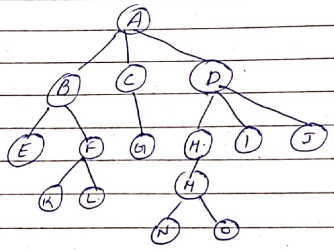
10/9/19

Tree

It is a collection of elements called nodes one of them is k/a is root node & represented by 'r'. The root node 'r' can have 0 or more non-empty subtree $T_1, T_2 \dots T_K$

The root node can be connected by a direct edge to the subtree. The root of each subtree is said to be a child of root 'r' & 'r' is a parent of all those subtrees.

Various terms used in Tree.



1) **Root**: It is a special node in a tree structure & the entire tree is referenced through it. This node does not have any parent. In the given fig. A is root node of the tree.

2) **Parent**: It is an immediate predecessor of a node. A is a parent of B, C, D.

3) **Child**: All immediate successor of a node are its children. For eg: B, C, D are children of A.

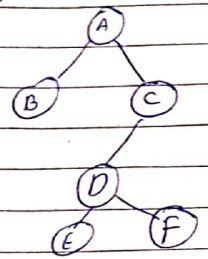
4) **Siblings**: Nodes with the same parents are k/a siblings eg: H, I, J are siblings because they have same parents node D.

5) **Path**: It is no. of successive edges from source node to destination node. The path length is a no. of edges from source node to destination nodes. for eg: Path length from A to M is 3
A, D, H, M.

6) **Degree of a node**: The degree of a node is a no. of children of a node.
deg(A) = 3 ; deg(K) = 0

7) **Leaf Node**: A node is said a leaf Node if it do not have any children.

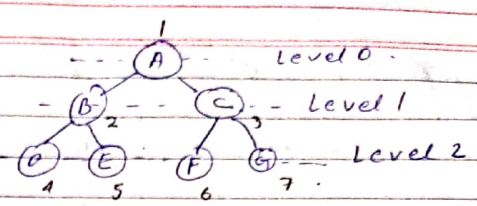
Binary Tree: A tree is Binary if each node of the tree can have maximum of two children. The children of a node in binary tree are ordered (one child is called the left child & other child called right child).

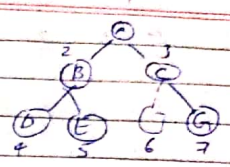


Representation of a Binary Tree:
The rep of Binary Tree can be done in two ways.

1) Array Rep.
2) Linked list Rep.

1. **Array Rep. of Binary Tree**
In order to represent a tree in a single one dimensional array, the nodes are numbered sequentially level by level. left to right. Even empty nodes are also numbered.

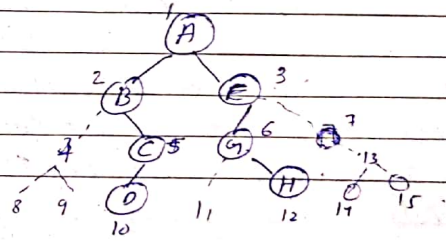




When the data of the tree is stored in an array, then the no appearing against the node will work as indices of the node in an array.
location no - 0 of the array can be used to store the size of the tree in terms of total no of nodes (existing or Non-existing).

7	A	B	C	D	E	F	G
0	1	2	3	4	5	6	7

If there are non-existing nodes.



0															15
15	A	B	E	\0	C	G	\0	\0	\0	D	\0	\0	H	\0	\0

All not existing children are represented by '\0' in the array.

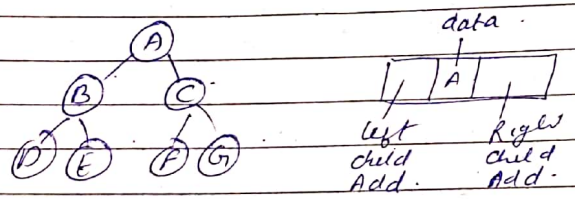
=) Index of the left child of the node 'i' = $2i$.

=) Index of right child of the node 'i' = $2i+1$.

=) Index of the parent node i = $i/2$;

=) sibling of a node i will be found at the location $i+1$ if i is a left child of its parent if i is a right child of its parent, then sibling will be found at $i-1$;

2. Linked list representation of Binary Tree.

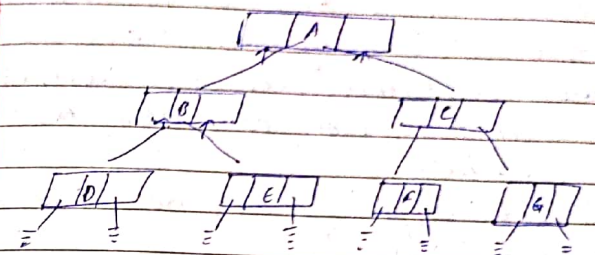


Linked rep. of a Binary Tree is more efficient than array representation A node of a Binary Tree consists of three fields

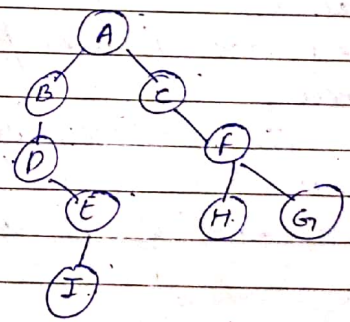
i) Data ii) Add. of left child iii) Add of Right child

typedef struct tree{

int data;
struct tree * prev left;
struct tree * right;
} tree,



Height of a node is the total no. of edges comes under visiting farthest node from that node.



H(F) = 1

(D) = 2.

(A) = 4

Height of a tree is the height of the root of the tree.

Traversal of Tree:

Most of the time requires traversing a tree in any given order. Traversing a tree is a process of visiting every node of the tree exactly once. Since a binary tree is defined in a recursive manner, tree traversal can also be defined recursively.

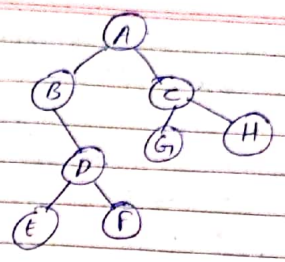
There are 3 types of traversal techniques

1) Preorder Traversal.
2) Inorder Traversal.
3) Postorder Traversed.

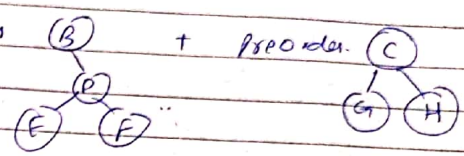
1). Pre-order: The functioning of pre-order traversal of a non empty binary tree is as follows

1) First, Visit the root node [we can print the data field of that node].
2) Next traverse the left subtree in pre-order manner.
3) At the last traverse the right subtree in pre-order.

Ex:



NLR.



A + preorder + preorder → C + Pre (G) + Pre (H)

A + B + D + Pre(E) + Pre(F) + C + Pre G + H.

A + B + D + E + F + C + G + H.

⇒ A B D E F C G H.

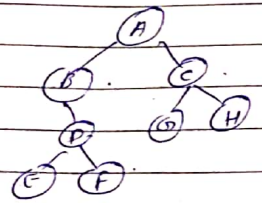
root.

Recursive f(n) of preorder traversal.

```
void preorder (Tree *T)
{
    if (T != NULL)
    {
        printf ("%d", T->data);
        preorder (T->left);
        preorder (T->right);
    }
```

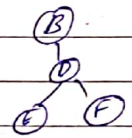
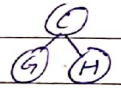
(ii) In-order Traversal: The functioning of inorder traversal of non empty binary tree is as follow

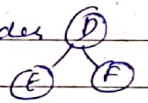
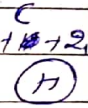
1) Traverse the left subtree in Inorder
2) Next visit the root node
3) Traverse the right subtree in - in-order;



LNR

B E D F A G C H

1) Inorder  + A + Inorder 

⇒ Inorder B + Inorder  + A + Inorder 

B + E + D + F + A + G + C + H

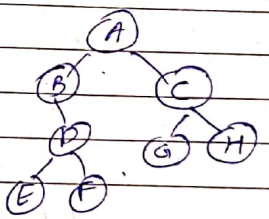
B E D F A G C H.

