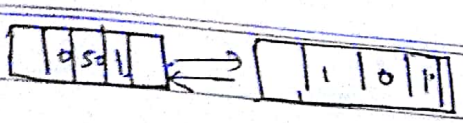
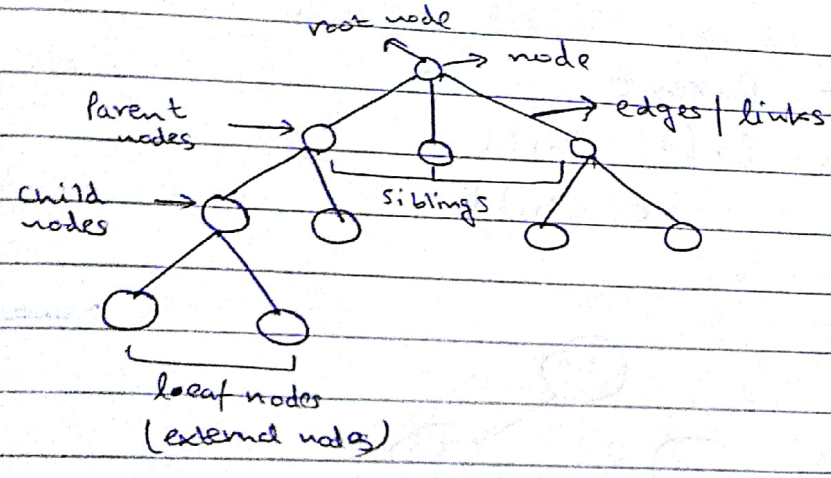


Trees

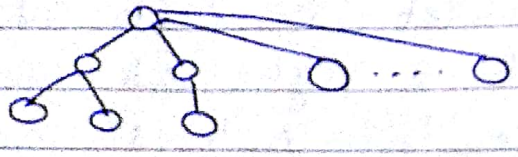
Useful to store hierarchical data.



Height of a tree : The length of the longest path from root node to leaf node

Level of a node : Assuming level of root node is 0, 1 + the level of the parent node is level of a node.

n-array tree : A node can have 0 to n no. of child.



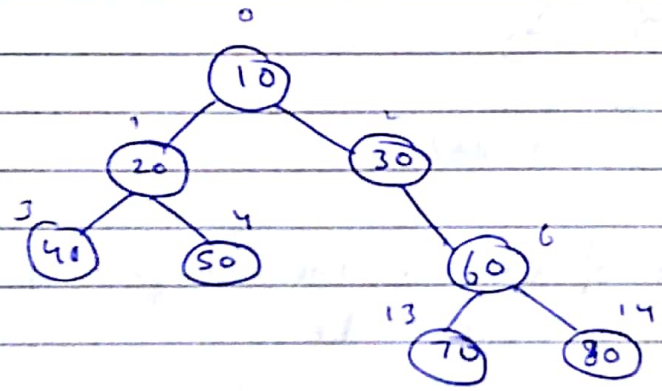
Binary tree : It's a two-ary tree.

Advantages of binary trees

1) It reduces no. of iterations.

Binary trees

n - parent
2n+1 - left child
2n+2 - right child



Binary tree is a finite set of element that can be partition into three disjoint subset. The first subset is root, the second subset is left sub tree, the 3rd subset is right sub tree. The sub trees are binary trees in themselves which can also be empty.

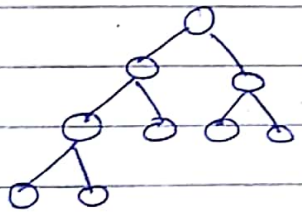
The depth of the tree is max^m level of any leaf node.

→ Types of B T

① Strictly B T : Every node must have 0 or 2 child nodes.

② Complete B.T : Is a strictly B T whose all leaf nodes are at same level

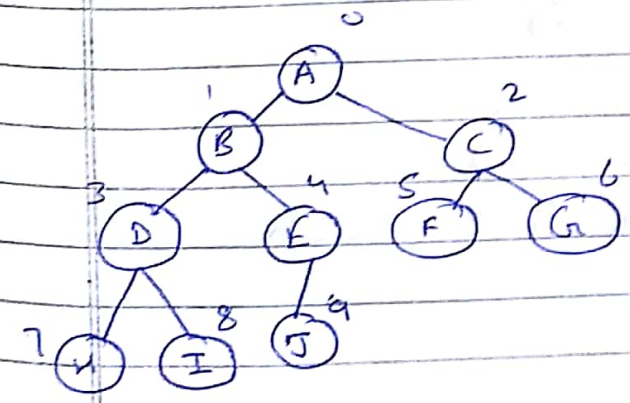
③ Almost Complete B.T :



A B T of depth 'd' is almost complete B.T if :

① Any node at level less then (d-1) has to sons. 'n'

② For any node, in the tree with right descendat at level 'd', n must have left sons. And every left descendent of 'n' is either a leaf at level or has to sons.



A B.T

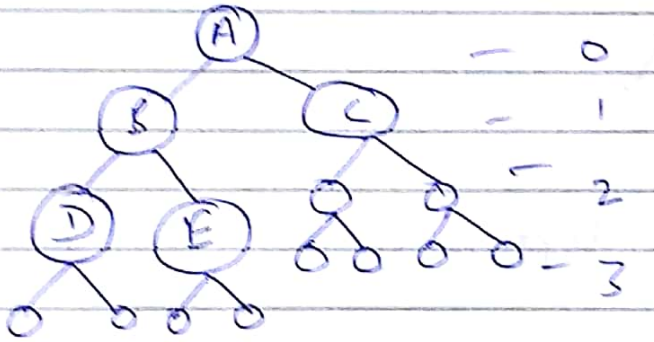
Complete

→ A B.T of depth 'd' will have $2^{d+1}-1$ no. of nodes

→ A complete B.T of depth 'd' will have 2^d leaf.

→ A complete B.T of depth 'd' will have $2^d - 1$ no. of internal nodes.

→ A complete B.T of T_n (total no. of nodes), the max^m depth will be equal to $\log_2(t_n+1) - 1$



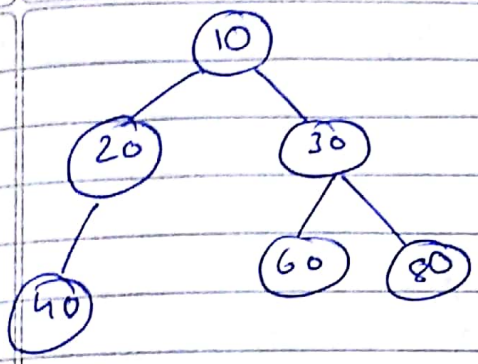
$t_n = 2^{3+1} - 1 = 16 - 1 = 15$

leaves $= 2^d = 2^3 = 8$

internal nodes $= 2^d - 1 = 2^3 - 1 = 7$

Traverse

① inorder - L (RT) R
② preorder - (RT) L R
③ post order - L R (RT)
④ Level order → Breadth first search



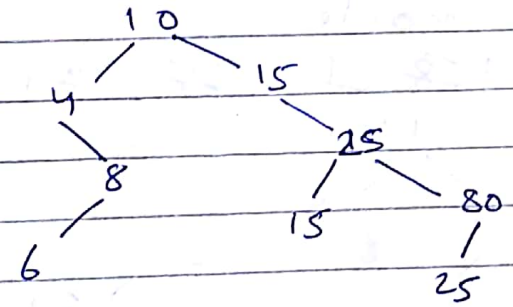
Inorder

40 20 10 60 30 80 → Inorder
10 20 40 30 60 80 → Pre order
10 20 30 40 60 80 → level order

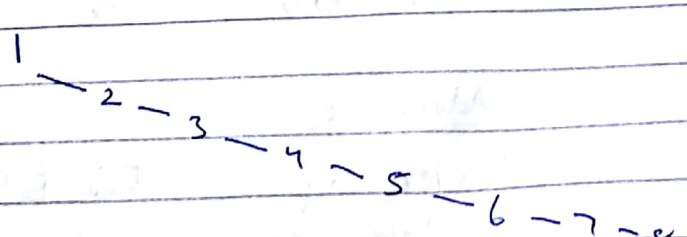
Binary search tree

nodes on left < root
nodes on right => root

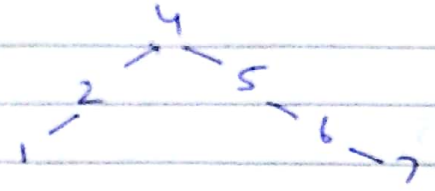
10, 4, 8, 15, 25, 80, 6, 15, 25



1, 2, 3, 4, 5, 6, 7, 8



4, 5, 2, 1, 6, 7



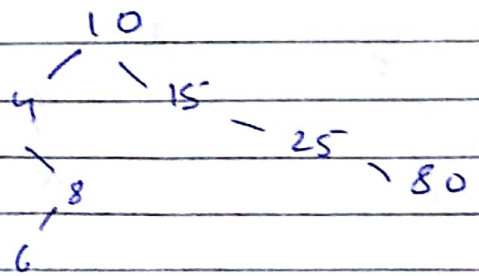
To handle duplicate there are two approaches

1) ~~Eg~~ Create a node on right i.e nodes on right => root. For eg adding 15, 25 two times in tree (~~back~~ previous page).
The drawback is that
   ① It increases height of the tree thereby making the operations inefficient.

