Date - 23/10/19

UNIT - V

TREE

Definition

Properties

Basic Terminology -

(a) Root (b) Edge (c) Parent (d) Child (e) Sibling
(f) Degree (g) Internal Nodes (h) Leaf Node
(i) Level (j) Height (k) Depth (l) Subtree
(m) Forest.

BINARY TREE -

Definition
Unlabelled Binary Tree → formula depend upon height depth
Labelled Binary Tree
Types of Binary Tree -

(a) Rooted B.T.
(b) Full / Strictly B.T.
(c) Complete B.T.
(d) Almost Complete B.T.
(e) Skewed B.T.
Binary Search Tree.
Various formula or Properties of B.T.

A Graph in which no circuit is formed or self looping is K/a tree.
A Node is collection of data & address.

BINARY TREE -

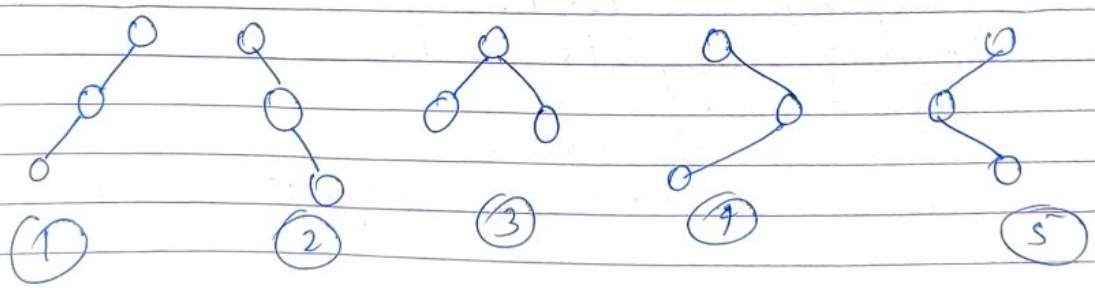
Binary Tree is the tree in which every node can have atmost two children. Each node can have 0, 1 or 2 children but not more than two

* Unlabelled Binary Tree -
A binary tree is unlabelled if it's node are not assign any label.

No. of different binary tree possible with n unlabelled node $= \frac{^{2n}C_n}{n+1}$.

ex- If we have three node then how many binary tree possible.

$$\frac{^{2\times3}C_3}{3+1} = \frac{6!}{3!(6-3)!} \Big/ 4 = 5.$$

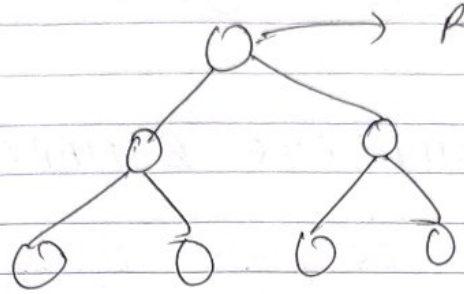


No. of binary tree possible with n labelled nodes $= \left(\frac{^{2n}C_n}{n+1}\right) * n!$

* TYPES OF BINARY TREE -

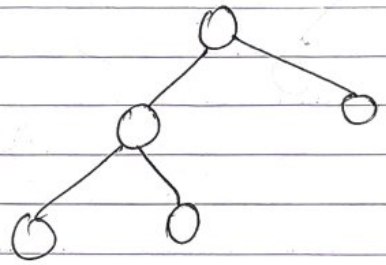
(i) ROOTED BINARY TREE -

A rooted binary tree is one that has a root node & every node has atmost two children



(ii) FULL OR STRICTLY BINARY TREE -

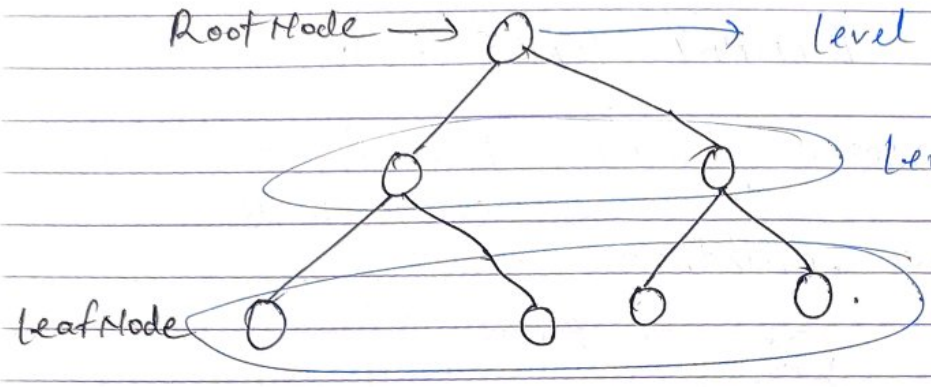
A full or strictly binary tree is a tree in which every ~~children has~~ nodes has 0 or 2 children



(iii) COMPLETE BINARY TREE (PERFECT B.T.)

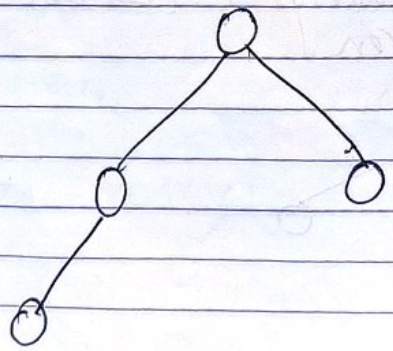
A Complete or perfect binary tree is a tree in which every internal node has exactly two children & all ~~and~~ leaf

node are at same level



(iv) ALMOST COMPLETE BINARY TREE-

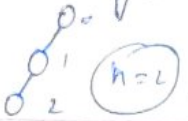
An almost complete binary tree is a tree in which all the levels are completely filled except possibly the last level & the last level must be strictly filled from left to right.



(v) SKEWED BINARY TREE -

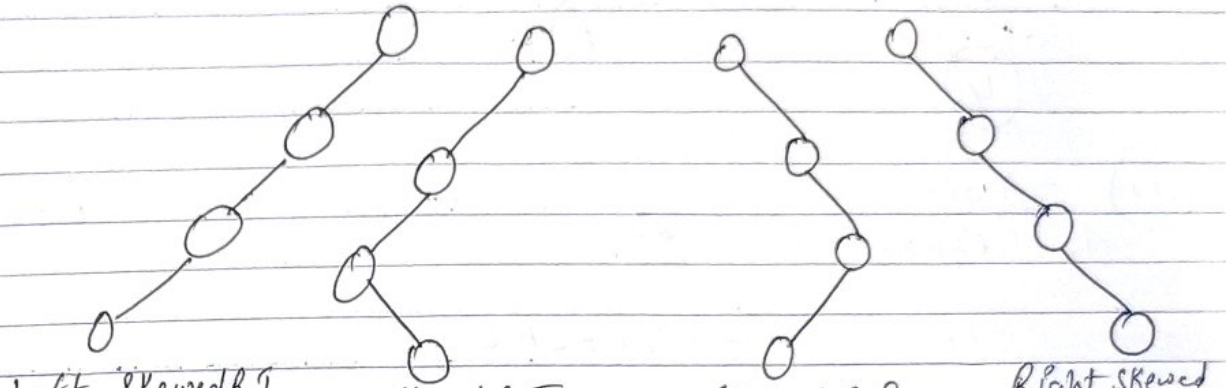
A skewed binary tree is the tree in which all the node except one node have one & only one child the

Height of the tree is a max. no. of edges b/w root node to last node.



remaining nodes has no child.

A SKewed binary tree is a tree of n node of n node such that it depth is n-1



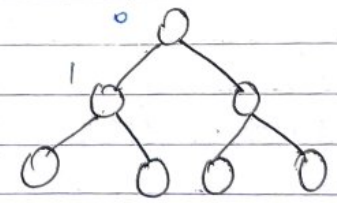
* PROPERTIES OF BINARY TREE -

(i) Minimum no. of node in a binary tree of height h is equal to h+1.

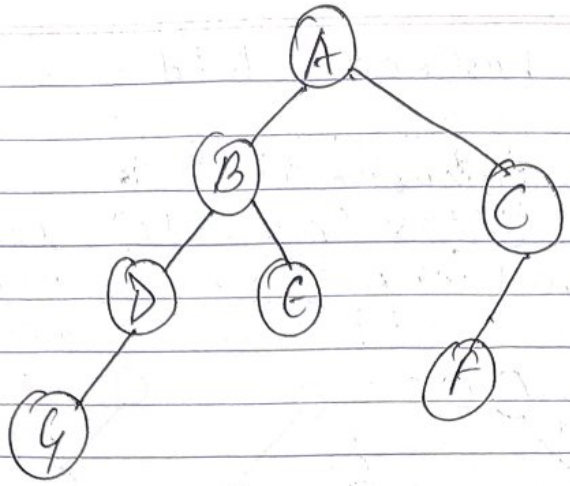
(ii) Maximum no. of node in a binary tree of height h is equal to $2^{h+1} - 1$

(iii) Total no. of leaf node in a binary tree is equal to total no. of degree two node + 1

Ex.



h = 3 =>
leaf node = 3+1 = 4.



(iv) Maximum no. of node at level L is equal to 2^L.