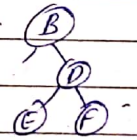
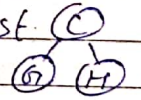
Recursive function

```
void Inorder ( Tree * T)
  { if (T → left)
      if ( T! = NULL)
        printf (" %d ", T → data);
        Inorder ( T → right);
```

(iii) Post Order Traversal: The functioning of Post-Order traversal of a non empty Binary tree is as follows. First

1) Firstly traverse left subtree in post order.
2) Next traverse the right subtree in post order.
3) At last visit the root node.



i) Postorder  + Post  + A

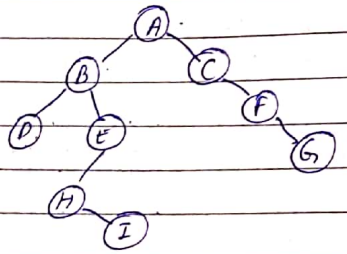
ii) Post  + B + Post G + Post H + C. + A

(iii) E F D B G H C * A.

Post Order Recursive fln)

```
void postorder( Tree *T).
{
  if (T! = NULL)
  {
    post order( T→left)
    postorder( T→right ).
    printf ("%d", T→data),
  }
}
```

Q.

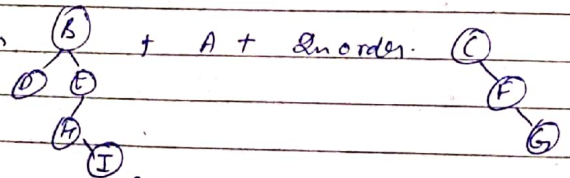


In
L N R

Post
L R N

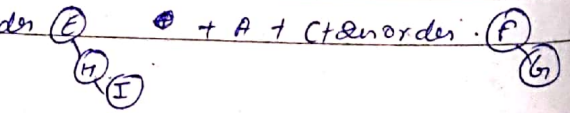
C C B +

1) Inorder (B) + A + Inorder (C)



B +

2) Inorder (D) + Inorder (E) + A + C + Inorder (F)

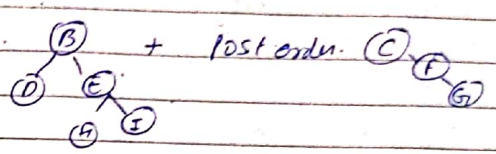


3) (D) + (E) + In order (H) + A + C + In order (F) + (G)

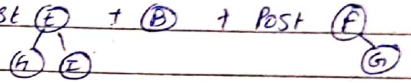
4) D B E H I A C F G.

Post Order

i) Post order (B) + Post order (C) + A



ii) Post (D) + Post (E) + (B) + Post (F) + C + A



iii) D + H + I + E + B + G + F + C + A

D H I E B G F C A.

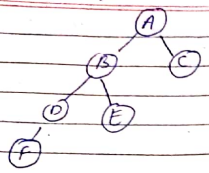
Non-Recursive Pre-order Traversal:

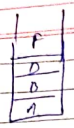
Non-Recursive Traversal can be designed using a stack

ALGORITHM:

STEP 1:- Start Traversing from the root (say 'T')
Traverse left & continue traversing left.
All Nodes traversed are pushed into a stack (say 'S')

Node Left Right





Backtracking

Step I:
```
while (T != NULL)
{
    visit (T);
push (&s, T->data);
    T = T->left;
}
```

Step 2: In previous steps, nodes are saved in the stack for the purpose of backtracking.
Eg: traversal of right subtrees of all node visited so far.

Step II:
```
if the stack S is empty
then traversal finished.
else
[visit the right subtree of
 node popped from stack S]
T = pop(&S);
T = T->right;
[ start traversing from T, Traverse left
All the node traversal pushed into stack);
while (T != NULL)
{
    visit (T);
push (&S, T->data);
    T = T->left;
}
```

C function for non recursive Pre Post Inorder traversal.

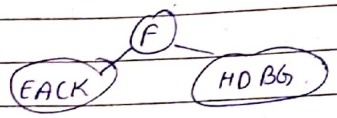
```
void inorder_non recursive (Tree *T)
{
    stack S;
    S.top = -1;
    while (T != NULL)
    {
        push(&S, T);
        T = T->left;
    }
    while (!empty(&S))
    {
        T = pop(&S);
        printf("%d", T->data);
        T = T->right;          // backtracking
        while (T != NULL)
        {
            push(&S, T);
            T = T->left;
        }
    }
}
```

Q. Construction of a Binary Tree when Inorder & preorder traversal are given.

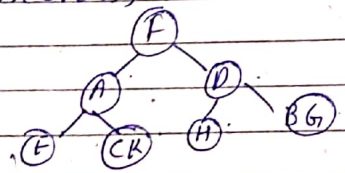
construct a Binary Tree
Inorder: EACKFHDBG
Preorder: FAEKCDHGB

Step 1: In preorder traversal root is the first node hence F is the root node of Binary Tree.

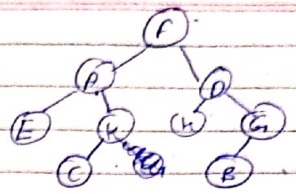
Step 2: From Inorder traversal we can find left and right subtree.



Step 3: Among Node EACK - A is a root node of subtree EACK. Among node HDBG D will root node of subtree HDBG (from preorder),



Step 4: From the preorder sequence K comes before 'C' in the subtree CK so K will be root node for subtree CK. Similarly G will root node for subtree BG.



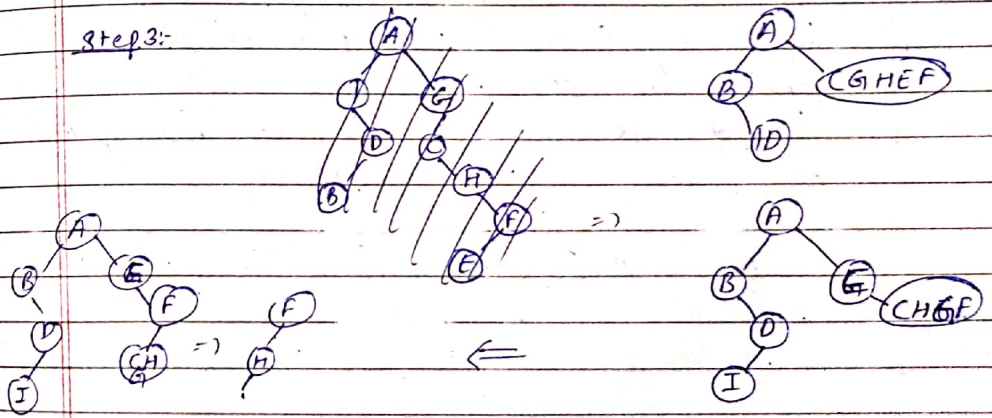
Creation of Binary Tree if Inorder & postorder are given construct the Binary Tree for the following order.

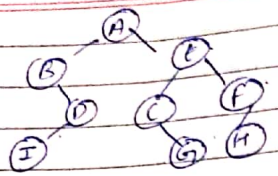
Inorder: BIDACGEHF
Postorder: IDBGCHFEA

Step 1: In post order root node = A.

Step 2: From Inorder.

