;
        }
```

```
        bptr -> balance = 0;
        pptr -> lchild = RotateLeft (aptr);
        pptr = RotateRight (pptr);
    }
    return pptr;
}

struct node *del_RightBalance (struct node
                    *pptr, int *pshorter)
{
    struct node *aptr, *bptr;
    aptr = pptr -> rchild;
    if (aptr -> balance == 0) {
        pptr -> balance = -1;
        aptr -> balance = 1;
        *pshorter = FALSE;
        pptr = RotateLeft (pptr);
    }
    else if (aptr -> balance == -1) {
        pptr -> balance = 0;
        aptr -> balance = 0;
        pptr = RotateLeft (pptr);
    }
    else {
        bptr = aptr -> lchild;
        switch (bptr -> balance) {
            case 0:
                pptr -> balance = 0;
                aptr -> balance = 0;
                break;
            case 1:
                pptr -> balance = 0;
                aptr -> balance = -1;
                break;
```

```
        case -1:
            pptr -> balance = 1;
            aptr -> balance = 0;
        }
        bptr -> balance = 0;
        pptr -> rchild = RotateRight(aptr);
        pptr = RotateLeft(pptr);
    }
    return pptr;
}
```

Multiway Search Tree (M-Way Tree)

Till now, in all the trees that we have seen, a node can hold only one key value and can have atmost two children.
In a m-way search tree, a node can hold more than one value and can have more than two childrens.

Till now, wa have studied about internal searching only, i.e., we have assumed that data to be searched is present in primary storage area. In this, we see external searc-hing in which data is to be retrieved from secondary storage like disk files.
We know that the access time in the case of secondary storage is much more than that of primary storage. So, while doing external searching we should try to reduce the number of accesses. when data is accessed from a disk, a whole block is read instead of a single word. For this purpose, m-way tree is used.

* B-Tree :- In external searching, our aim is to minimize the file access, and this can be done by reducing the height of the tree.

The height of m-way tree is less because of its large branching factor but its height can still be reduced if it is balanced.
So, a new tree structure was developed, i.e., height balancing m-way search tree, also called b-tree.
A B-Tree of order m can be defined as an m-way search tree which is either empty or satisfies the following properties :-

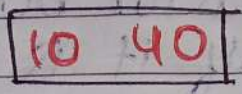
i) All leaf node are at the same level.
ii) All non-leaf nodes (except root node) should have atleast (m/2) children.
iii) All nodes (except root node) should have atleast (m-2)-1 keys.
iv) If the root node is a leaf node (only node in the tree), then it will have no children and will have atleast one key.
If the root node is a non-leaf node, then it will have atleast 2 children and atleast one key.
v) A non leaf node with n-1 key values should have n non NULL children.

Insertion in B-Tree :- (order 5)

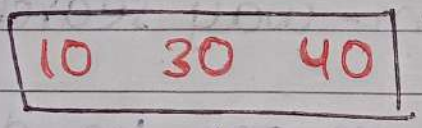
10, 40, 30, 35, 20, 15, 50, 28, 25, 5, 60, 19, 12, 38, 27, 90, 45, 48.

Insert 10,

Insert 40,

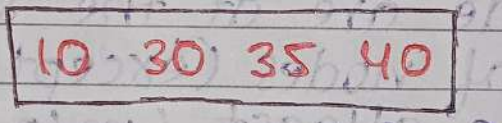


Insert 30,



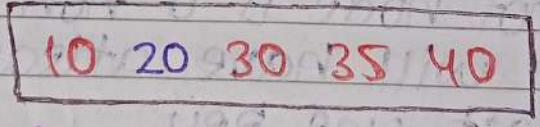
30 will be inserted between 10 & 40 since all the keys in a node should be in ascending order.