2) Use a count variable which takes count of every duplicates.

Q Convert in inorder

~~10, 4, 8, 15, 25, 80, 6~~



4 6 8 10 15 25 80

Advantages

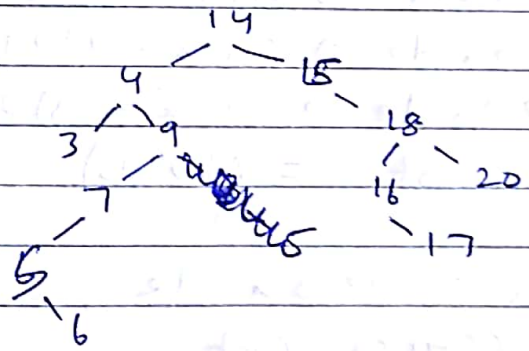
→ Traversing of ~~this~~ BST "inorder" will give ascending order arrangement of element.

→ Inerst, delete, search operations are efficient of order log n.

Disadvantage

→ If the i/p for BST is sorted then it generates a skewed tree.
Skewed tree is inefficient ~~or~~ because it requires more no. of comparison.

Q Construct a BST for i/p 14, 15, 4, 9, 7, 18, 3, 5, 16, 6, 20, 17.



Operation BST

```
Struct node
{
    int data;
    struct node * l link;
    Struct node * r link;
};
typedef Struct node tree;

tree * insert (tree * root, int ele)
{
    tree * newnode;
    newnode = getnode();
    newnode -> data = ele;
    newnode -> r link = NULL;
    newnode -> l link = NULL;
    if (root == NULL)
{
    root = newnode;
    return root;
}
    if( ele < root -> data)
    root -> l
```

```
tree * insert (tree * root , int ele)
{
    tree * temp;
    if ( root == NULL)
    {
      temp = getnode ;
      temp -> data = ele ;
      temp -> l link = NULL ;
      temp -> r link = NULL ;
       root = temp ;
        return root ;
    }
    if( ele < root -> data)
      root -> l.link = insert( root -> llink,ele);
    else if (ele > root -> data)
      root -> r link = insert( root -> r link, ele);
    else
     printf(" Duplicate node");

    return root ;
}

void
tree * search ( tree * root , int key)
{
    if ( root == NULL)
    {
      Pf (" Unsuccessful search");
        return ;
    }
    if ( key < root -> data)
        search (root -> llink , key) ;
    else if ( key > root -> data)
        search ( root -> r link , key);
```

```
else
    pf("Successful");
}
```

```
void preorder(tree * root)
{
    if(root != NULL)
    {
        visit(root);        → pf("%d", root->data);
        preoder(root->llink);
        preorder(root->right);
    }
}
```

```
void inorder(tree * root)
{
    if(root != NULL)
    {
        inorder(root->left);
        vist(root);
        inorder(root->right);
    }
}
```

```
void postorder(tree * root)
{
    if(root != NULL)
    {
        postorder(root->left);
        postorder(root->right);
        vist(root);
    }
}
```

The visit funct^n here can be generic funct^n here which can be used to perform diff. funct^n like,

1. Display
2. Counting no. of nodes
3. Finding sum of all elements in tree
4. Incrementing the node values

Q Write a funct^n to count the no. of nodes in a tree.

A-
```
int preOrder( tree * root , int * count)
{
  if(root != NULL)
  {
    *count = *count + 1 ;
    preorder( root -> left , count)
    pre-order( root -> right , count)
  }
}
```

→ ~~Write a~~ ~~to find~~ min. value in ~~right~~ "left" of tree

```
Node * min( Tree * t)
{
  Node * temp , * prev ;   temp = t ;
  while( temp != NULL)
  {
    prev = temp ;
    temp = temp -> left ;
  }
  return prev ;
```

```
Node * min(tree *t)
{
  Node * temp, * prev;
     temp = t;
     prev = temp;
    while (temp != NULL)
{
     prev = temp;
     temp = temp -> left;
}
   return prev;
}
```

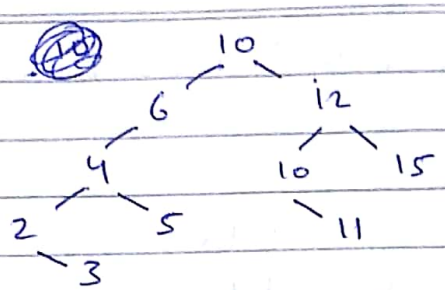
→ Max in right

```
Node * max(temp * t)
{
  Node * temp, * prev;
     temp = t;
     prev = temp;
    while( temp != NULL)
{
     prev = temp;          right
     temp = temp -> ~~left~~;
}
     return prev;
}
```

Detetion

replace with [in order Successor
inorder predessor (max value on left sub tree

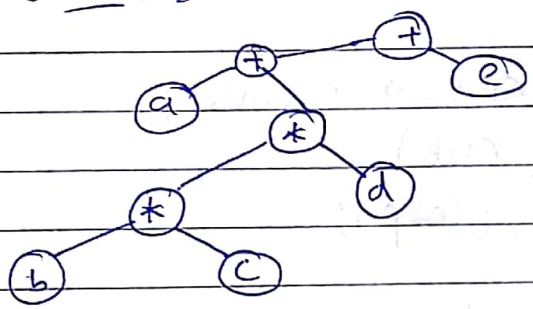
```
Tree * delete (Tree * root , int key)
{
  Tree * temp ;
   if (key > root → data)
     root → r link = delete( root → r link , key );
   else if ( key < root → data)
      root → l link = delete( root → l link, key);
   else if ( root == NULL)
        printf (" key not found");
   else if ( key == root → data)
{
    if ( root → l link == NULL)
  {
     temp = root → r link ;
      free ( root);
      return (temp);
}
  else if ( root → r link == NULL)
{
     temp = root → l link ;
       free ( root) ;
         return ( temp);
}
   else
{
     temp = min ( root → r link);
     root → data = temp → data ;
     root → r link = delete ( root → r link ,
                              temp → data );
}
    return root;
}
```



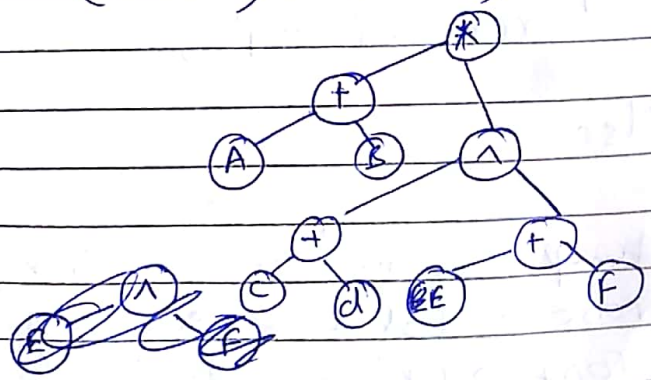
delete (root, 6)