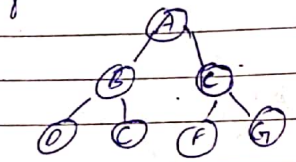
Step 3:-





Complete Binary Tree:
A Binary tree is said to be complete B T in which every level of B T is completely filled except last level. Incomplete B T each node will have two children except leaf Node.

~~except leaf Node~~
~~complete~~



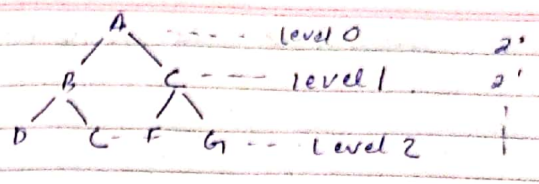
Q. Prove that a height of complete BT $h = \log_2(n+1) - 1$ where n is a total no. of nodes.

A. No. of nodes in a complete Binary Tree at every level i will be.

No. of Node $= 2^i$

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$



$2^0 + 2^1 + \cdots 2^h$

$\Rightarrow \frac{2(2^h - 1)}{2-1} \Rightarrow 2^{h+1} - 1$ $\qquad \frac{a(r^n - 1)}{a-1}$

$n+1 = 2^{h+1}$

$\log(n+1) = h+1 \log 2$

$\log_2(n+1) - 1 = h$

Basic Tree Operations:

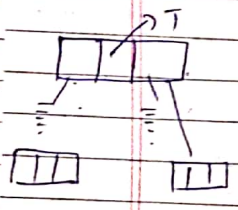
1). See f(n) for counting of Nodes in a Tree:

```
int count (Tree * T)
{
    int i;
    if (T == NULL)
        return 0;
    i = 1 + count(T->left) + count(T->right);
    return (i);
}
```

2). Recursive C f(n) for counting of leaf Node in a tree:

```
int countl ( Tree * T)
{
  int i;
if ( T == NULL)
   return 0;
if ( T→ left == NULL && T→right == NULL)
   return (1);
i = countl( T→ left) + countl ( T→right);
return (i);
```

3). C f(n) for counting No. of nodes of degree '1'.




```
int count1 ( Tree * T)
  {
   int i
   if ( T == NULL )
     return 0;
if (T→ left == NULL && T→ right == NULL)
     return 0,
if ( T→ left == NULL || T→ right == NULL)
   {
     i = 1 + count1(T→ left ) + count1(T→right);
     return (1 ).
   }
   i = count (T→ left) + count1(T→ right),
   return i;
  }
```

4) C f(n) for counting Node of degree '2'.

```
int count2 ( Tree * T )
  {
   int i ;
   if (T == NULL)
     return 0,
   if (T→ left == NULL && T→right == NULL)
     return 0;
   if (T→ left == NULL || T→right == NULL)
     {
       i = count2( T→ left) + count2(T→right);
       return (i,
     }
   i = i + (count2 (T→left ) + count2( T→right),
   return (i).
```

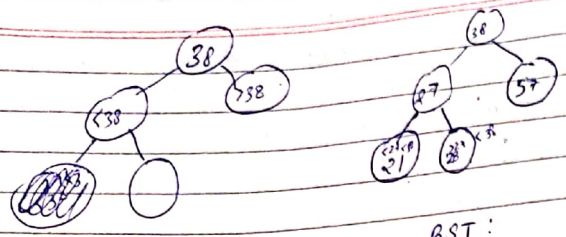
Binary Search Tree:-

Is a Binary Tree which is either empty or in which each node contains a key that satisfies the following condition.

(i) All keys are distinct.

(ii) For every node x in the tree, the values of all the keys in its left subtree are smaller than the key value in node x

(iii) For every node x in the tree, the value of all the keys in its right subtree are larger than the key value in x.



BST:

Operations on Binary Search Tree:
We can perform various operations on BST some of them are:

(i) Initialize
(ii) Find
(iii) makeEmpty
(iv) delete
(v) Insert
(vi) Create
(vii) findMin
(viii) Find Max

```
typedef struct BSTnode
{
    int key;
    struct BSTnode *left, *right;
} BSTnode;
```

(i) Initialize

```
BSTnode * initialize()
{
    return NULL;
}
```

(ii) Find

It is required to find whether a key is there or not in the tree. If the key x is found in the node T, the fn returns the address of T or NULL if there is no such node found.

```
BSTnode * Find (BSTnode *T, int x)
{
    while (T != NULL)
    {
        if (T->key == x)
            return (T);
        if (x > T->data)
            T = T->right;
        else
            T = T->left;
    }
    return (NULL);
}
```

(iii) Make Empty(): This fn deletes every node of the tree. It also releases the memory acquired by the root.

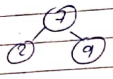
```
BSTnode * MakeEmpty (BSTnode * T)
{
    if (T != NULL).
    {
        makeempty (T->left);
        makeempty (T->right);
        free (T);
```

Q. Insert 7, 2, 9, 0, 5, 6, 8, 1 into a BST by repeated application of Insertion.

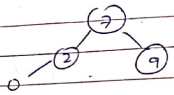
Insert: 7:

Insert 2:

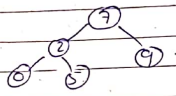
Insert 9



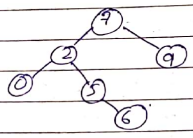
Insert 0



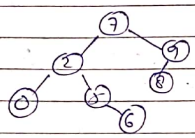
Insert 5



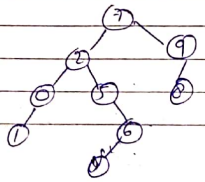
Insert 6



Insert 8



Insert 1

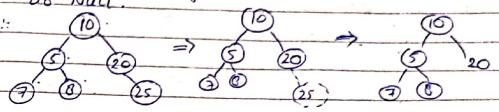


In order to delete a node

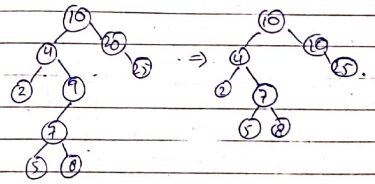
STEPS: To find out node to be deleted. The node to be deleted may be a ① Leaf Node.
② A Node with one child
③ A Node with 2 child.

① Deletion of a leaf Node: If the node is the leaf Node, it can be deleted by setting the corresponding parent pointer to NULL.

eg:



② Deletion of a Node with one child: If a node 'q' is to be deleted if it is right child of its parent node 'p'. Its only child of 'q' will become the right child of p, after deletion of q. Similarly if a node q is to be deleted if it is left child of its parents node p, its only child of q will become the left child of p after deletion.

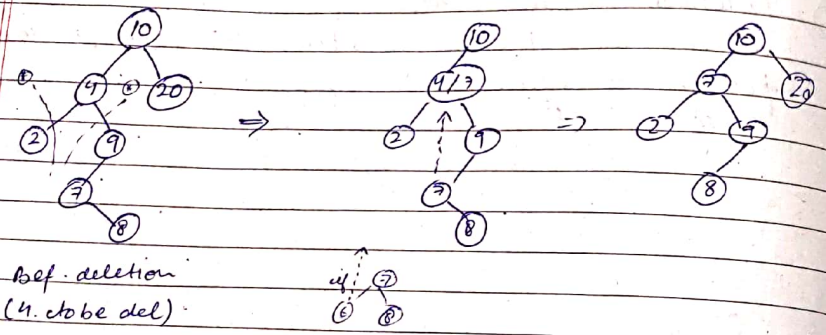


③. Deletion of a node with 2 child: The general strategy is to replace the data of this node with

① ② Either the smallest node in the right subtree.

② ② OR Largest Node from the left subtree (any one of these two).