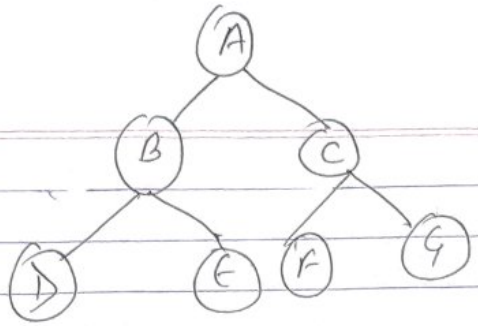
TREE TRAVERSAL -

It is also known as tree search or tree display. Tree traversal is a form of graph traversal & refirst to the process of visiting each node in a tree data structure exactly once. Tree traversal techniques are depth first search which includes three techniques (i) Preorder Tree Traversal
(ii) Inorder " "
(iii) Postorder " "

* ALGORITHM OF TREE TRAVERSAL

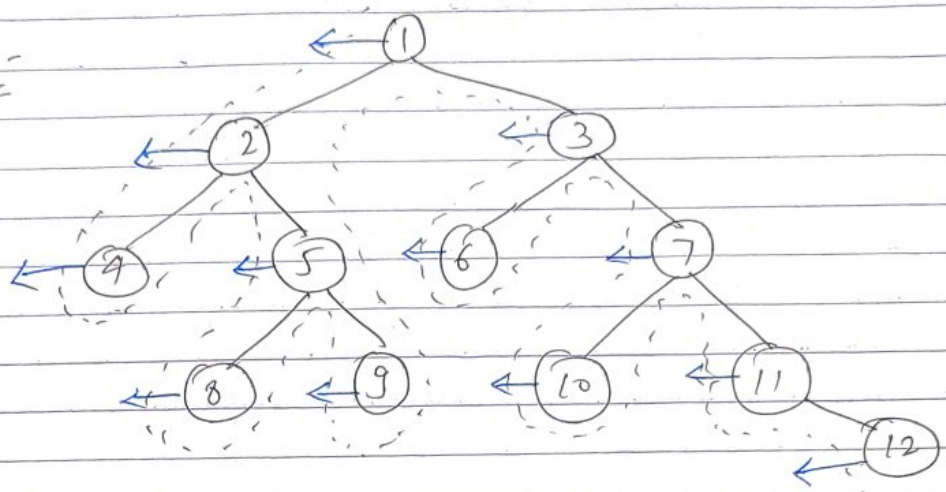
(i) Visit the root.
(ii) Traverse left subtree i.e., called Preorder left subtree.
(iii) Traverse right subtree i.e., called Preorder right subtree.

ex-



A B D E C F G.

ex-



1, 2, 4, 5, 8, 9, 3, 6, 7, 10, 11, 12

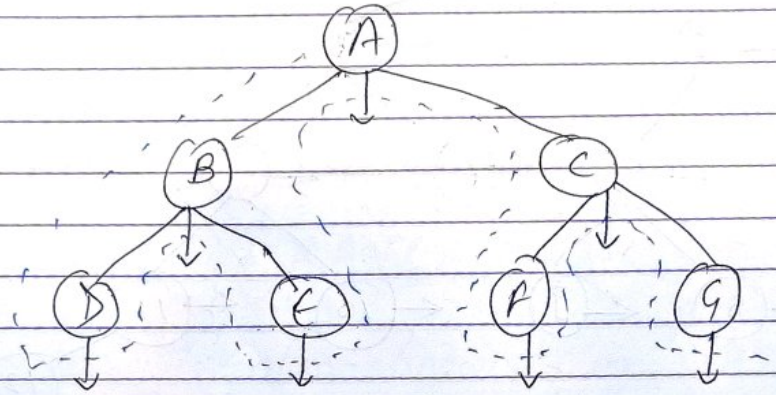
Write a algo to traverse a tree in preorder.

```
void Preorder (struct node *root)
{
    if (root != NULL)
    {
      printf ("%d \t", root -> data);
      preorder (root -> left);
      preorder (root -> right);
    }
}
```

* ALGORITH FOR INORDER.

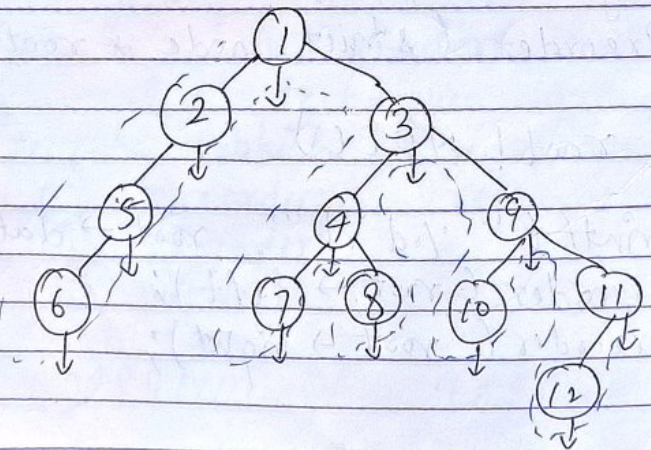
(1) TRAVERSE the left subtree i.e., call the inorder left subtree
(2) Visit the root.
(3) Traverse the right subtree i.e., call inorder right subtree.

ex -



D, B, E, A, F, C, G.

ex -



6, 5, 2, 1, 7, 4, 8, 3, 10, 9, 12, 11.

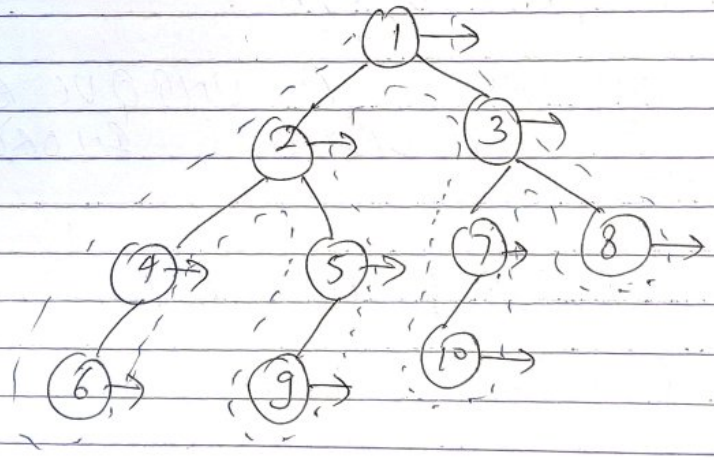
Q WAP in C to implement algo of inorder tree traversal.

```
void inorder ( struct node * root )
{
    if ( root != NULL)
    {
        inorder (root → left);
        printf ("%d\t", root → data);
        inorder ( root → right );
    }
}.
```

ALGORITHM OF POSTORDER

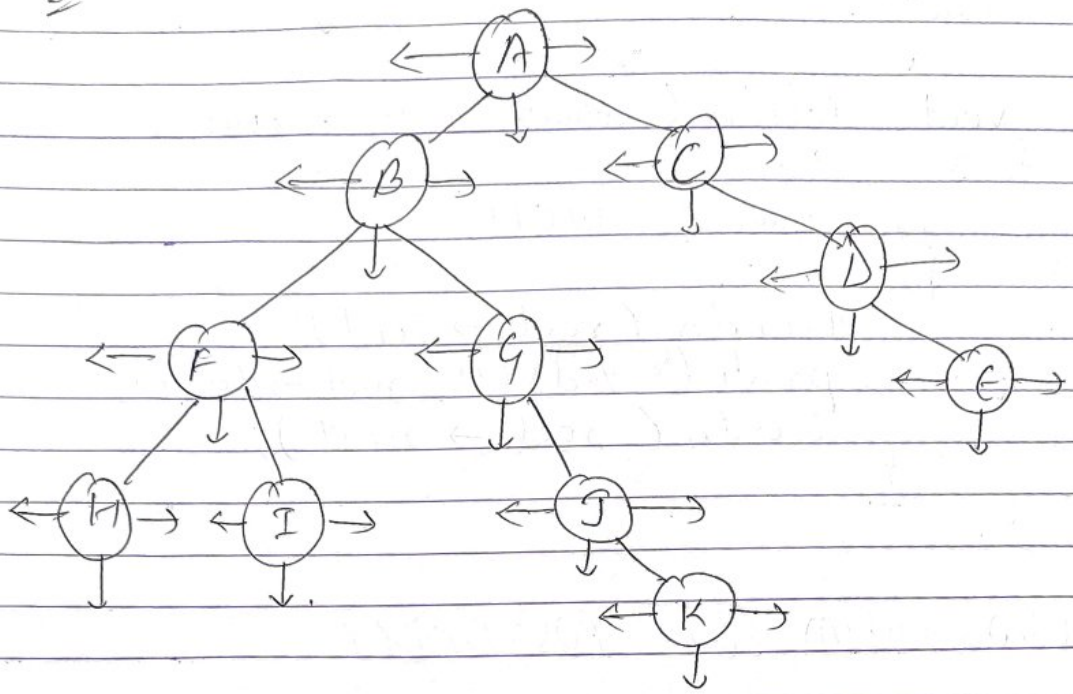
1. Traverse the ~~right~~ left subtree i.e., called postorder left subtree.
2. Traverse the right subtree i.e., call postorder right subtree.
3. Visit the root.

Ex -



6, 4, 9, 5, 2, 10, 7, 8, 3, 1.

Q. Construct Preorder, Inorder & Postorder.



Pre A, B F, H, I, G, J, K, C, D, E.

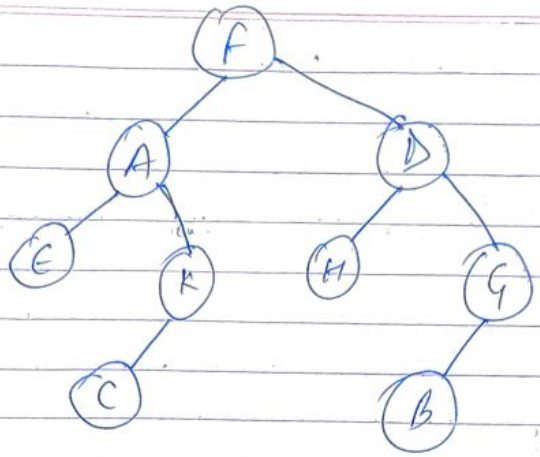
In H, F, I, B, G, J, K, A, C, D, E

Post H, I, F, K, J, G, B, E, D, C, A.