Expression trees

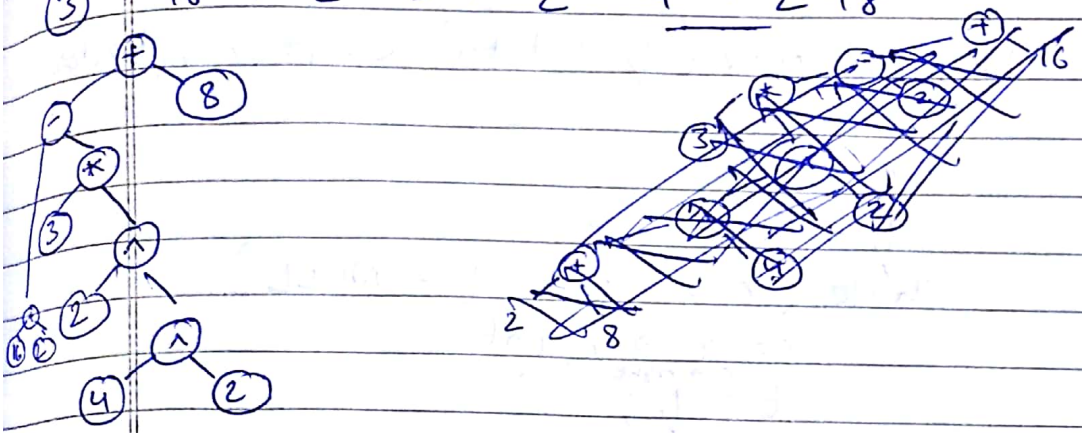
① a + b * c * d + e



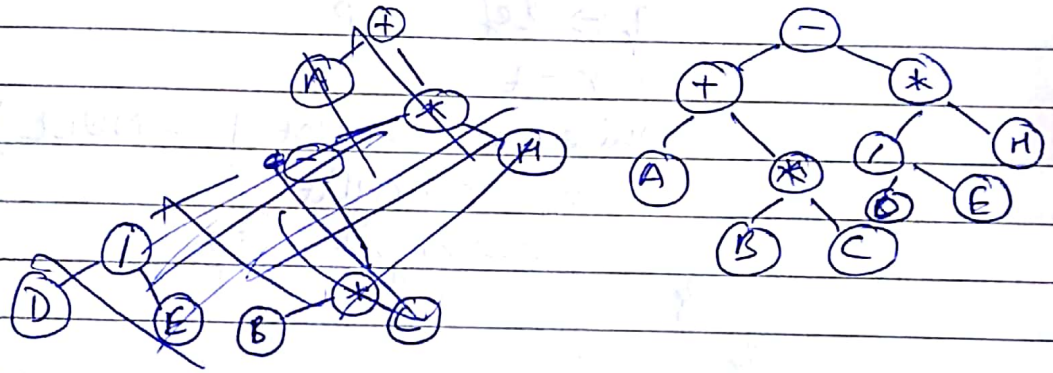
② (A+B) * (C+D) ^ (E+F)



(3) 16 + 2 - 3 * 2 ^ 4 ^ 2 + 8



(4) A + B * C - D / E * H



Program :

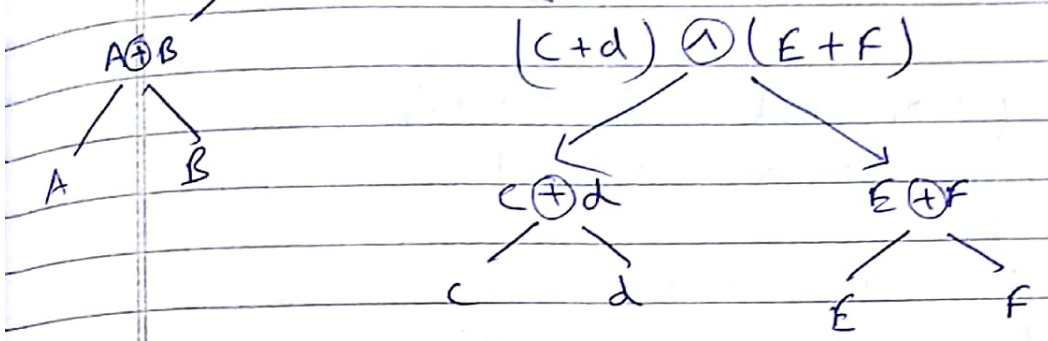
```
tree insert ( Char a[ ])
{
  int i = 0 ;
  tree p, q, r, t = NULL;
  while (1)
{
  p = Maketree ( a[i++]) ;
  q = Maketree ( a[i++]) ;
   if ( q -> data != '\0')
{
     if (t != NULL)
```

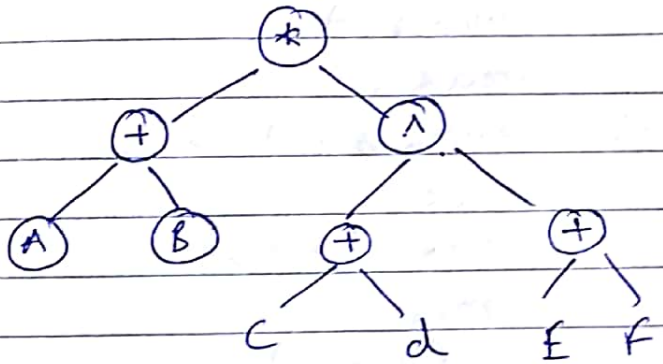
```
{
  if (priority(q->data) <= priority(t->data))
  {
    r = t;
    q->left = t;
    while (r->right != NULL)
      r = r->right;
    r->right = p;
    t = q;
  }
  else
  {
    q->left = p;
    r = t;
    while (r->right != NULL)
      r = r->right;
    r->right = q;
  }
}
else {
    q->left = p
    t = q;
  }
}
  else {
    r = t;
  while (r->right != NULL)
    r = r->right;
    r->right = p
    return t;
  }
}
```

1 5 2 4 3

① (A + B) * (c + d) ^ (E + F)

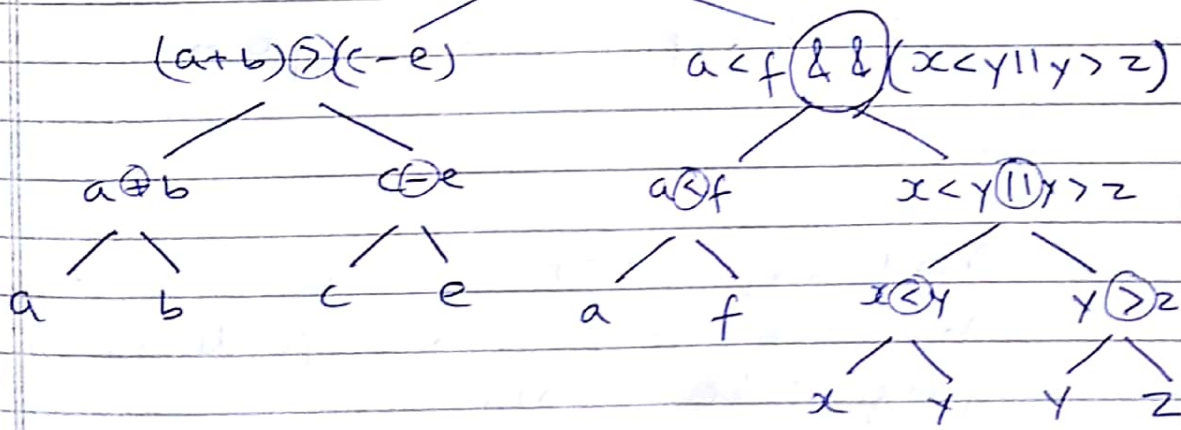
(Divide in two with lowest priority)





1 3 2 9 7 8 6 4 6 5

② ((a+b) > (c - e)) || a < f && b (x < y || y > z)



→ Solving tree

```
int solve (tree * root)
{
  if( isoperand (root -> data))
        return root -> data;
  else {
    A = solve( root -> left);
    B = solve( root -> right);
  switch (root -> data)
  {
    case '+' : return (A+B);
               break;
    case '-' : return (A-B);
               break;
    Case '*' : return (A * B)
               break;
    case '/' : if (B != 0)
               return (A/B);
             else
               pf(" Division not possible");
               break;
    default: pf("Wrong option");
  }
}
```

Disadvantage of BST : If BST is skewed, the efficiency of the operations is going to be O(n).