The smallest node in right subtree will either be a leaf node or a node with degree one.




```
BSTnode *Delete (BSTnode * T, int x)
 {
  if ( T == NULL)
  {
  printf ("Underflow");
   return (T);
  }
 if (x < T→data)          // to move left
 {
 T→left = delete (T→left, x),
    return (T),
 }
 if (x > T→data)          // to move left
  {
   T→right = delete (T→right, x),
   return (T);
  }
 if (T→left == NULL && T→Right == NULL) // leaf Node.
  {
      temp = T
      free (Temp);
      return (NULL);
 if (T→left == NULL) // one child
  {
      temp = T;
      T = T→right;
      free (temp);
      return (T);
  }
 if (T→right == NULL).
  {
    temp = T;
    T = T→left;
    free (temp);
    return (T);
  }                          // node with 2 childs.
temp = find_min (T→right),
  T→data = temp→data;                  → replace with min data.
  T→right = delete (T→right, temp→data),
  return (T),
```

AVL Tree:- (Adelson, velskii & Landi's) is a ht balanced tree. These trees are BST in which the ht of 2 siblings are not permitted to differ by more than one.

| Height of left subtree − Height of right subtree | ≤ 1;

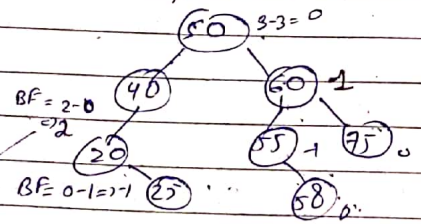
Height Balanced Tree:- ① An Empty tree is height balanced

② A Binary tree with h_l & h_r ht. height of left subtree and right subtree respectively is height balanced if $|h_l - h_r| \leq 1$

③ A Binary tree is said to be height balanced if every subtree of the given tree is ht - balanced.

Ex:-

Balance factor:- The value of $h_l - h_r$ is k/a Balance factor.



BF = 00
=0

) Not AVL Tree:-

Worst case Time for Tree = O(n)

AVL Tree:

W.C Time complexity = O(log n) (depends on ht of AVL tree)

$h = \log_2(n+1) - 1$

$n = 2^h - 1$
↓ ↑ ht of tree
max. no of node

$2^h = n+1$

$h = \log_2(n+1)$

$h = \log n$

$T(n) = O(h)$ → for all tree

$= O(\log_2(n+1))$

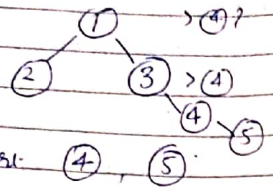
$= O(\log_2 n)$ → insertion & deletion of AVL tree

if $h = 1$
$2^h - 1$

if $h = 0$
$2^{h+1} - 1$.

Advantage:- Time complexing cannot be > O(log n).

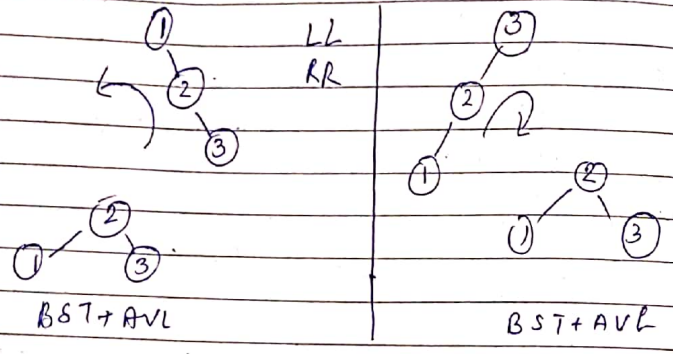
Insert in AVL Tree

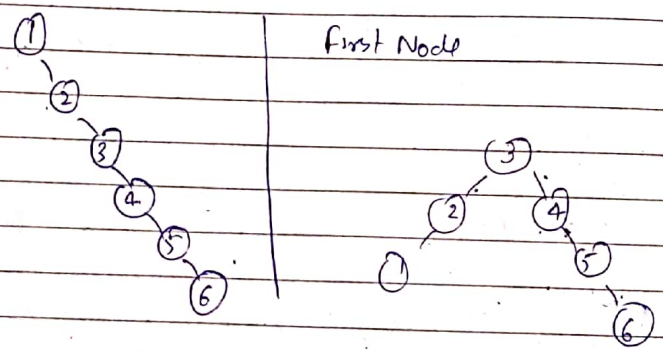


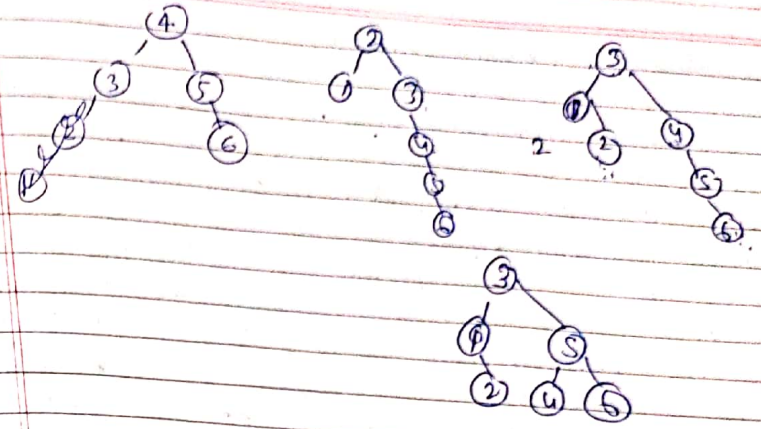
Insert: 4, 5

Not AVL Tree. 3-1=2.

Rotation:-





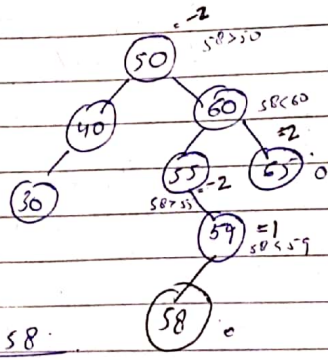


Insertion of Nodes in AVL Tree:

Insertion of a new data into an AVL Tree is carried out in two step.

STEP1: Insert the new element creating AVL Tree as a Binary Search Tree.

STEP2: Update the balance factor working upward from the point of insertion to the root node. It should be clear that after insertion the nodes on the path from point of insertion to the root may have balance factor altered. In such cases we need to perform different types of rotation to full fill properties of AVL tree.



Insert 58.