Insert 35,



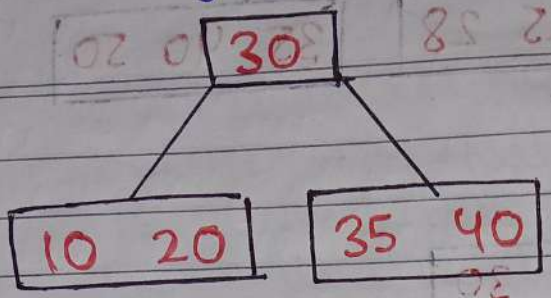
the max no. of keys for a node of B-tree of order 5 is 4
After insertion of 35, node become full.

Insert 20,

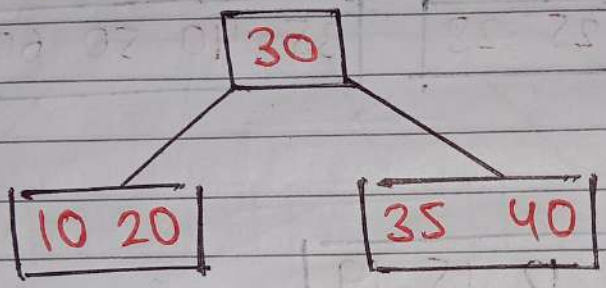


if we insert 20 here, the node become overfull. So, Splitting is done at the median key 30. After splitting, the median key goes to the parent node.

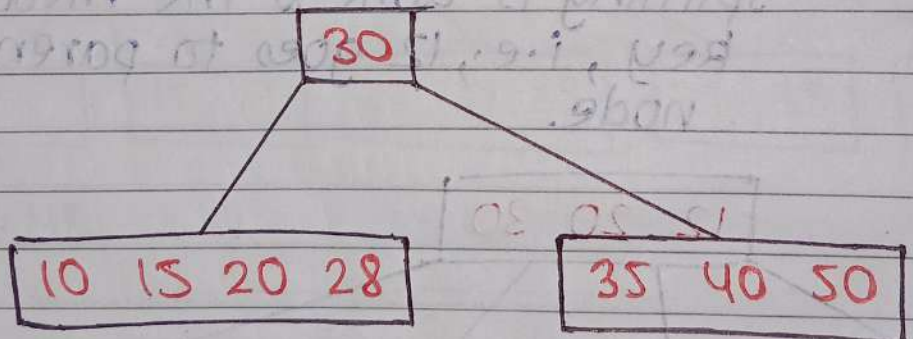
The key greater than the median key, goes in the right side & keys smaller than the median key goes in the left side.



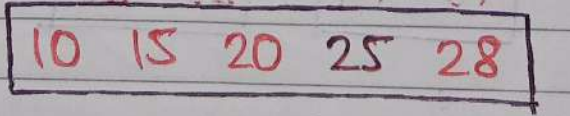
Insert 15, 50, 28,



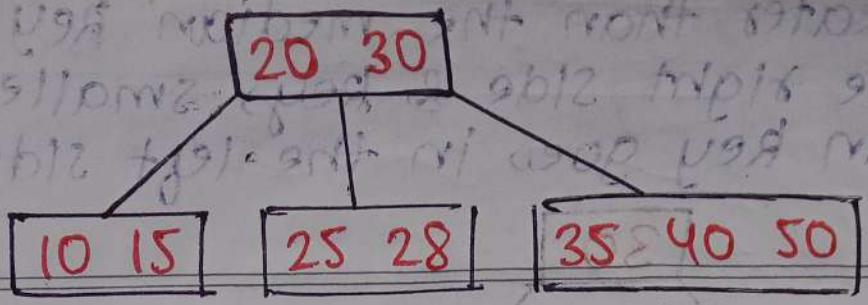
15 & 28 are less than 30, so they are inserted in the left node at appropriate place & 50 is greater than 30, so it is inserted in the right node.