AVL trees : Balanced Binary search trees

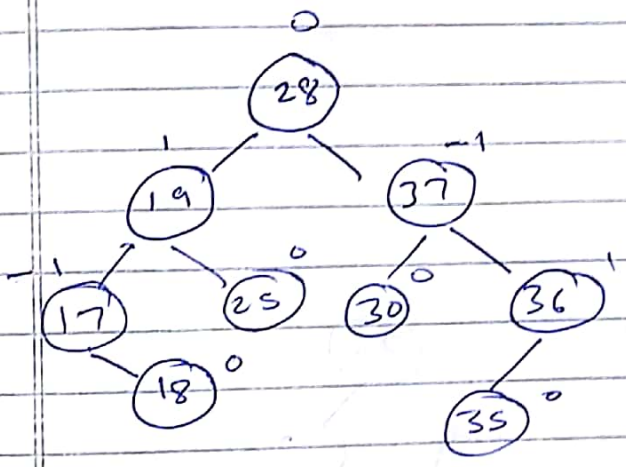
~~bane~~ Balance factor = height ((left subtree)

balance factor = height (left subtree) - height (right subtree)

↓
(should never be more than 1)

{ 0, +1, -1 }

(1)



→

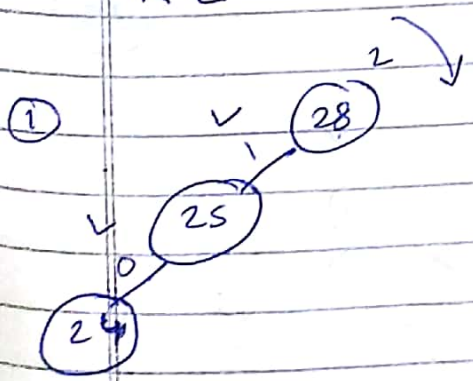
L L

R R

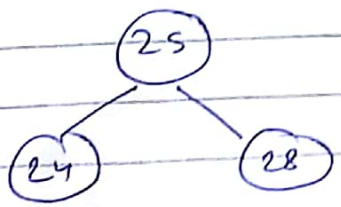
L R

R L

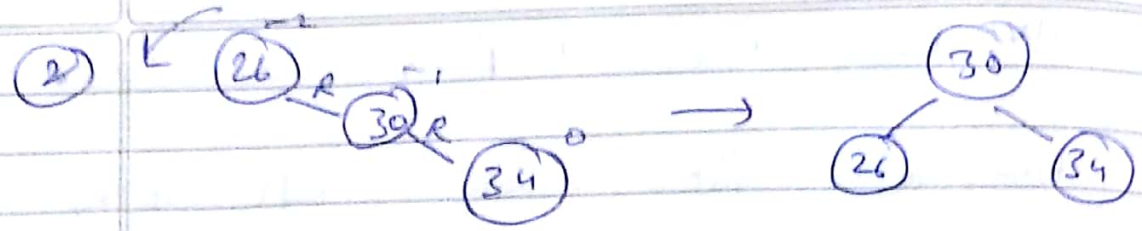
(1)