Rotation to rectify B.F from ↑ direction

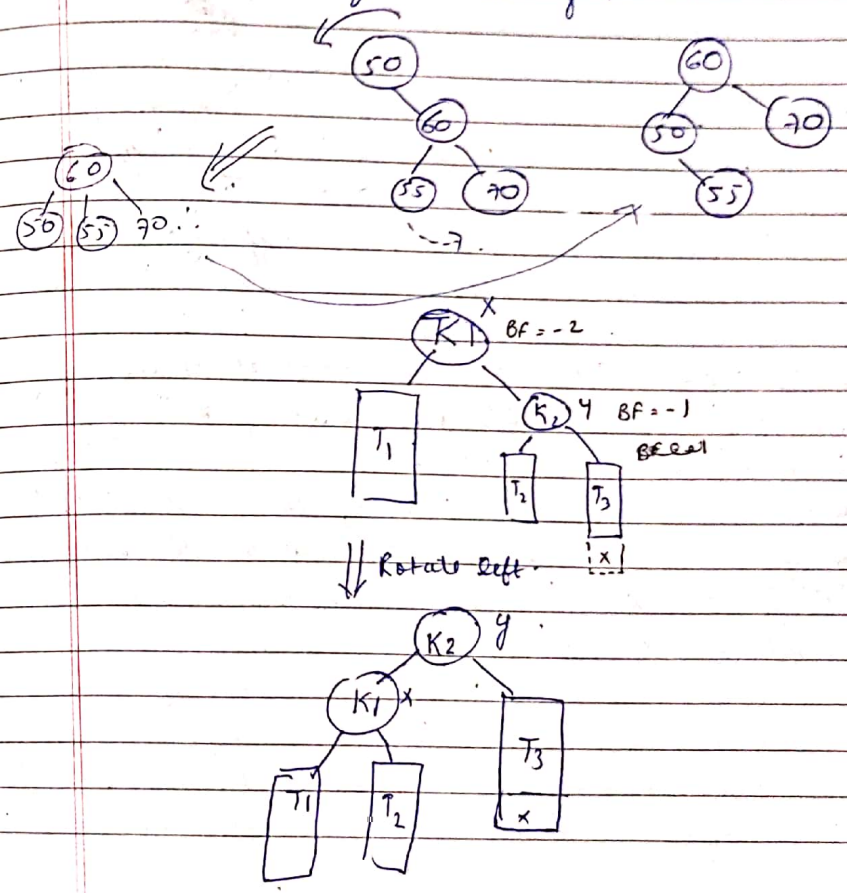
Rotation in AVL Tree:

Rotation:

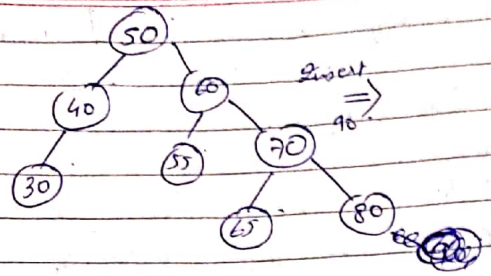
1) Rotate left: When a tree rooted at x is rotated left, the right child of x (say y) becomes the root of the tree.

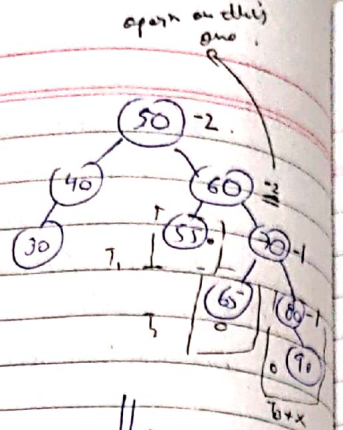
Step2: The node x will become the left child of y

Step3: Tree T2 which was the left child of y will become right child of x.

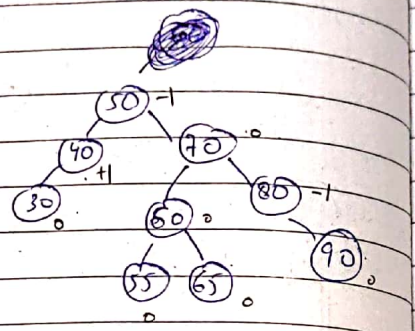


Ex



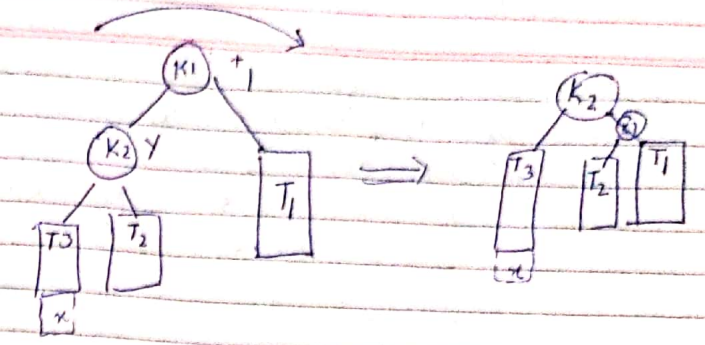


Rotate left



Rotate Right: When a tree rooted at x is rotated right. The left child of x (say y) become the root node.

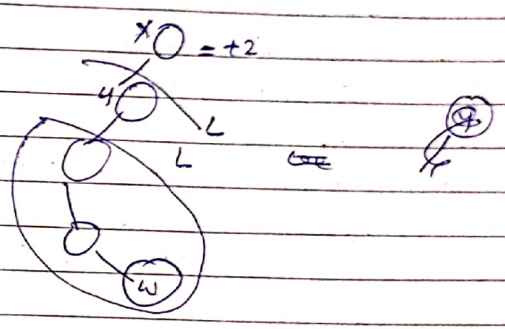
Step2 The node x will become the right child of y : Tree T2 which was the right child of y will become the left child of x.

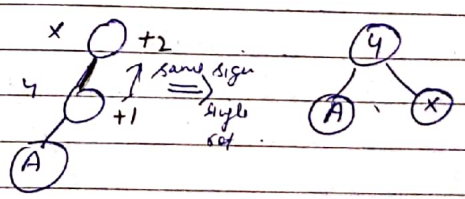


Single Rotation:

1) LL Rotation: When the insertion performed on the left subtree of the left subtree of a node x.

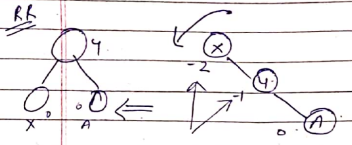
Balance Nature of the tree can be restored through single Right rotation of the node x.





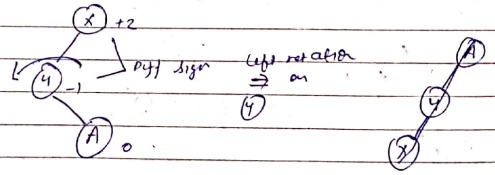
Single rotation if ↑ upwards
sign +1 & +2 same sign
∴ single rotation
if both two + = LL Rot = Right Rot
− = RR Rot = Left Rot.

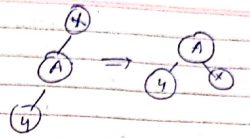
RR



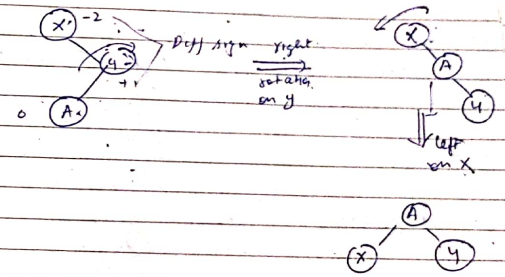
Double Rotation

1) LR Rotation. Let X be the node with BF = +2 after insertion of the new node A if new node A is inserted in the right subtree of the left subtree of X. Balance nature of the tree can be restored through double rotation.
   i) Node Y is rotated left
   ii) Node X is rotated right.





When a new node A is inserted in the left subtree of the right subtree of X. Balance nature of the tree can be restored through double rotation,
i) Node Y is rotated right
ii) Node X is rotated left.

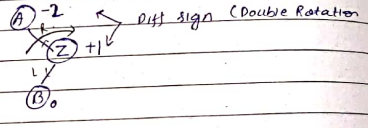


Q Draw Diagram to show different stages during the building of AVL tree for the following sequence of key A, Z, B, Y, C, X, D, V, E on each case show the balanced factor of all the nodes & name the type of rotation used for balancing.