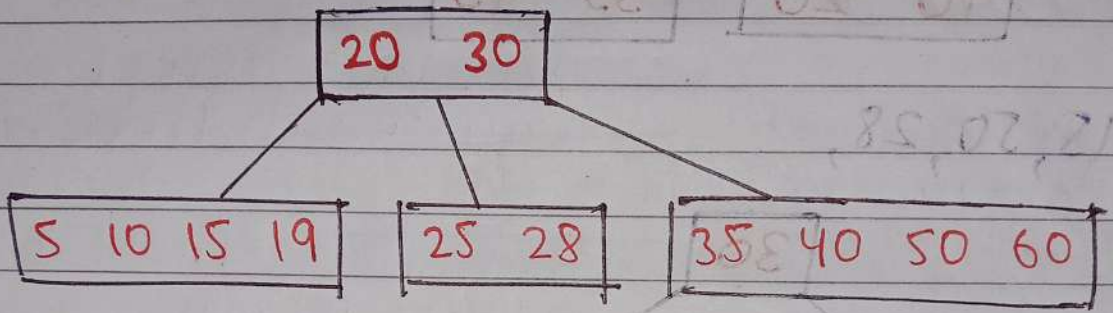
Insert 25;



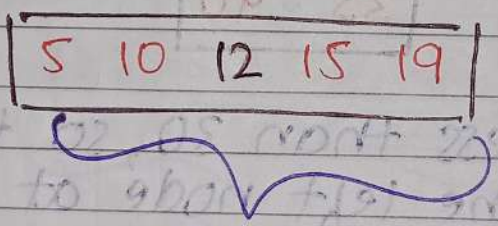
after inserting 25, the node became overfull. So, Splitting is done at the median key, i.e., 20, and 20 goes to the parent node.



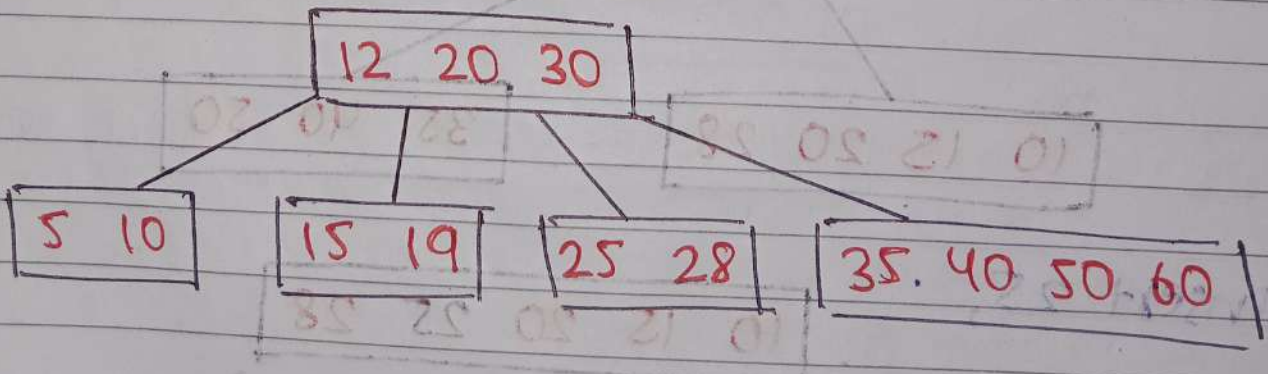
Insert 5, 60, 19,



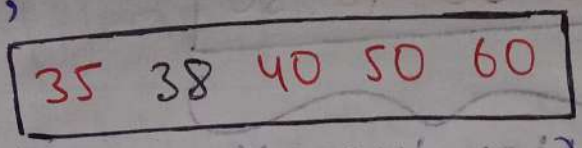
Insert 12,



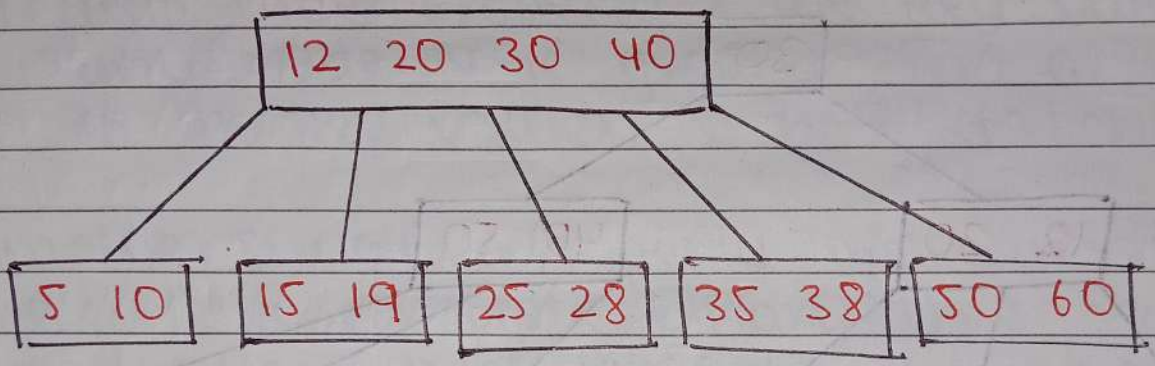
after inserting 12, the node became overfull. So, splitting is done & the median key, i.e., 12 goes to parent node.