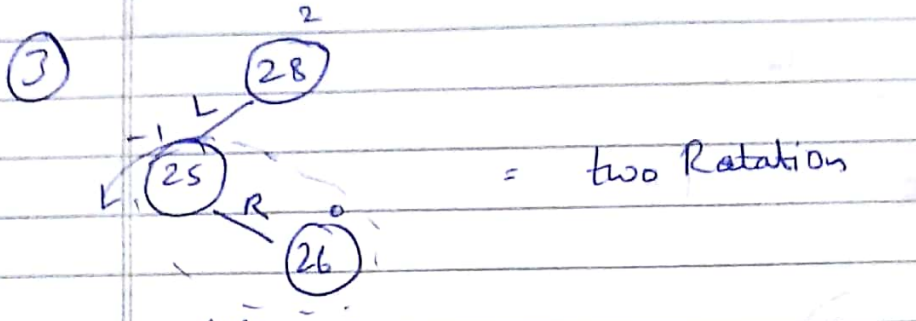
→

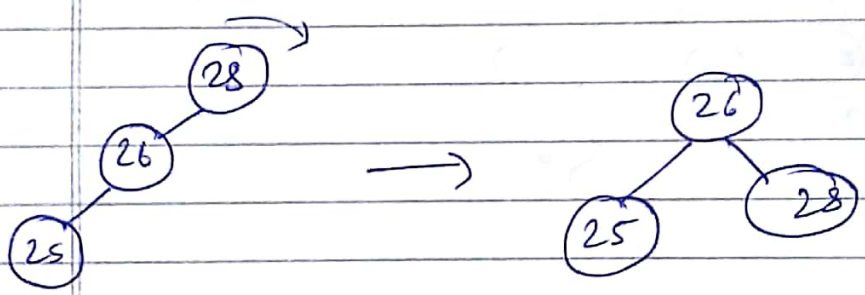


2

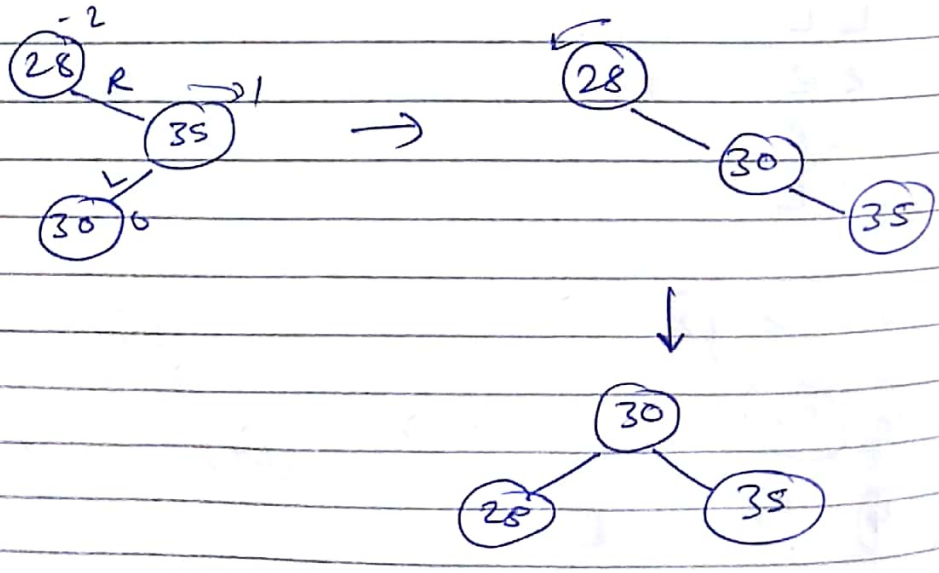


3



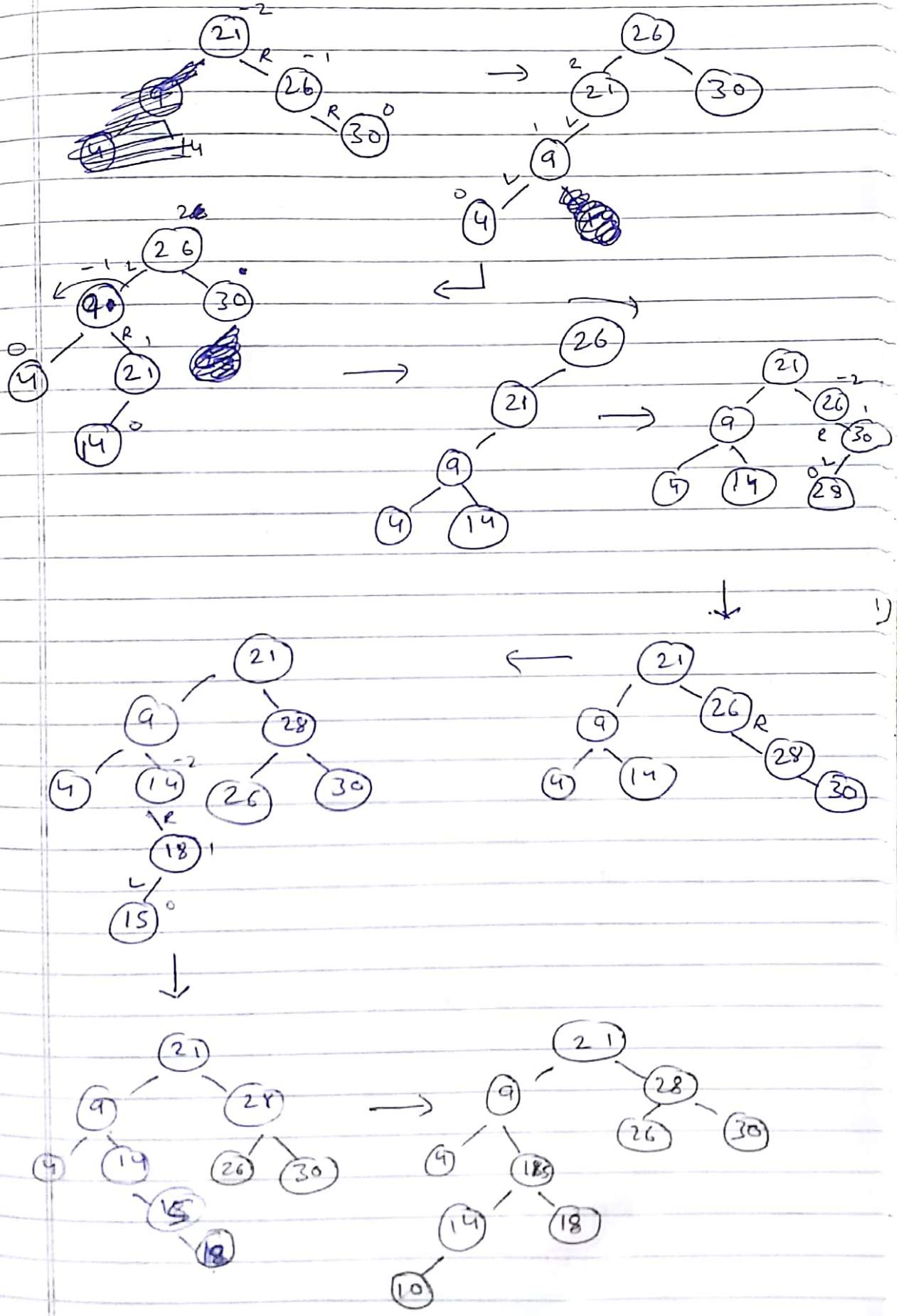


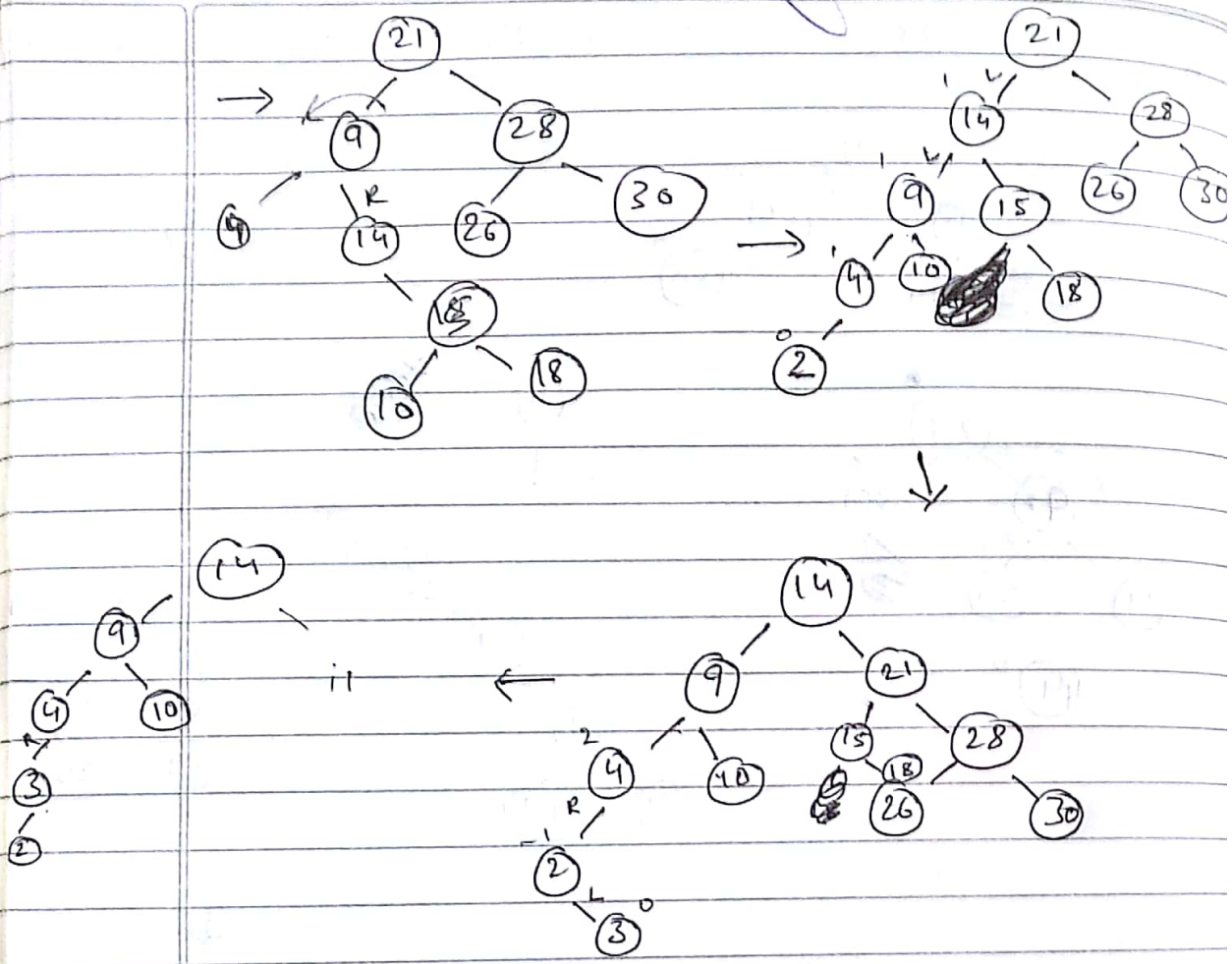
4

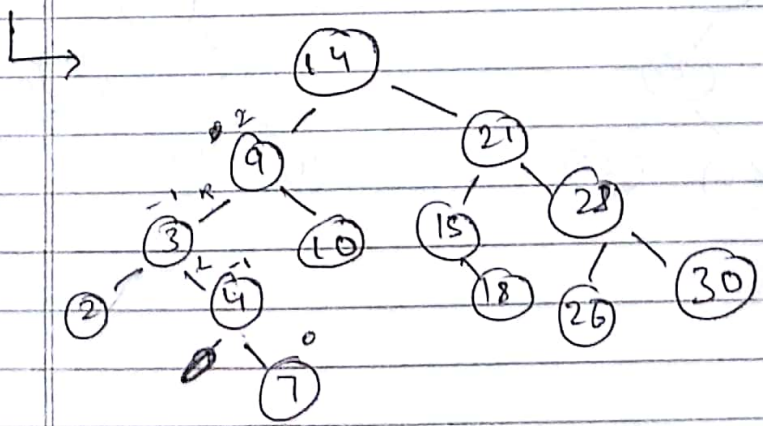


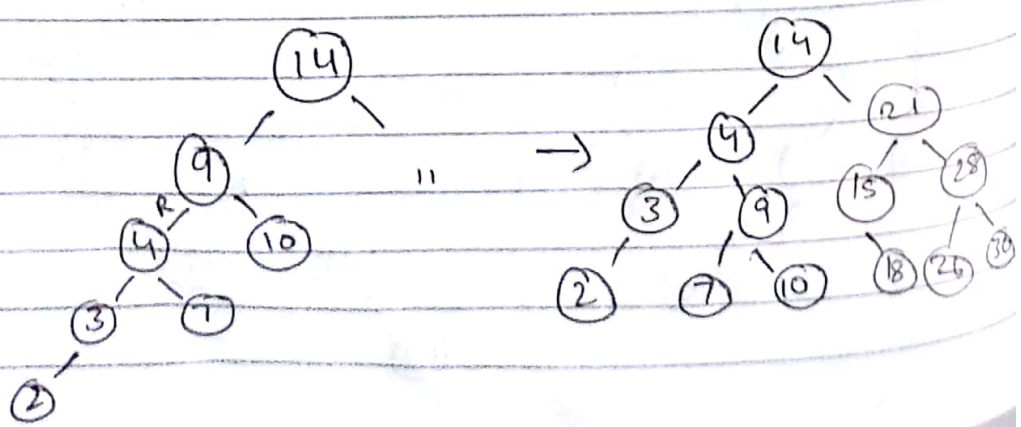
Q 21 26 30 9 4 14 28 18 15 10 2 37

A-

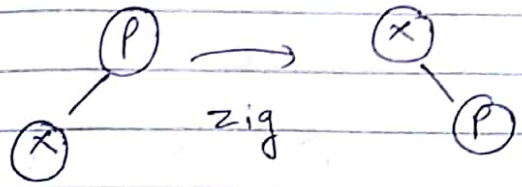


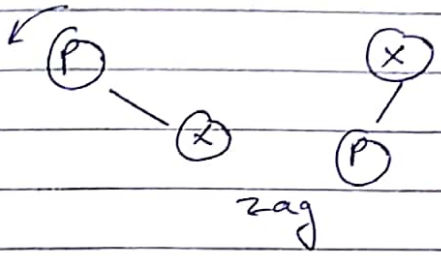


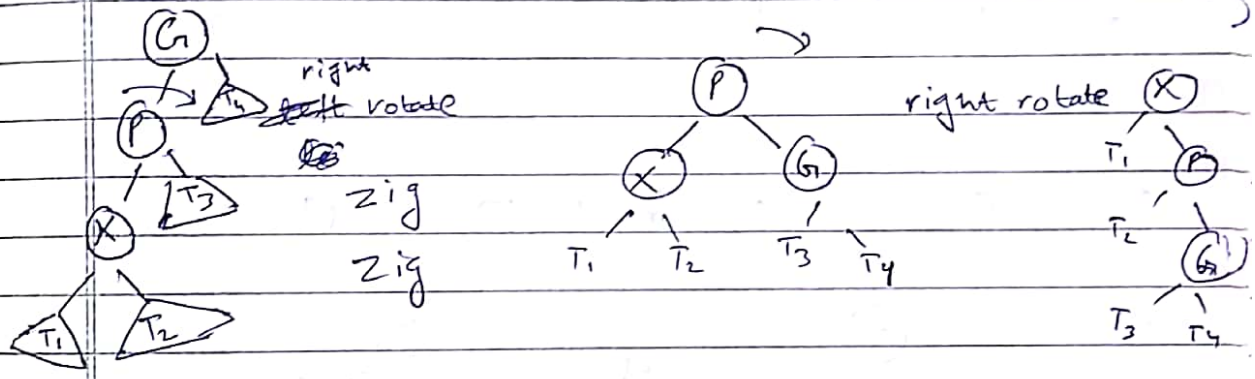




→ Splay trees