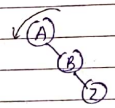
Insert A

Insert Z

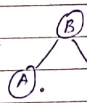
Insert B



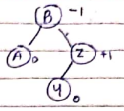
Right Rotation on Z



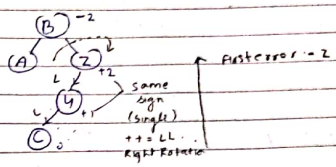
Left Rotation on A

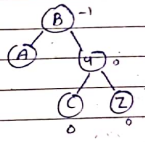


Insert Y

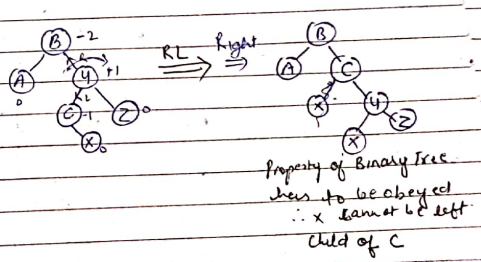


Insert C

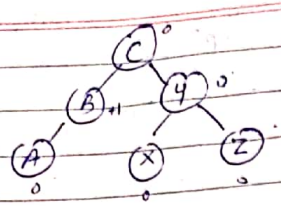




Insert X

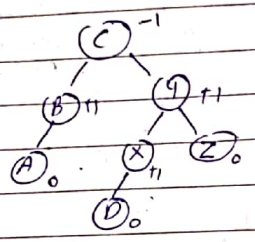


Left (B)

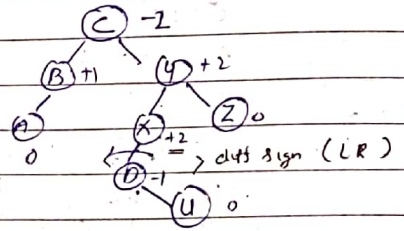


(B) -2 → RL
(Y) +1
R on child (Y)
L on parent (-2) where joint occurs.

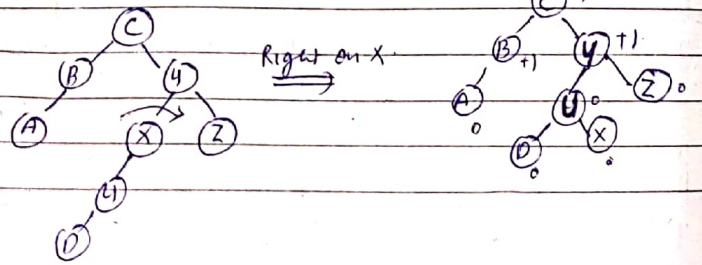
Insert D



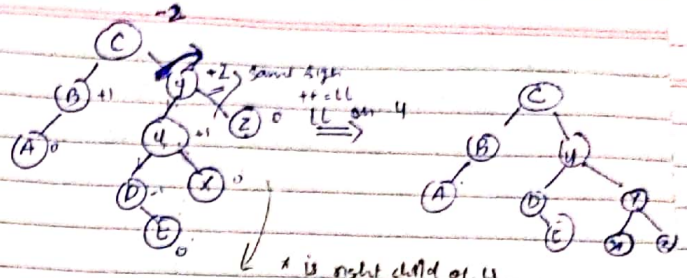
Insert U.



⇓ Left on (D) · (child of X



Insert (E)

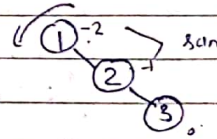


Q. Insert the following sequence of keys into an AVL Tree. Find out the type of rotations required in each case from the
1, 2, 3, 4, 8, 7, 6, 5, 11, 10, 12

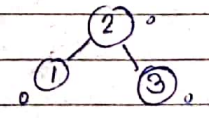
Insert 1

Insert 2

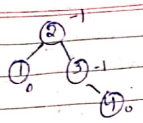
Insert 3



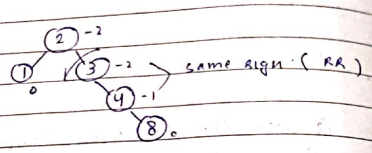
⇓ RR on (1)



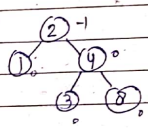
Insert 4



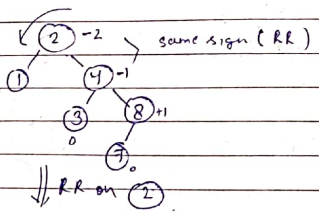
Insert 8



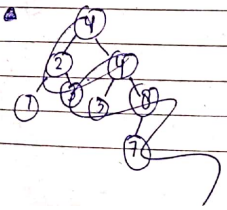
RR on 3

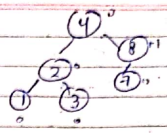


Insert 7

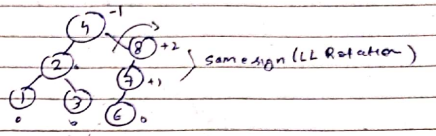


RR on 2

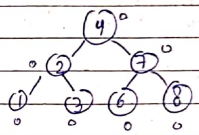




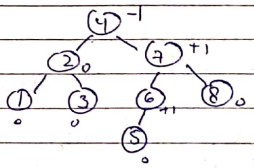
Insert 6



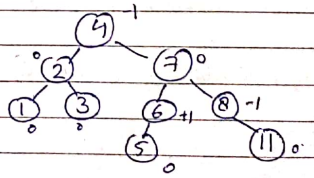
LL on 8



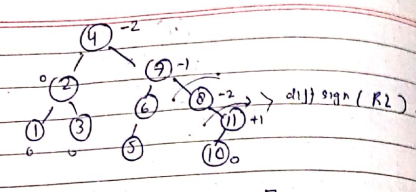
Insert 5



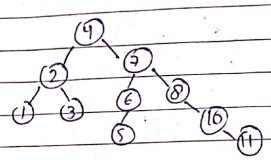
Insert 11



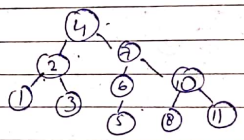
Insert 10



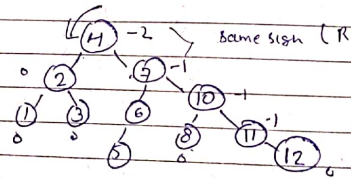
RL [R on 11]



[L on 8]

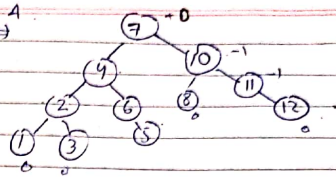


Insert 12



left child of 7 i.e (6) & (5) s

RR on A



Scanned by CamScanner

14/11/19 → Balance

B Tree: It is a self balancing tree. It is different from AVL. All the leaf Nodes are at the same level. We have 3 types self balancing tree

1) AVL.
2) B (hl - hr = 0 always)
3) Red - Black.

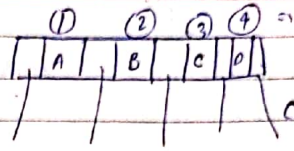
The time complexity of inserting a node in a B-Tree or deleting or searching in a B-Tree is O(h) where h is a ht of the tree

If $h = \log n$

Properties of a B-Tree:

1) ~~The order of a~~ of the order of a B tree is given m then total No of keys in a particular Node will be m-1.
2) All the node except root node must have atleast $\frac{m}{2} - 1$ keys & maximum m-1 keys.
3) If the root node is a non leaf node, then it must have atleast two children. All the key values in a node must be in ascending order.

order (m) = 5.



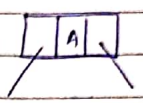
= m-1 (keys)

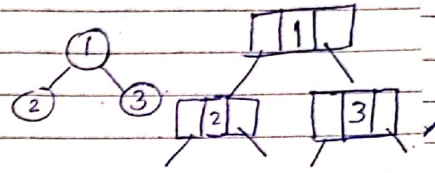
(Links) = 5 = m

order = 2 (Binary Tree) → two-order tree.

↓

2 links (left & right).





The growth and the shrink of a B-Tree are always start from the root (in AVL ⇒ from leaf Node)

D H Z K B P Q E A S W T C L N Y M

order = 5.