TO CONSTRUCT A UNIQUE BINARY TREE WHEN PREORDER & INORDER IS GIVEN-

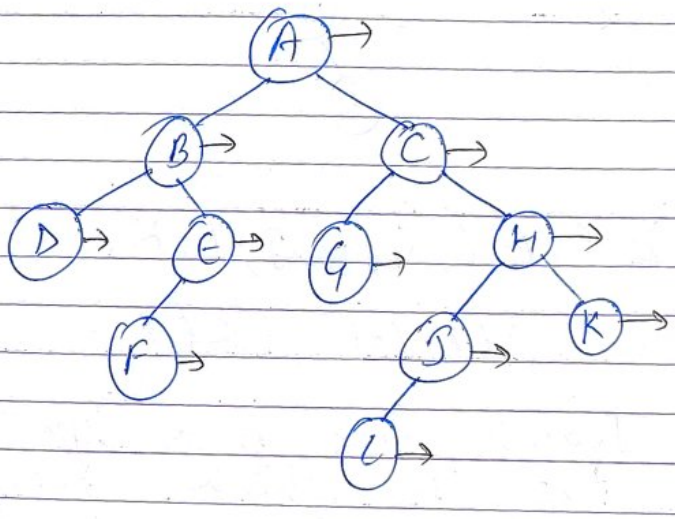
INORDER - E, A, C, K, F, H, D, B, G.

PREORDER - F, A, E, K, C, D, H, G, B.



INORDER - D, B, F, E, A, G, C, L, J, H, K.

PREORDER - A, C, H, K, J, L, G, B, E, F, D.



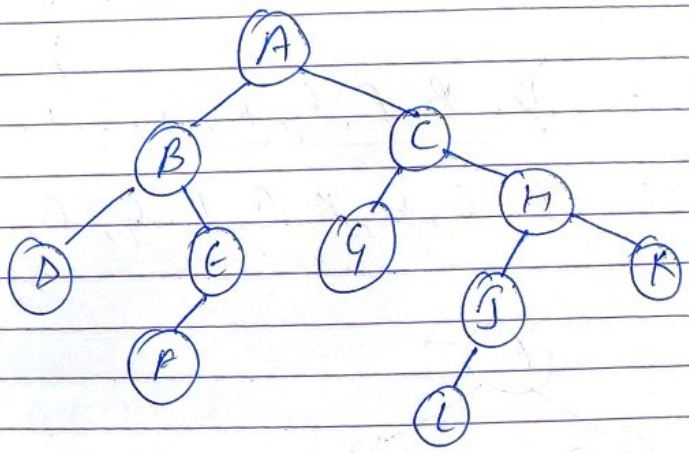
D, F, E, B, G, L, J, K, H, C, A.

HOW TO CONSTRUCT TREE WHEN POSTORDER & INORDER ARE GIVEN -

INORDER D, B, F, E, A, G, C, L, J, H, K

POSTORDER D, F, E, B, G, L, J, K, H, C, A.

FIND PREORDER.



Preorder - A, B, D, E, F, C, G, H, J, L, K.

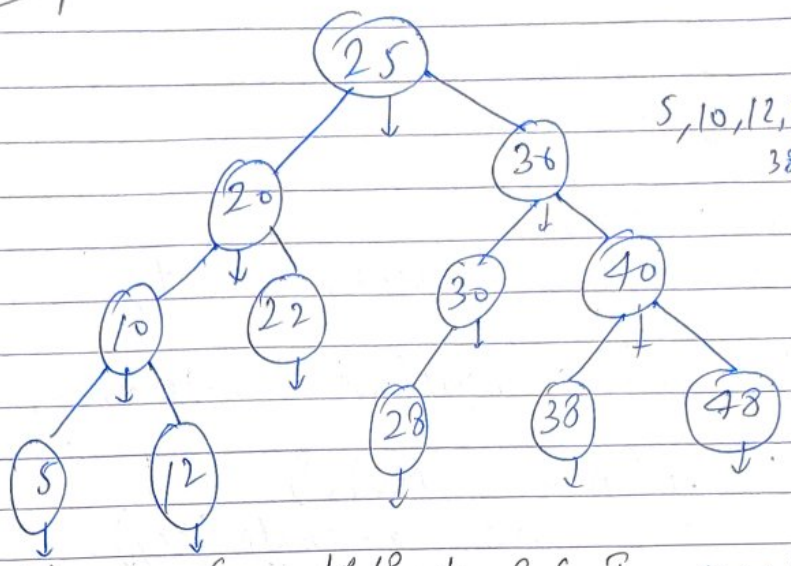
BINARY SEARCH TREE - (B.S.T.)

Binary Search Tree is a special kind of binary tree in which every node are arranged in specific orders.

In binary Search Tree, each node contains :

1. Only smaller value in its left sub tree.
2. Only greater value in its right sub tree.

for example: 25, 20, 10, 5, 12, 22, 36, 30, 28, 40, 38, 48.



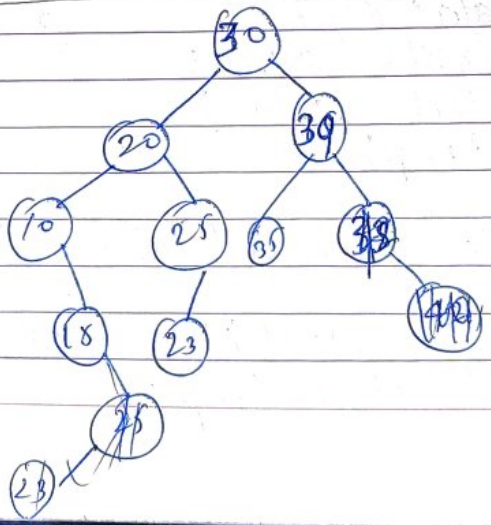
5, 10, 12, 20, 22, 25, 28, 30, 36, 38, 40, 48.

* No. of distinct B.S.T. possible with n distinct Key = $\frac{2nC_n}{n+1}$

BINARY SEARCH TREE TRAVERSAL -

Q Preorder of binary search tree is
30, 20, 10, 15, 25, 23, 39, 35, 42
find the Postorder of B.S.T.

10, 15, 20, 23, 25, 30, 35, 39, 42



OPERATION OF B.S.T. -
INSERTION, DELETION & UPDATION -

```
#include<stdio.h>
struct node;
{
    int data;
    struct node *left, *right;
};
struct node * insert(struct node *new, int Key)
{
    if(new == NULL)
        return newnode(Key);
    if( new->data < Key)
        new->right = insert(new->right, Key)
    if(new->data > Key)
        new->left = insert(new->left, Key)
    return node;
}
struct node * newnode (int Key)
{
    struct node *temp;
    temp = (struct node *) malloc(sizeof(struct node));
    temp->data = Key;
    temp->right = NULL;
    temp->left = NULL;
    return temp;
}
struct node *search ( struct node *temp, int value)
{
    if( temp == NULL || temp->Key == value)
        return temp;
```

Insert, particular element search, traversal (inorder - preorder, postorder

```
if ( temp → Key > value )
    search (temp → left, value);
else
    search (temp → right, value);
}
```

HEIGHT OF A TREE

Longest Path from root to leaf node i.e., in that path the no. of edges.

AVL.

Balance factor of node = height of left subtree - height of Right subtree.

TIME COMPLEXITY OF B.S.T. –

The time complexity of B.S.T. for insertion, searching & deletion operation is O(h). where h is the height of B.S.T.

* Worst Case Analysis of B.S.T –

The worst case of B.S.T. is SKewed B.S.T. It happens when values are inserted either in increasing or decreasing order occurence. In this (SKewed B.S.T) case the height of B.S.T. becomes equivalent to n where n is no. of values inserted in B.S.T.. Hence, for searching, insertion & deletion

order will be O(n).

* BEST CASE ANALYSIS OF B.S.T. -

The Best case of Analysis of B.S.T. will be a Balanced B.S.T. where the height of Balanced B.S.T. will be $h = \log_2 n$. where n is the no. of nodes or value inserted. So, by the above analysis we find if we balanced the tree by insertion & deletion operation we can reduce the time complexity for further searching, insertion & deletion operation. for balancing their are many algorithm that are -

(i) AVL Tree
(ii) Red-Black Tree.
(iii) B-Tree.

AVL TREE -

AVL Tree are self balancing B.S.T. where the difference of height of left subtree & right subtree of any node is not greater than 1.

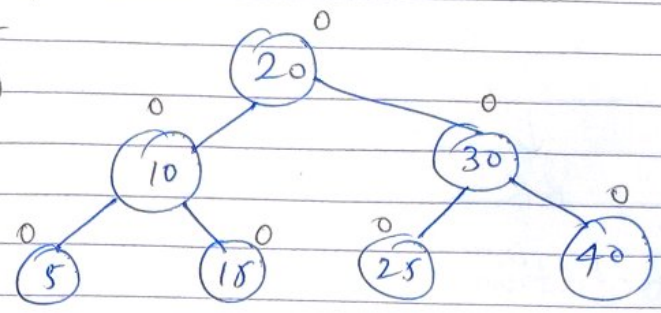
* Balanced Factor.

Balanced factor of are node in B.S.T. is equal to height of left subtree - height of right subtree.

So, AVL tree are those B.S.T. where Balance factor of each node will be either 0, -1 or 1.