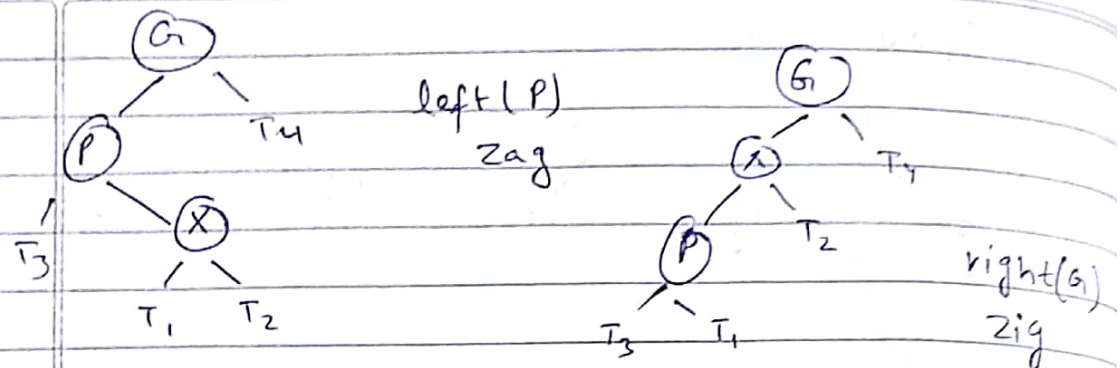


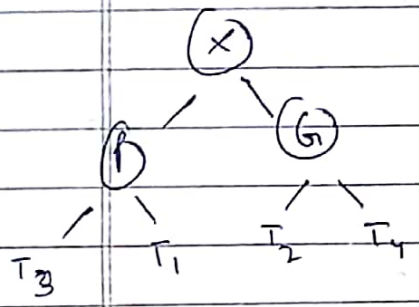




Splay trees are BST in which the recently accessed node is going to be splayed (move) to the root so that the subsequent operatⁿ on that node ~~is~~ takes const. time. Splay trees are based on the concept that the recently accessed element is likely to be accessed again. Here we have 6 types rotation:

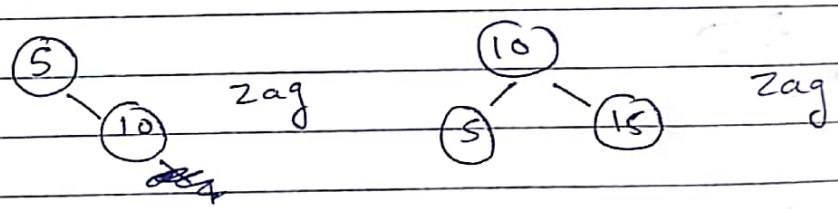
1) zig → R rotⁿ
2) zag → L rotⁿ
3) zig-zig → RR
4) zag-zag → LL
5) zig-zag → RL
6) zag-zig = LR