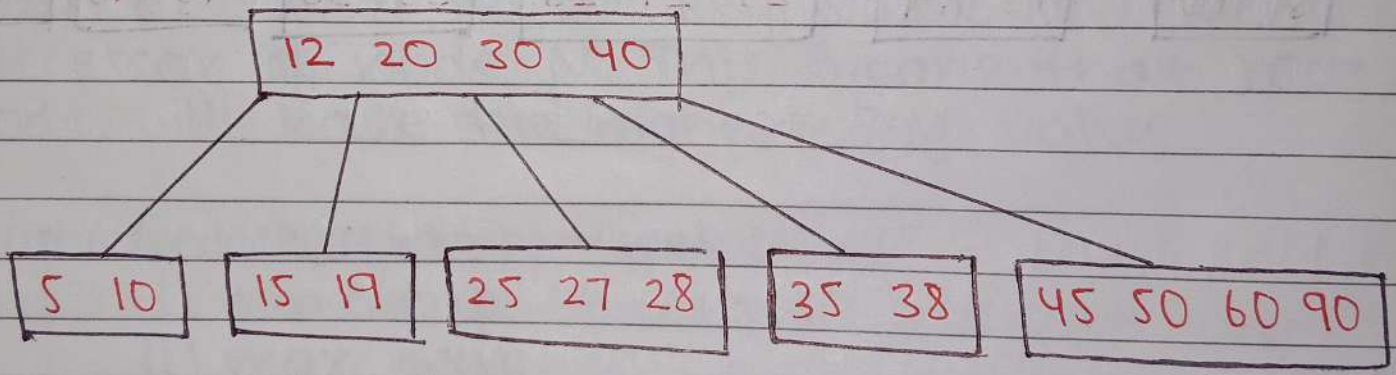
Insert 38,



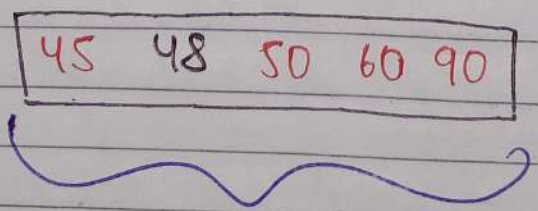
after inserting 38, node became overfull. so, splitting is done & the median key, 40 goes to the parent node.



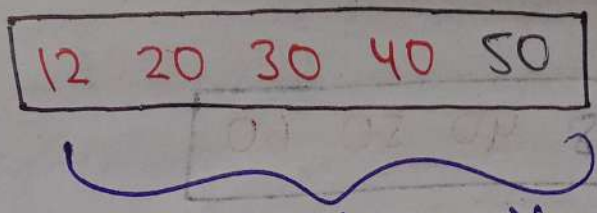
Insert 27, 90, 45,



Insert 48,



after inserting 48, node became overfull. so, splitting is done & the median key, i.e., 50 goes to the parent.



after inserting 50 in the parent node, parent node also become overfull. So, again splitting is done, and the median key 30 become the parent node.