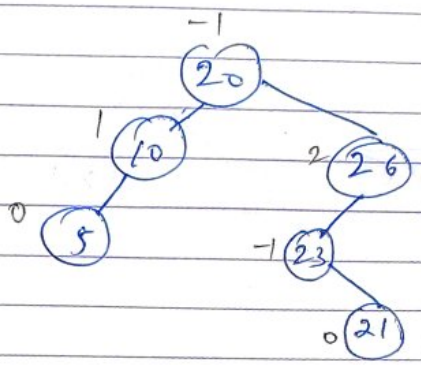
for example -

(i)



← AVL Tree

(ii)



Not a AVL Tree.

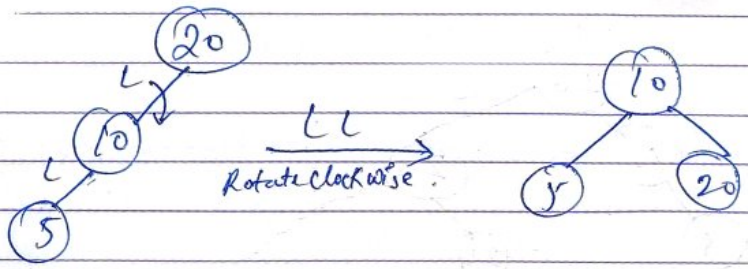
HOW BALANCING IS DONE IN AVL TREE?

In AVL Tree after performing insertion & deletion, we check Balanced factor of every node. If every node satisfy the balanced factor (i.e., 0, -1, 1) then the operation is concluded, if it is not so then we must make it balance. ~~For Balan~~ For Balancing we use rotation operation which can be classified as.

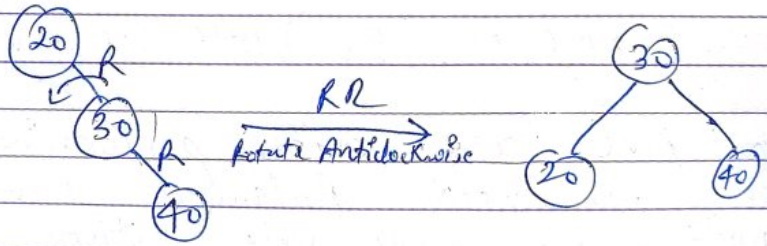
Rotation

- Single Rotation
  - LL
  - RR
- Double Rotation
  - LR
  - RL

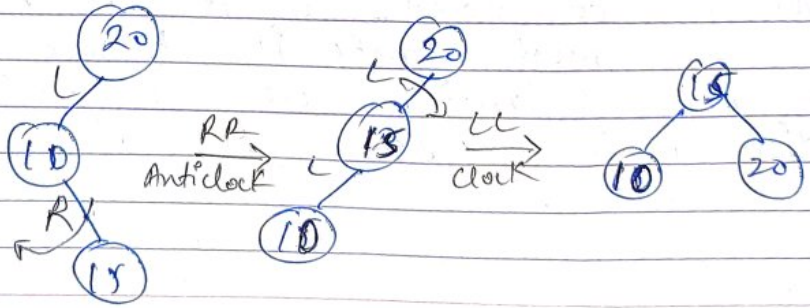
⇒ LL Rotation.