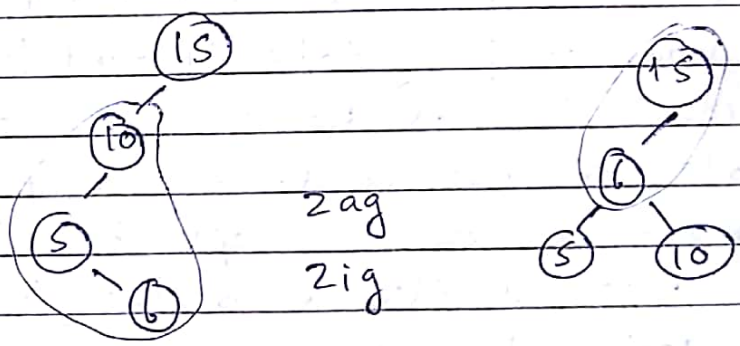


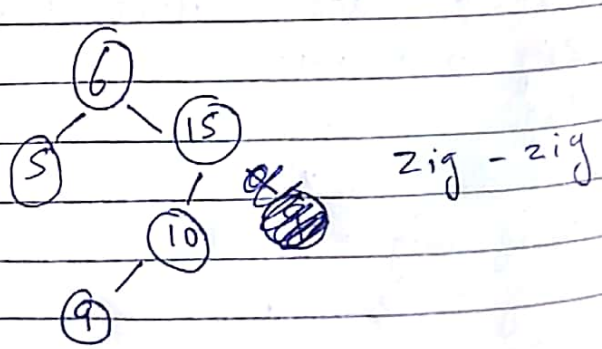


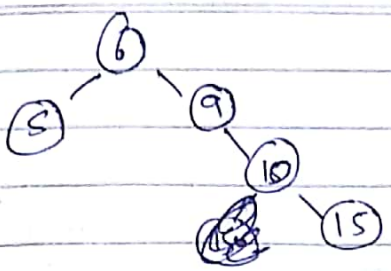
Q 5, 10, 15, 6, 9, 17, 11, 19, 12

A

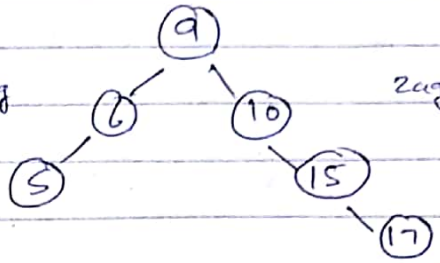




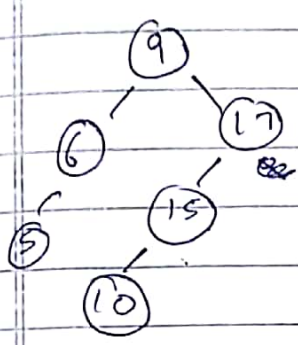




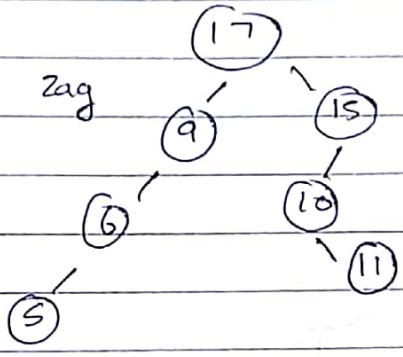
Zag Zag



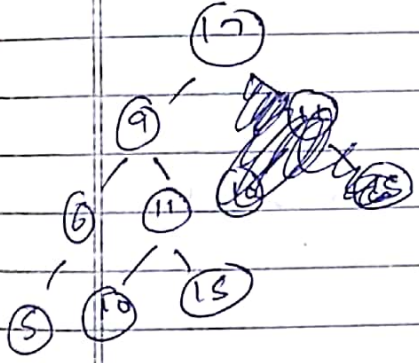
Zag Zag



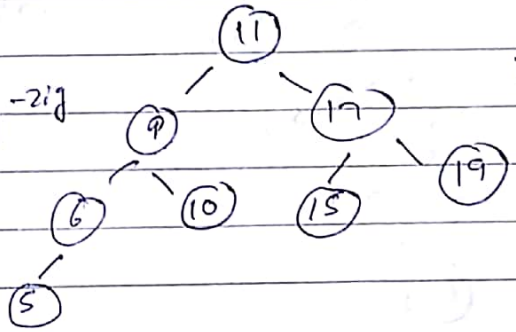
Zag



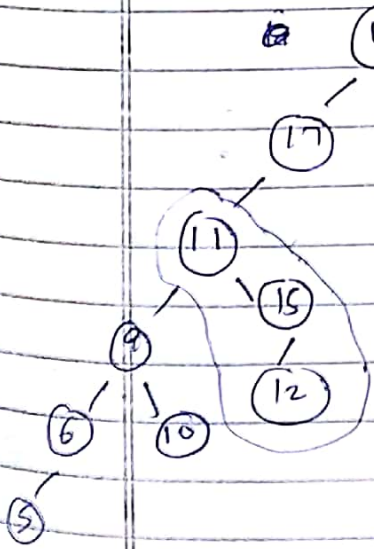
Zag - Zig



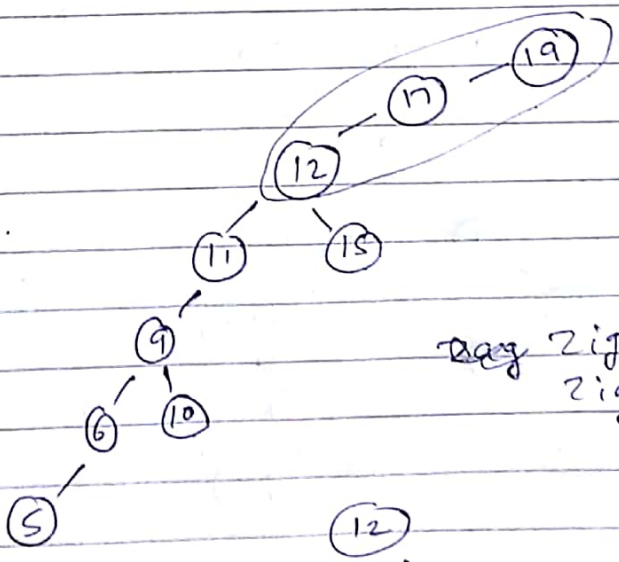
Zag - Zig



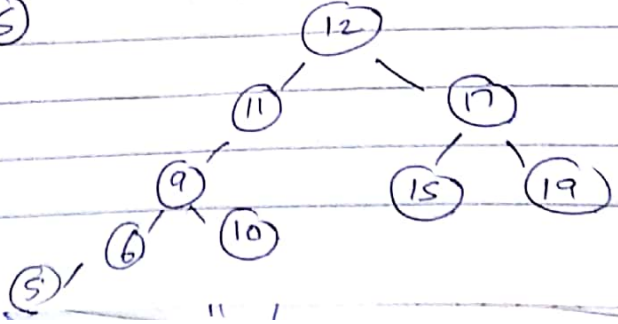
Zag Zag



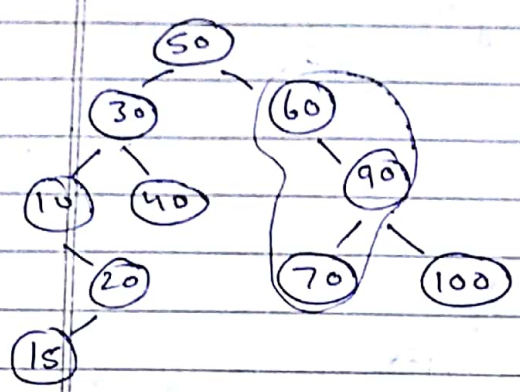
Zig
Zag



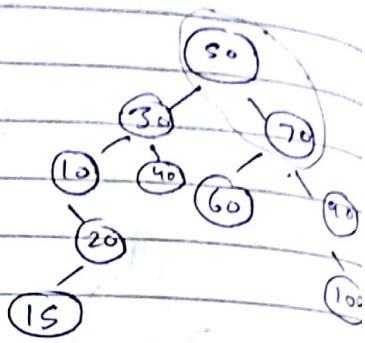
~~Zag~~ Zig
Zig



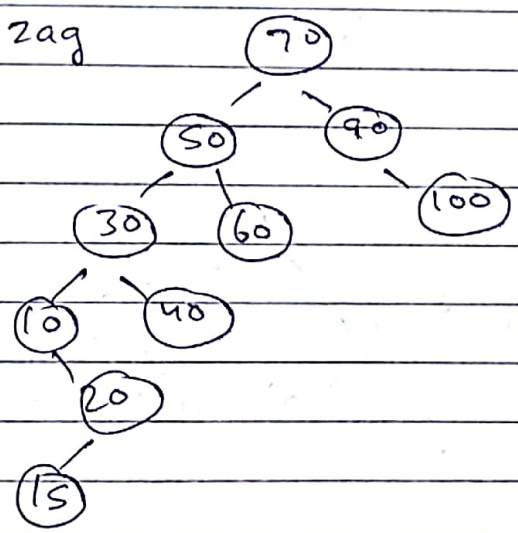
8 Search (80)



Zig - Zag

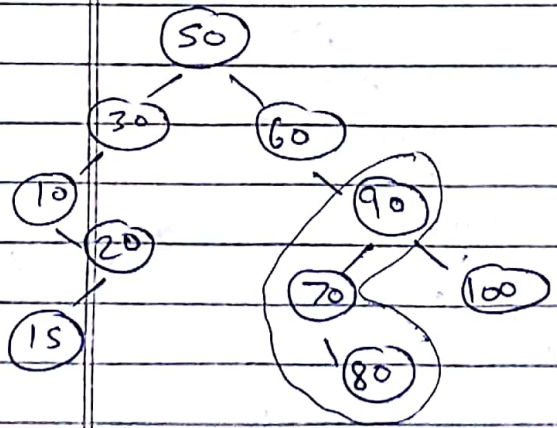


Zag

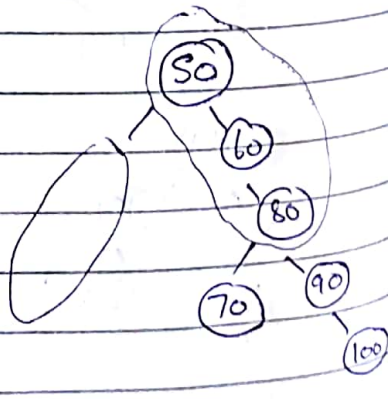


Not found

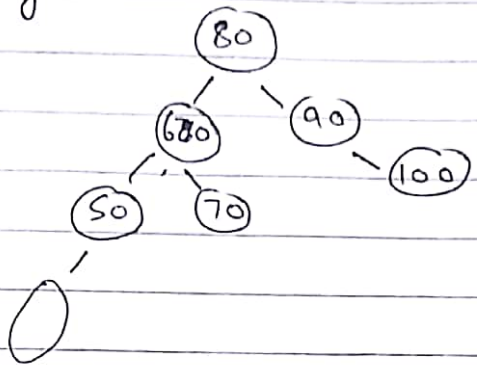
9 insert 80



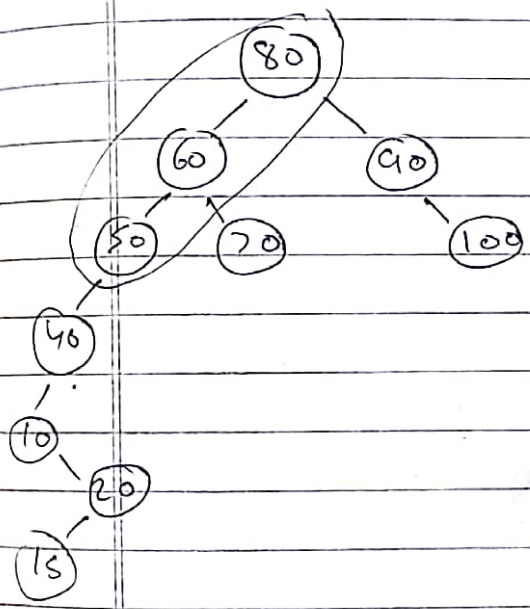
Zag zig



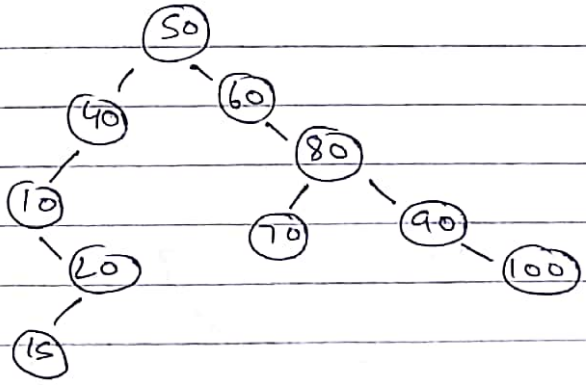
zag - zag



Q splay (50)