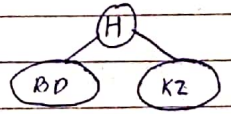
D H Z

↓

D H K Z

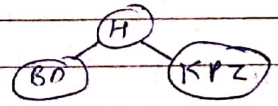
↓

B D H K Z (we grow the tree ∵ Node becomes 5 but m = 5)

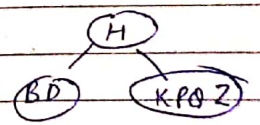


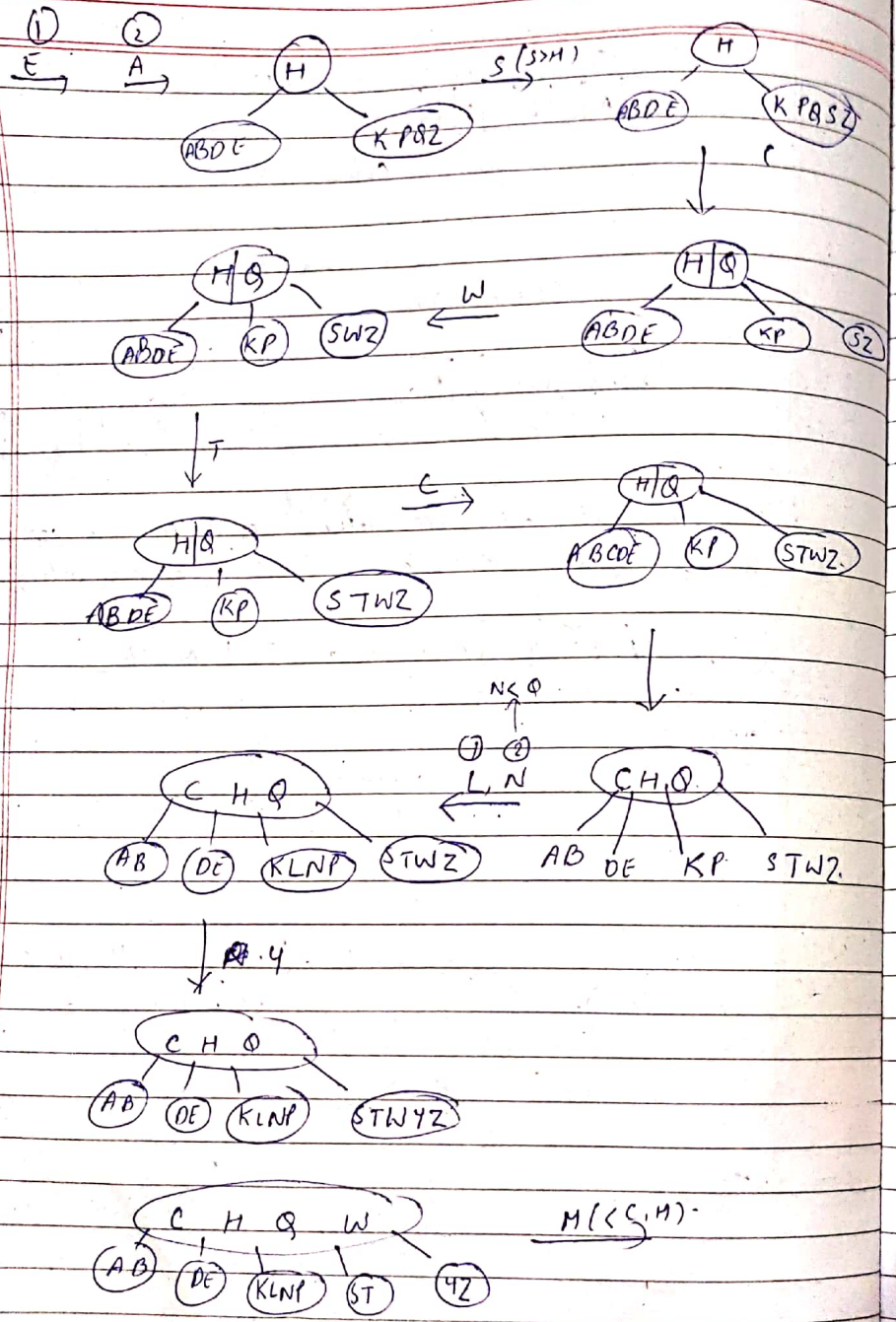
P →

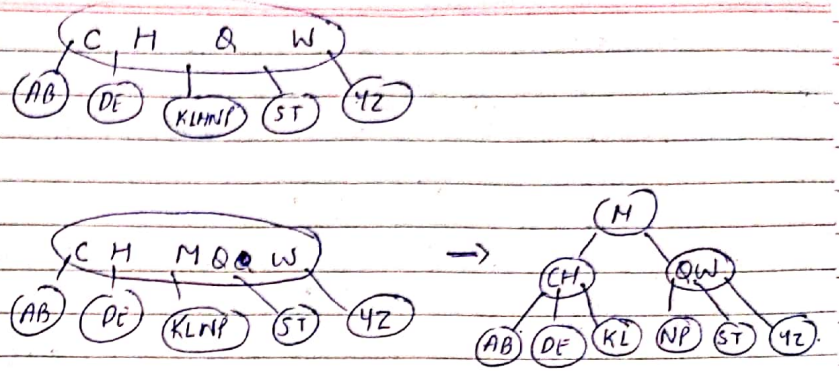
↓



Q →







Uses of Balance Trees -

1) Balance Tree is used to store the large no of values

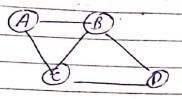
Main Memory (large Stored values of DS can be accessed using blocks) of B Tree).

Secondary Memory

18/11/19

Graphs

Terminology



Undirected

(pair (A,B) & (B,A) represents the same edge

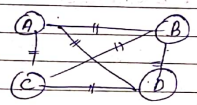
Directed

↓ diff edge.

Complete Graph

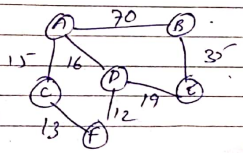
An undirected graph in which every vertex has an edge to all other vertices is called a complete graph.

A complete graph with n vertices has $\frac{n(n-1)}{2}$ edges.



$= 6$ edge $\left(\frac{4\times3}{2}\right)$

Weighted Graph: A graph in which edges are assigned some values.



Representation of Graph

A graph can be represented using

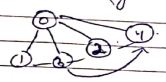
1) The adjacency matrix.
2) Link list representation

Adjacency matrix: A 2-D Matrix can be used to store a graph.

adj[i][j] = 1, indicates presence of edge b/w 2 vertices i & j

adj[i][j] = 0, indicates absence of edge b/w 2 vertices i & j.

eg:



represent the following graph using adjacency matrix.

Adj[5][5]

	0	1	2	3	4
0	0	1	1	1	1
1	1	0	0	0	0
2	1	0	0	1	1
3	1	1	1	0	1
4	1	0	1	1	0

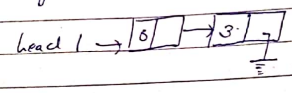
Adjacency List Representation:

A Graph can be represented using a linked list using a list of adjacent edges is maintained using a linked list. It creates a seperate linked list for each vertex V_i in the Graph $G = (V, E)$.