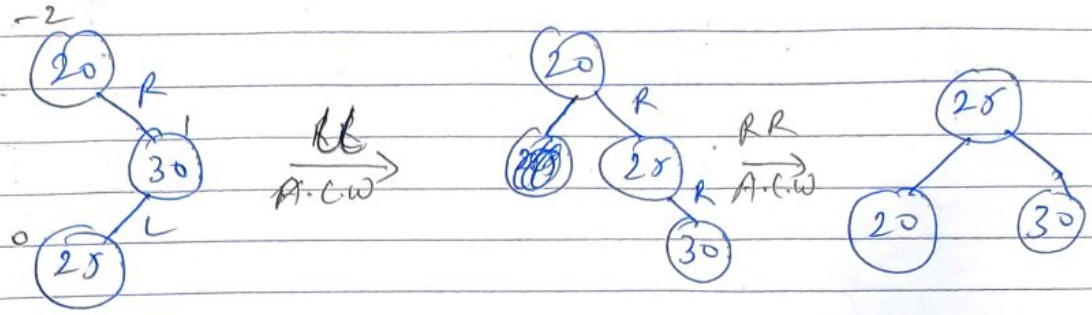
⇒ RR Rotation



⇒ LR Rotation

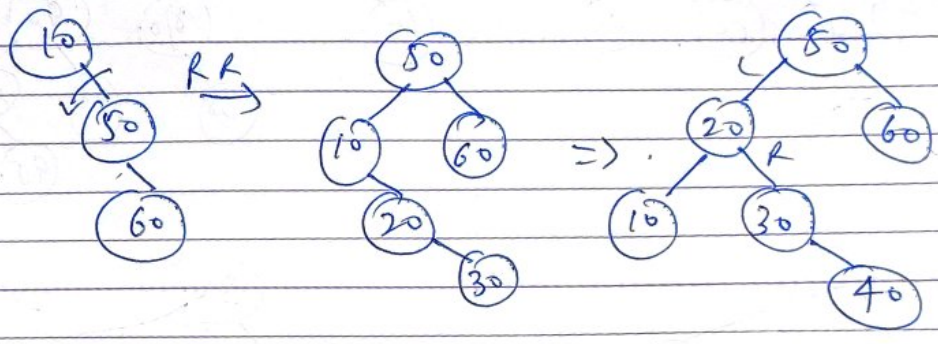


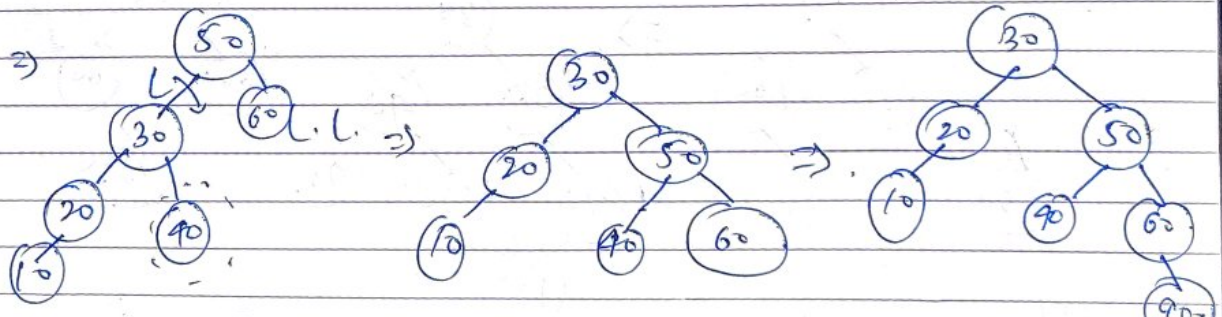
⇒ RL Rotation -

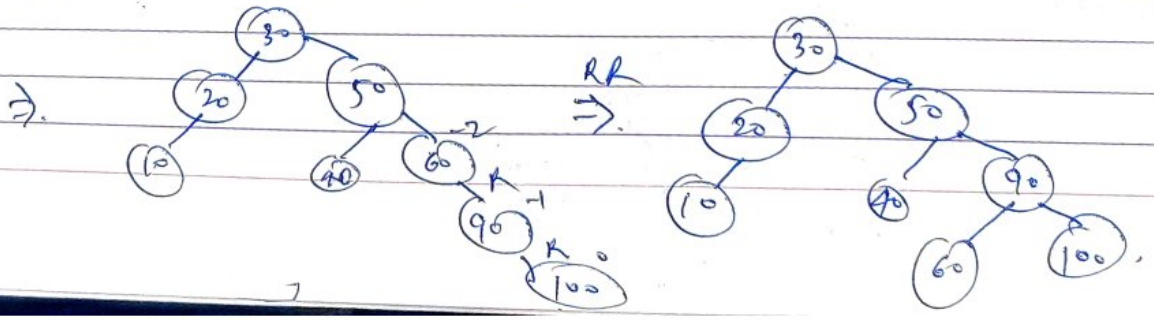


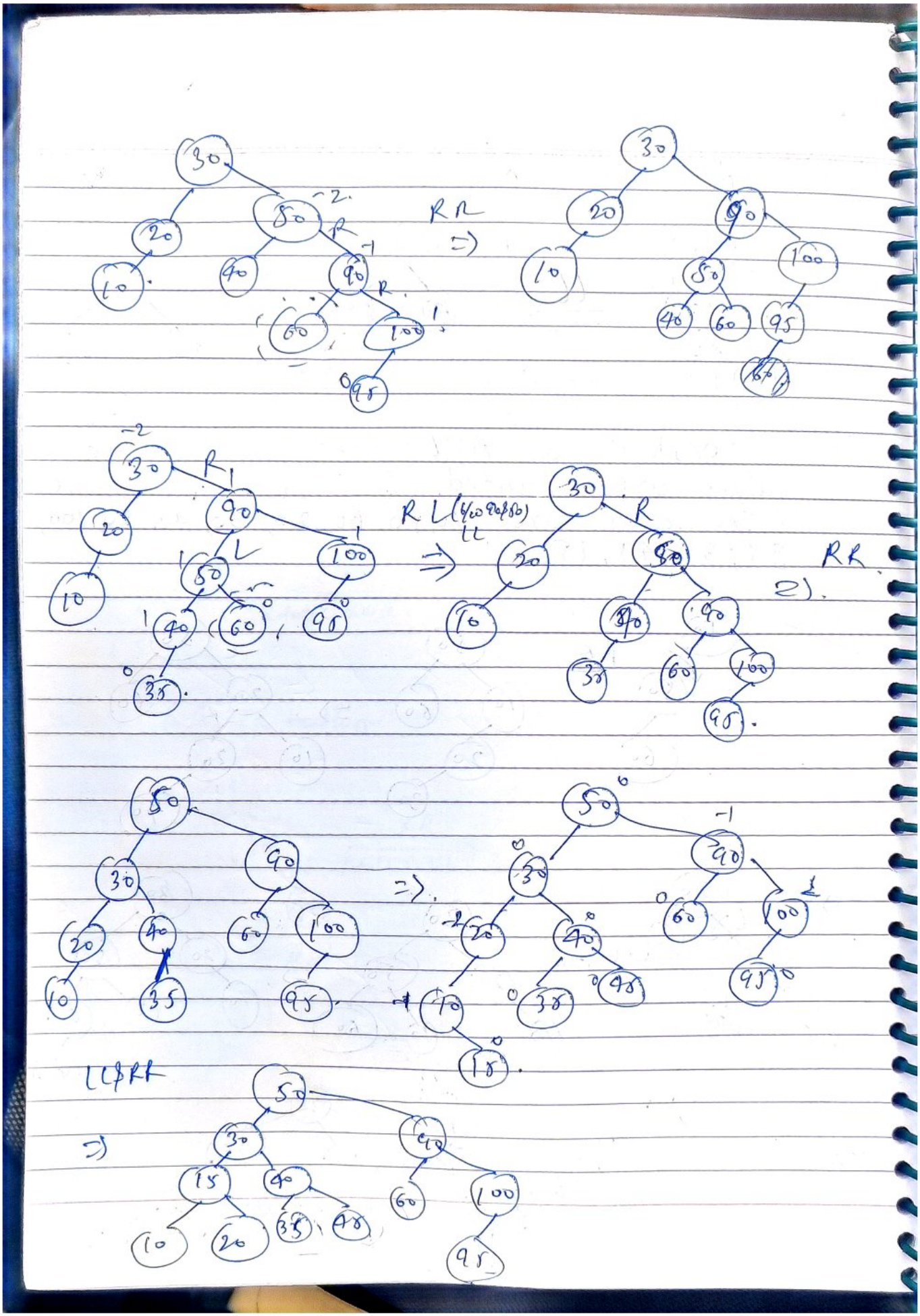
Ex- Construct a AVL Tree when the values are inserted in following sequence & The values are 10, 50, 60, 20, 30, 40, 90, 100, 95, 35, 45, 15.