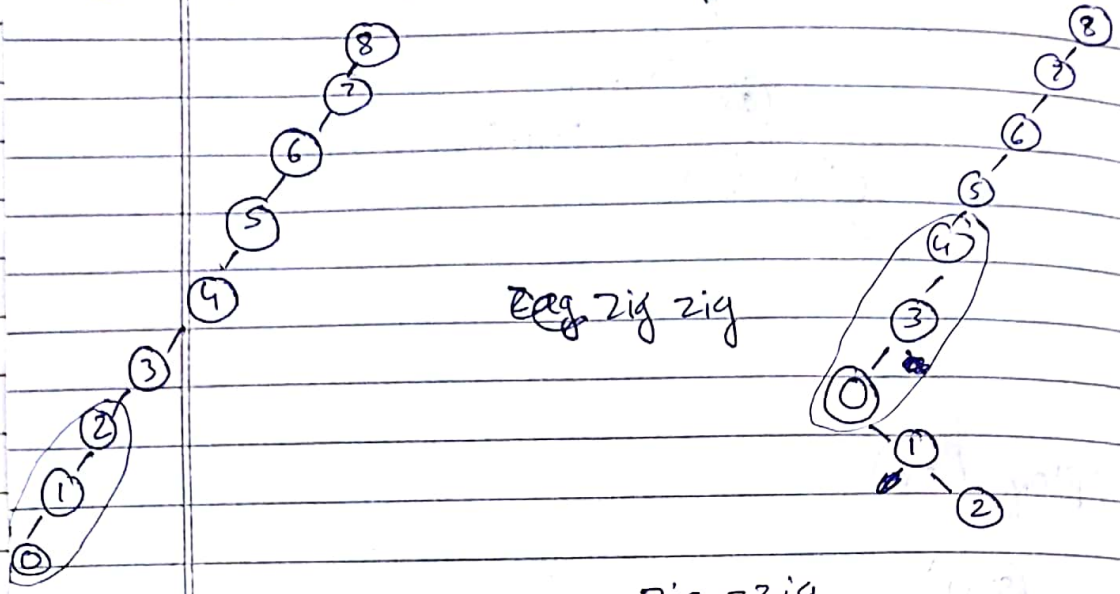


zig - zig



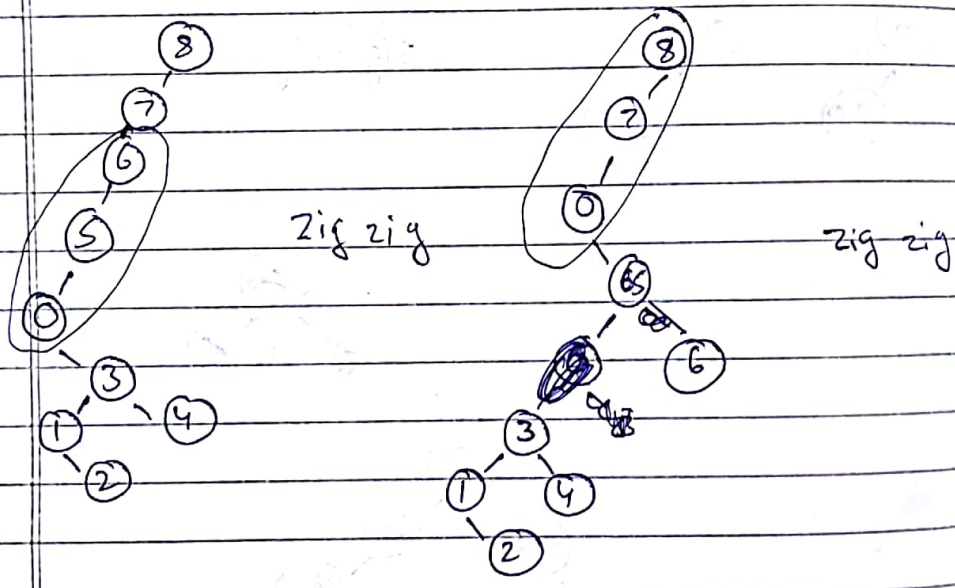
SURYA Gold
Date _____ Page _____

Q find(0)



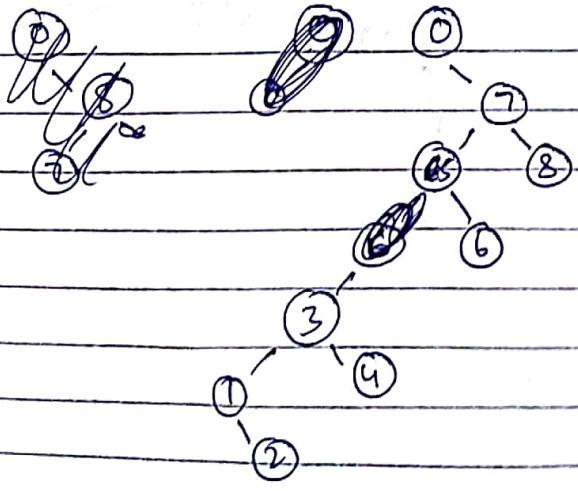
zig zig

zig-zig

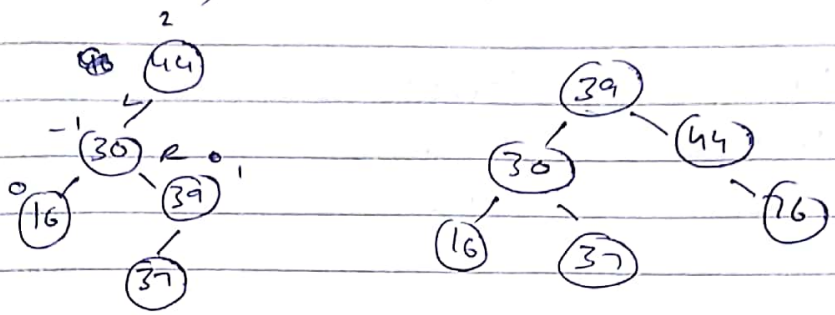


zig zig

zig zig



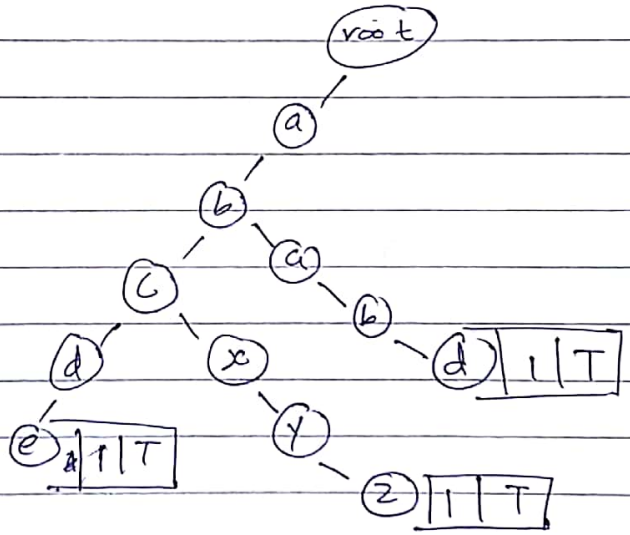
insert (37) (AVL)



Trie

data	count	End of string	Pointer

ab cde
abc xy
ab abd
abc xyz



Searching

Deletion

Decrement the count
if c = 0 f
if there is child node we cannot delete the node