Solution-



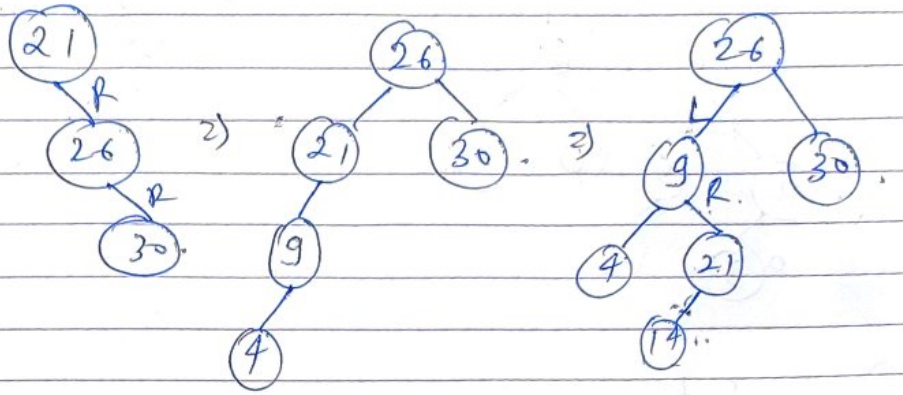


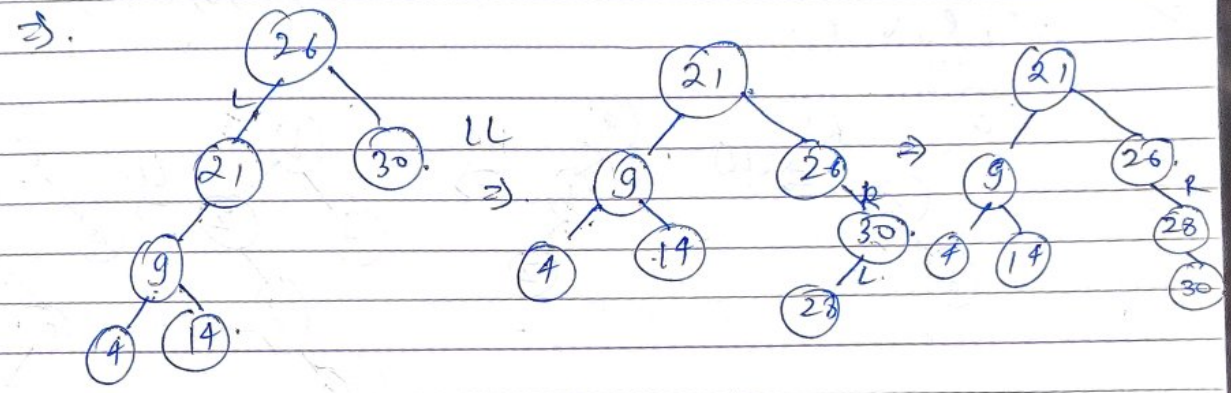


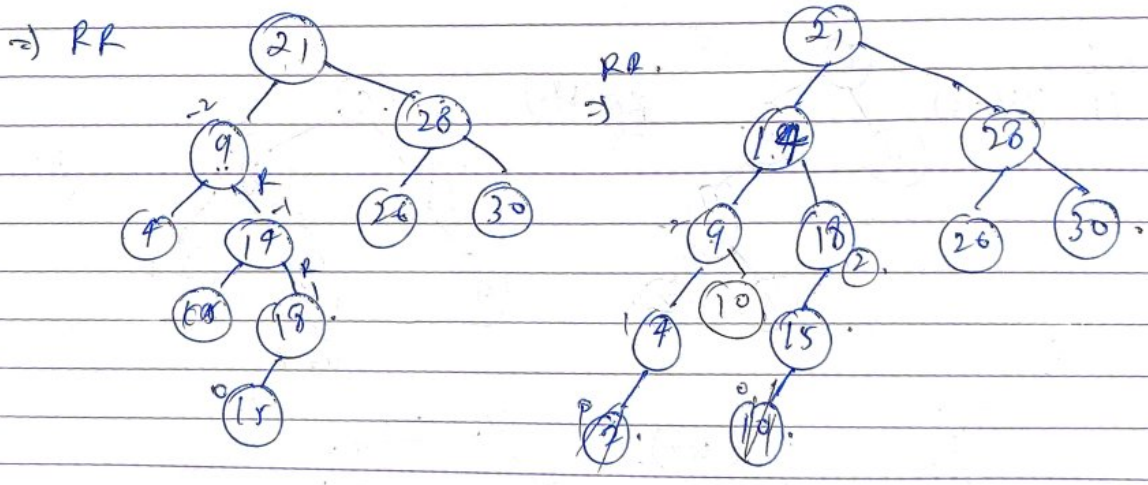


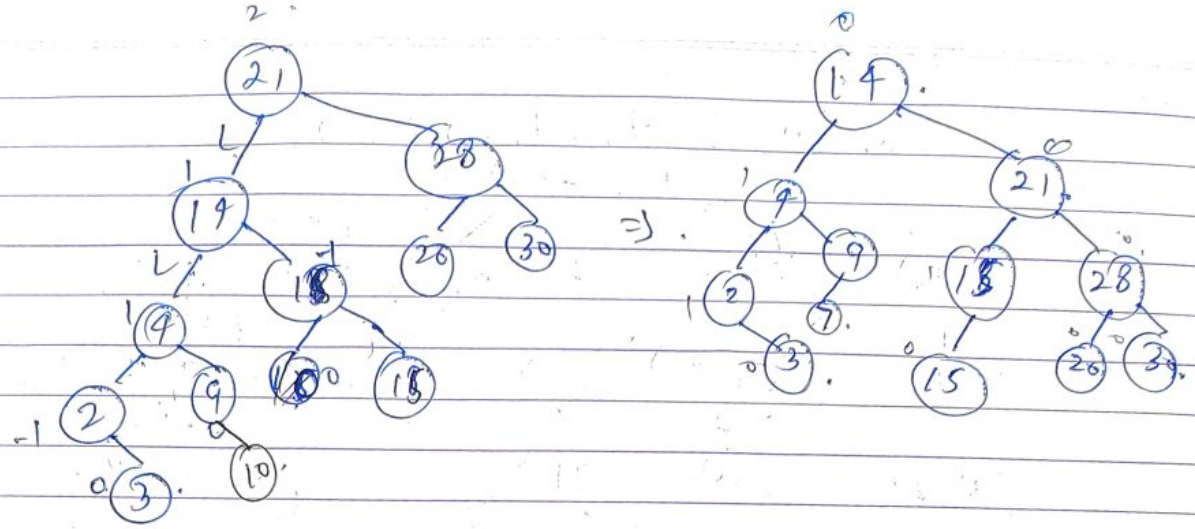
Q AVL :→

21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7

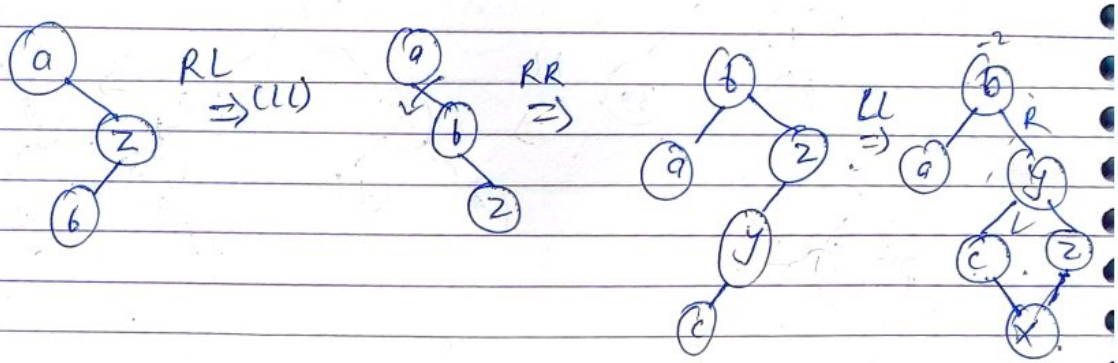


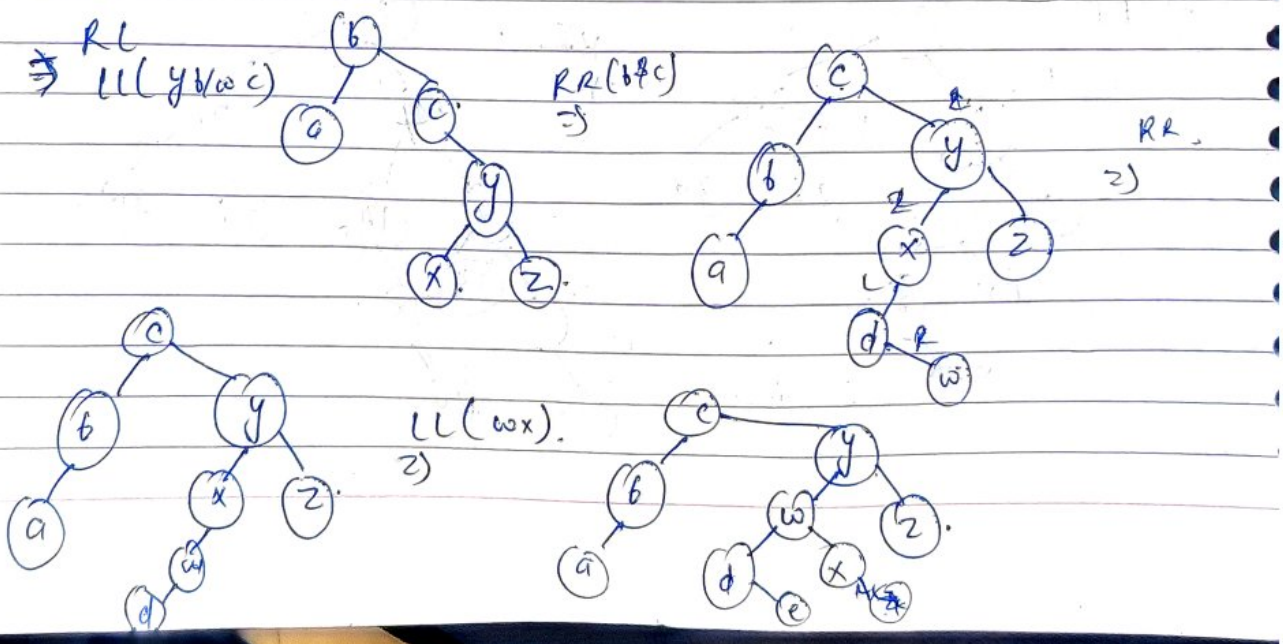




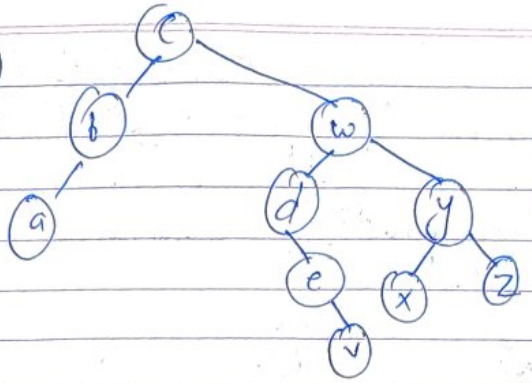


Q Find the AVL tree of a, z, b, y, c, x, d, w, e, v, f.

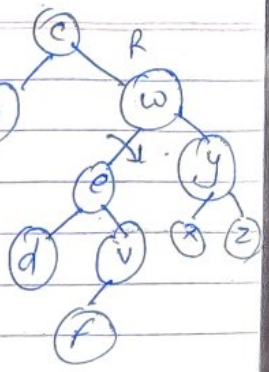




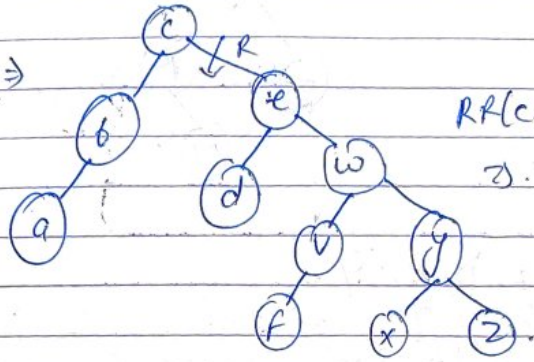
LL(y ω)
⇒



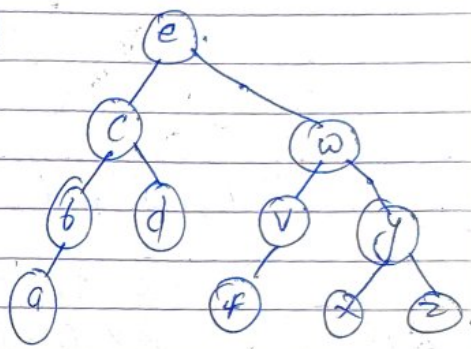
RR(ele).
⇒.



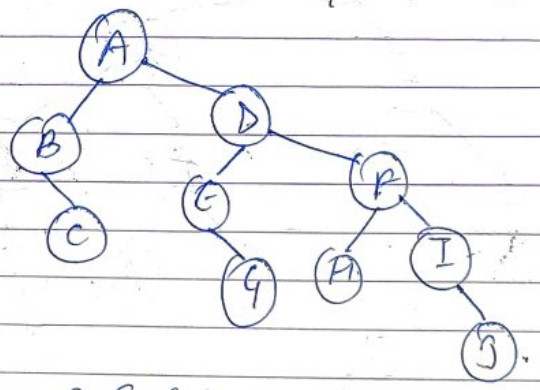
RL
LL(ωfe) ⇒



RR(cfe)
⇒.



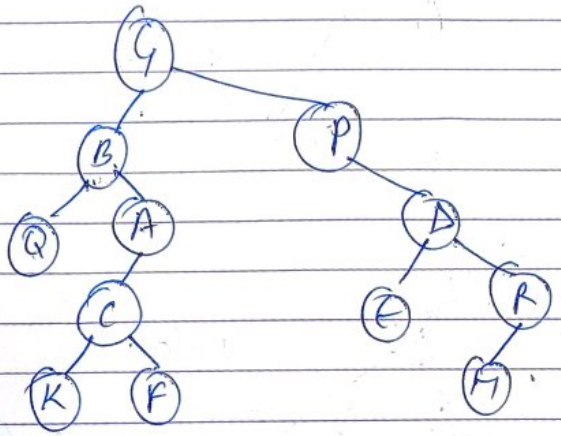
Q Inorder → B, C, A, E, G, D, H, F, I, J.
= Preorder → A, B, C, D, E, G, F, H, I, J.



Post - C, B, G, E, H, J, I, F, D, A.

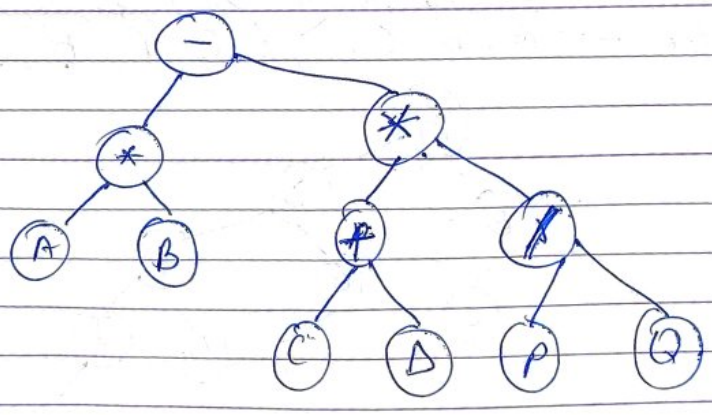
Q Inorder ⇒ Q, B, K, C, F, A, G, P, E, D, H, R.
Preorder ⇒ G, B, Q, A, C, K, F, P, D, E, R, H.
Postorder ⇒ ?

Solution:



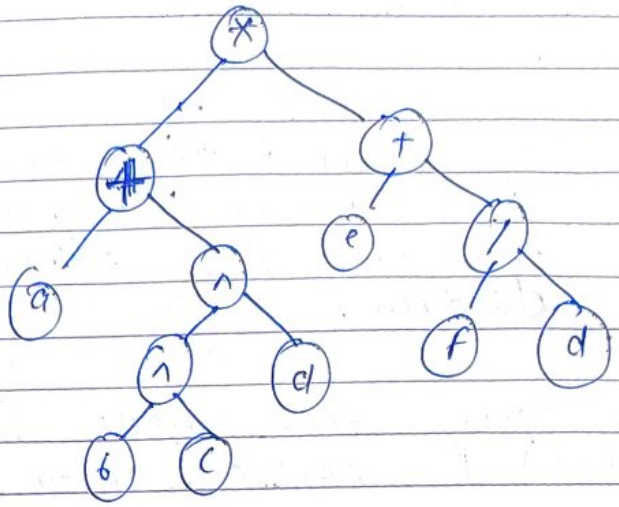
Post - Q, K, F, C, A, B, E, H, R, D, P, G.

Q ⇒ Draw the binary tree of the following expression -
A * B - (C + D) * (P / Q).



Preorder = - * A B.

Q Convert the infix string (a+b^c^d)*(e+f/d). to reverse polish.



a, b, c, ^, d, ^, +, e, f, d, /, +, *.

Q 6+2^3+9/3-4*5.

Imp

B-TREE → Self Balance Tree.

(i) B-Tree is a balanced 'm'-way tree where m is known as order of B-Tree.

(ii) B-Tree is a generalization of binary search tree (B.S.T) in which a node can have more than one key elements & more than two children.

(iii) In B-Tree each node maintain data (keys value) in sorted order.

(iv) All leaf node should be at same level

(v) B-Tree of order m has following property-

(a) Every node has maximum 'm' children.
(b) Minimum children: leaf → 0.
Root → 2.
Internal Node → $\left\lceil \frac{m}{2} \right\rceil$

ex- $\left\lceil \frac{5}{2} \right\rceil = \lceil 2.5 \rceil = 3$.

(c) Every node has maximum (m-1) keys.
(d) Minimum keys: root → 1.
all other node → $\left\lceil \frac{m}{2} \right\rceil - 1$.

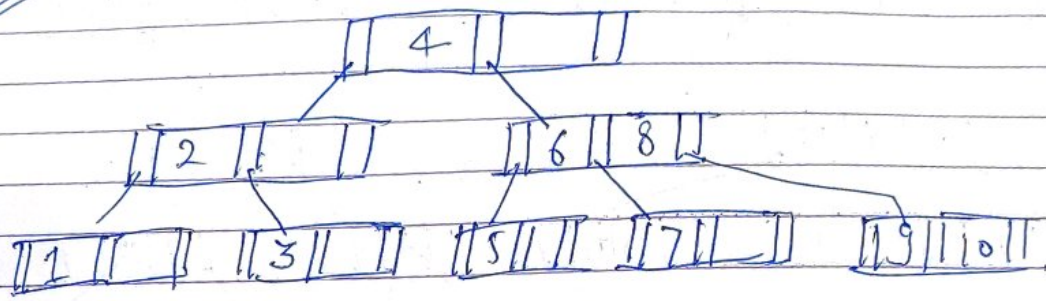
(vi) Every node would be inserted on leaf.

NOTE - B-Tree will grow upward

Q Construct a B-Tree of order 3. by inserting values 1 to 10.

Solution -

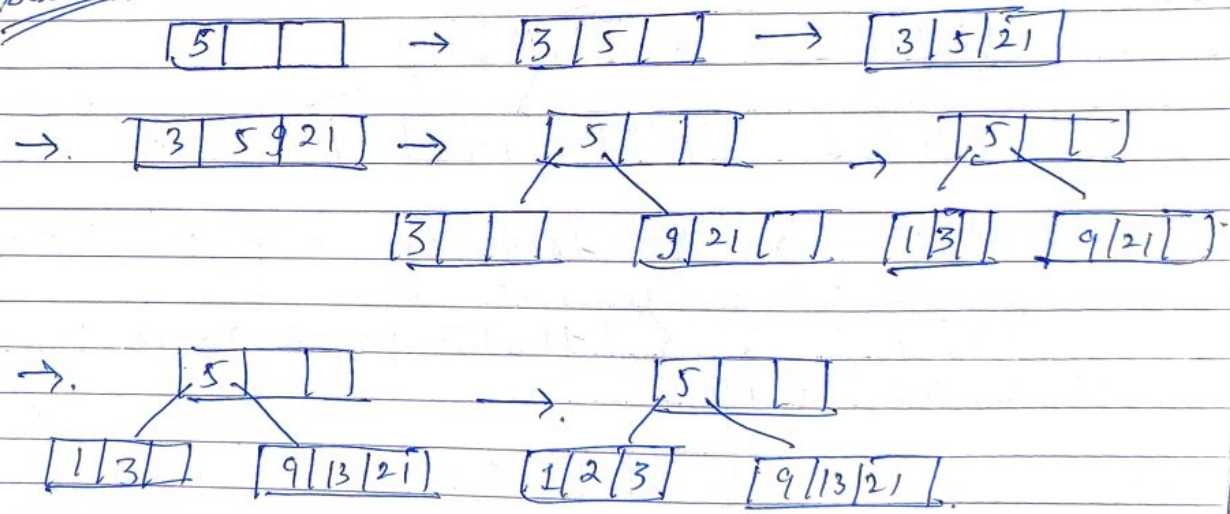
$m = 3$, $K = 2$.

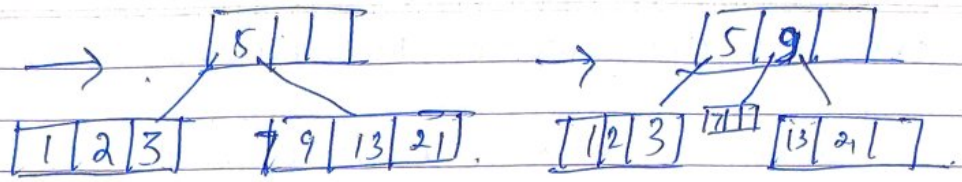


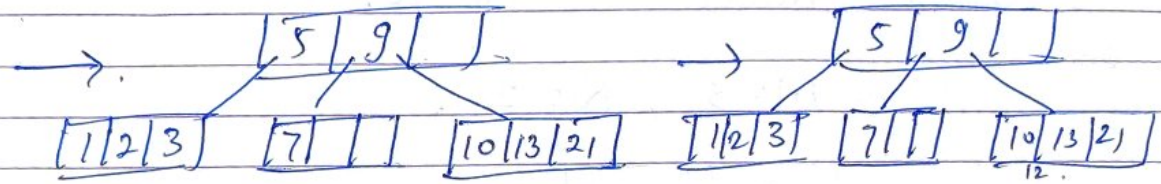
Q Construct a B-Tree of order 4 with following element -

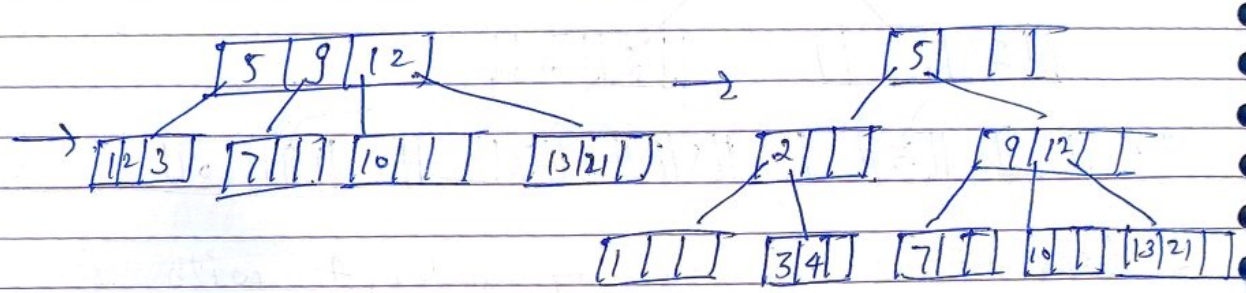
5, 3, 21, 9, 1, 13, 2, 7, 10, 4, 8.

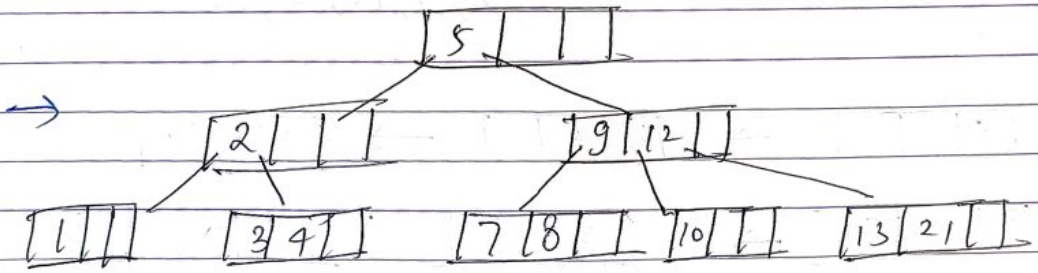
Solution -





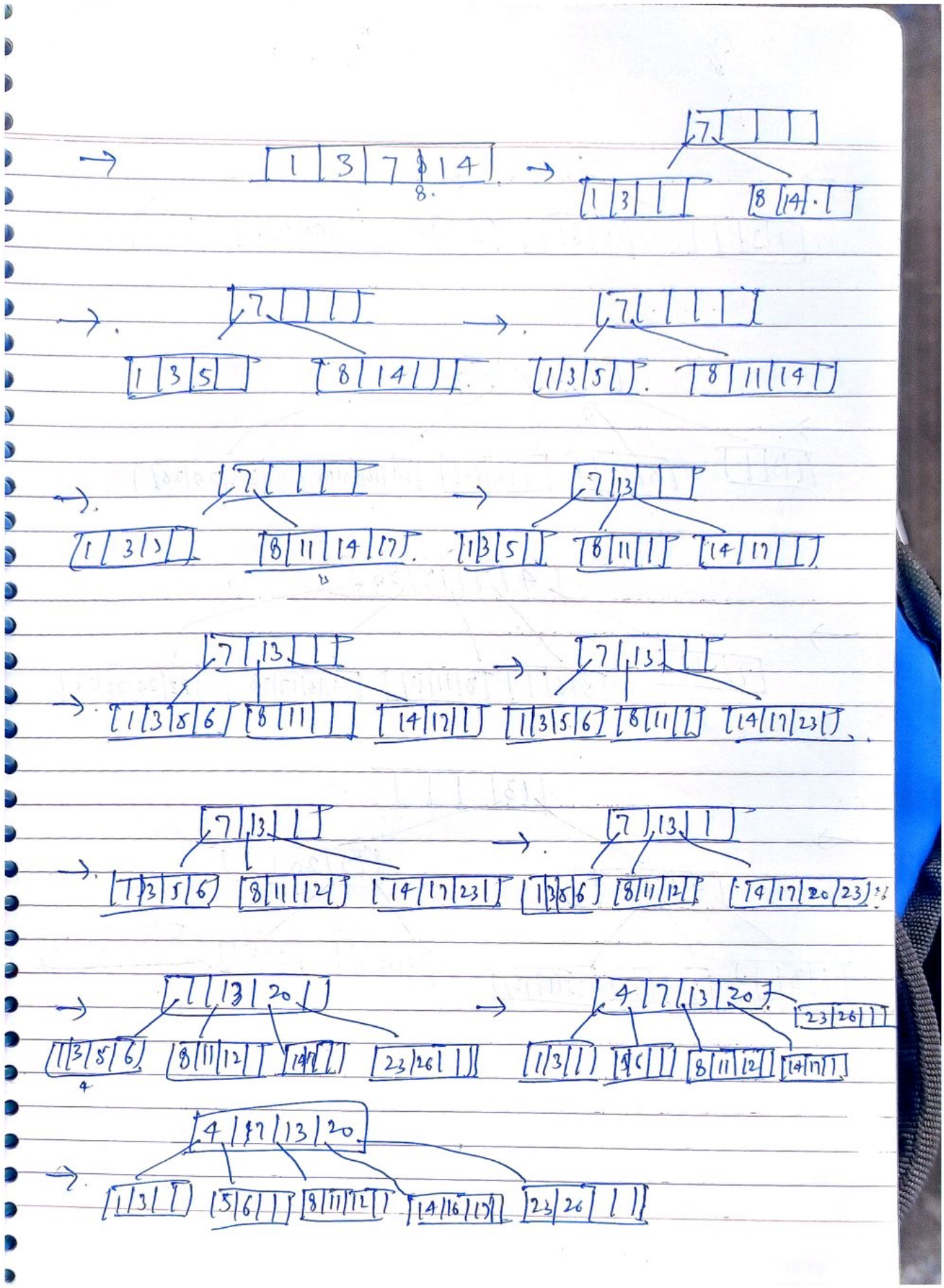


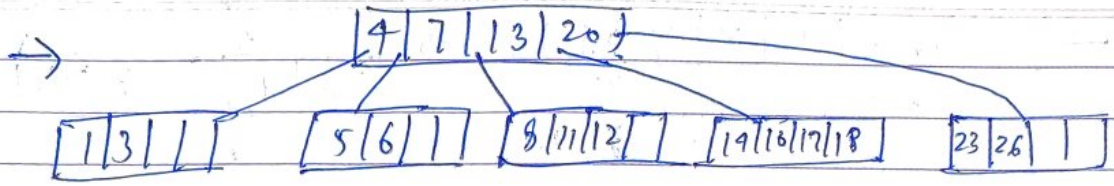


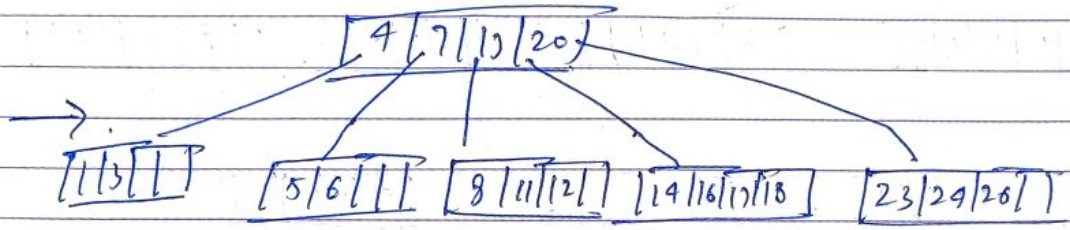


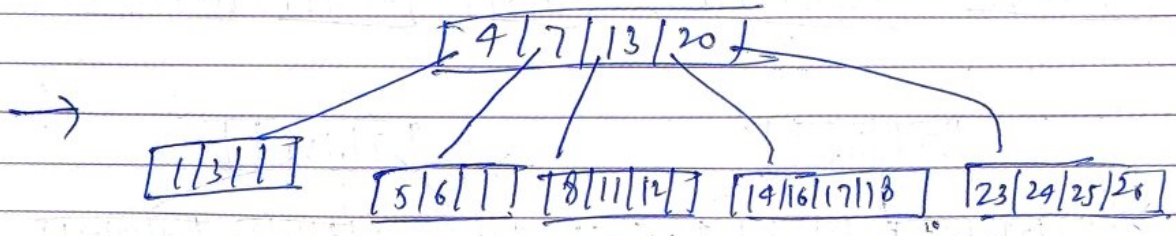
Q Construct a B- Tree of order 5 inserting the following sequence elements.
3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19.

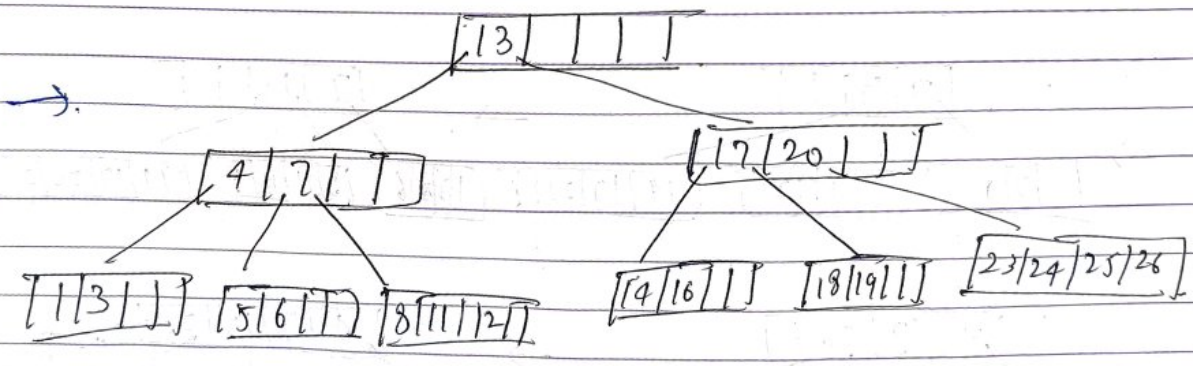
⇒



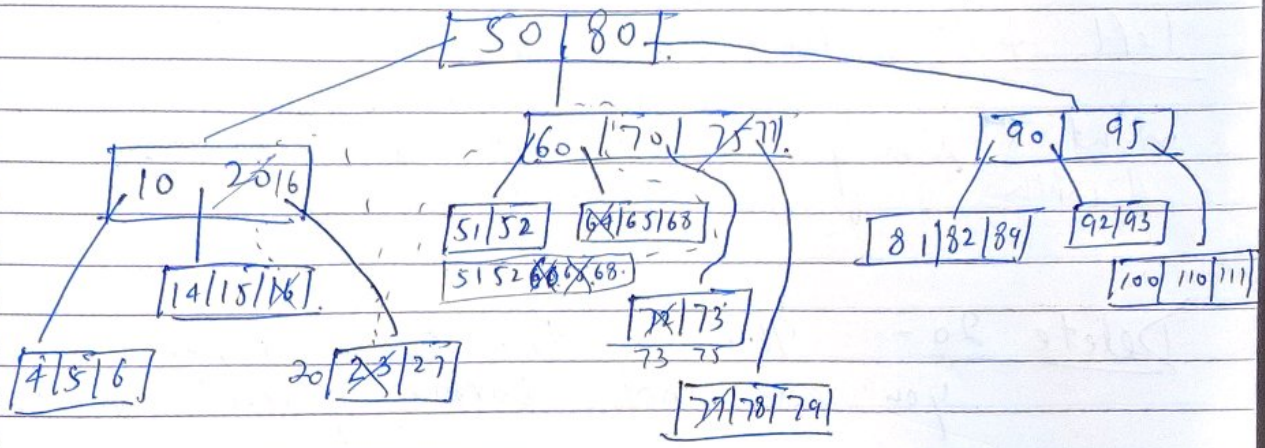








Q Deletion Operation on B~~inary~~ Tree.



B. Tree of order 5.

Min. Child = $\lceil \frac{m}{2} \rceil = 3$.

Max. Child = $m = 5$.

Min. Key = $\frac{m}{2} - 1 = 2$.

Max. Key = $m - 1 = 4$.

The value can found on

1. Root.
2. Intermediate.
3. Leaf.

Leaf → More than min. Keys. / Exactly min Key.

Exactly min Key → Left Sibling borrow / Right Sibling borrow.

Left & Right both having min Key.

Left Borrow –

Left Sibling max. value will move to its parent & parent Key will move to target node & then delete the target node.

Right Borrow –

Right Sibling min. value will move to its parent & parent Key will move to target node in sorted order &.

delete the target node.

Left & Right Borrow :- Merge the target node with its right sibling or left sibling along with its parent.

Delete 20 - This is the case where you can not borrow parent node.