List Representation of previous Graph

List Adjacent to 0

List adjacent to 1



$G = (V, E)$

Adjacent Node:

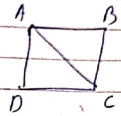
$Adj[A] = B, C, D$

$Adj[A] = B, C$

indegree of A = 0

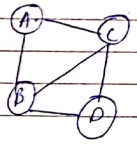
outdegree of A = 2

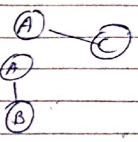
Cycle:



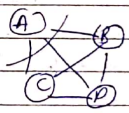
A - B - C - A

Sub-Graph





Degree of vertex



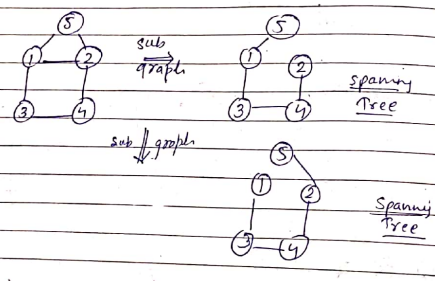
A = 3

Degree → indeg

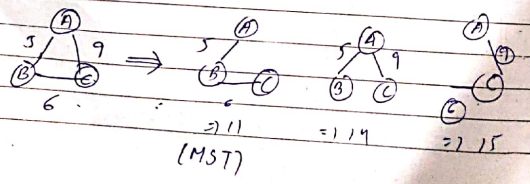
→ outdeg.

- Self edge / self loop

- Spanning Tree: The spanning tree of a graph $G(V,E)$ is a sub-graph of G having all vertices of G & no cycles in it.



- Minimum spanning Tree (MST): The cost of a graph is the sum of the cost of the edges in the weighted graph. A spanning tree of graph $G=(V,E)$ is called a Minimum cost Spanning Tree if its cost is minimum.



Kruskal's Algorithm (to find out MST)

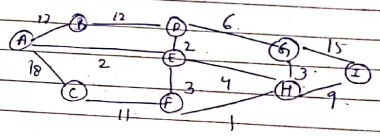
STEP 1: We arrange all edges i,j according to the ascending order of their weight.

STEP 2: Start picking the minimum weight edge & place to the spanning tree.

STEP 3: If there exist a cycle, then remove the edge from the spanning tree.

STEP 4: Repeat step 2 & 3 till all the nodes are not connected in the spanning tree.

Ex: Find minimal Spanning Tree (MST) of the given graph using Kruskal Algo.

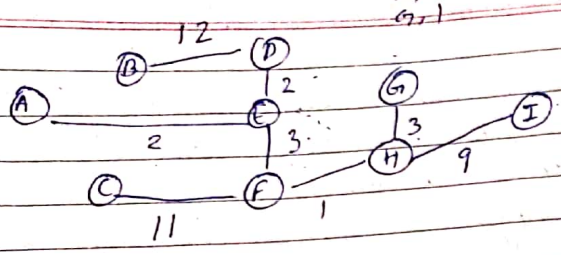


Weight	Source	Destination
1	F	H
2	D	E
2	A	E
3	G	H
3	E	F
4	E	H
6	D	G
9	H	I
11	C	F
12	B	D
15	G	I
17	A	B

18 A C

Ignore E,H
D,G
A,B
A,C
G,I

cycle



Insertion Sort:

Shortest list of elements

e.g.,

5	0	1	9	2	6	4

→ unsorted list

1	2	3	4	5	6
5	0	1	9	2	6	4

0 < 5

Sorted Array — Unsorted Array

I st Iter (pass): | 0 5 | temp 1 9 2 6 4

0 < 5
1 > 0

II nd Iter | 0 1 5 | temp 9 2 6 4

9 > 5
9 > 1
9 > 0

3rd pass | 0 1 5 9 | 2 6 4

4th Pass | 0 1 2 5 9 | 6 4

2 < 9
2 < 5

5th Pass | 0 1 2 5 6 9 | 4

2 > 1
2 > 1

6th Pass | 0 1 2 4 5 6 9 |

no. of passes = n − 1

C fn / algorithm

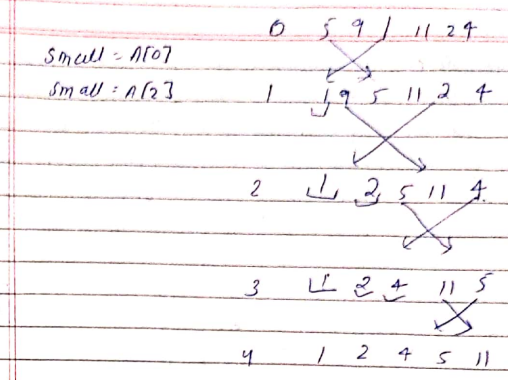
```
void Insertion (int a[], int n)
{
    int i, j, temp;
for (i=1; i<n; i++)   // no. of passes
    {
        temp = A[i];
for (j=i-1; j>=0 && a[j]>temp; j--)
        {
            a[j+1] = a[j];
        a[j+1] = temp;
    }
}
```

Selection sort:

It is a very simple sorting techniques In the ith pass, we select the element with lowest value with A[i], A[i+1]... A[n-1] and swap it with A[i]. After i passes first, i elements will be in sorted order.

Eg: Perform selection sort on given list of elements.

5 9 1 11 2 4



C fn:

```
void selection (int a[], int n)
{
    int i, j, temp, min;
for (i=0; i<n-1; i++)
    {
        min = i;
for (j=i+1; j<n; j++)
        {
            a[j] < a[min]
            min = j;
        }
    temp = a[i];
    a[i] = a[min];
    a[min] = temp;
    }
}
```

Bubble sort: is most popular technique just because it reduce time complexity after each pass by discarded sorted elements.

Eg:- Perform bubble sort on the given set of elements.

5 9 6 2 8 1

pass i=1

j=0	5 9 6 2 8 1
j=1	5 6 9 2 8 1
j=2	5 6 2 9 8 1
j=3	5 6 2 8 9 1
j=4	5 6 2 8 1 9

pass i=2

j=0	5 6 2 8 1 9
1	5 6 2 8 1 9
2	5 2 6 8 1 9
3	5 2 6 8 1 9
4	5 2 6 1 8 9

i=3

j=0	5 2 6 1 8 9
j=1	2 5 6 1 8 9
j=2	2 5 6 1 8 9
	2 5 1 6 8 9

```
Void Bubble sort (int a[], int n)
{
    int i, j, temp,
    for(i=0; i<n; i++)
    {
        for(j=0; j<n-i; j++)
        {
            if (a[j]>a[j+1])
            {
                temp= a[j],
                a[j]= a[j+1],
                a[j+1]= temp,
            }
```

DATE: / / 20

Quick Sort: is the fastest internal sorting. The basic algo for sorting an array A of 'n' elements can be described recursively

(1). if ($n <= 1$) then return

(2). pick any element V (pivot element) in A[],
Rearrange elements of the array by moving all elements $x_i > V$ right of V and all elements $x_i \leq V$ left of V. If the place of the V after rearrangement is j all elements with value less than V appear in a[0], a[1] ---- a[j-1] and all those with value greater than V appear in a[j+1], a[j+2], --- a[n-1];

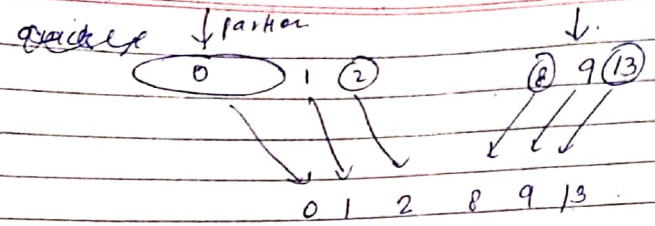
3) Apply quicksort recursively to
a[0] -- a[j-1]
and
a[j+1] --- a[n-1],

Q. Perform quick sort in the sorting of elements
5 1 2 9 0 8 13

A1. (5 1 2 9 0 8 13.) partition
↑
V.

(1 2 0) 5. (8 9 13)

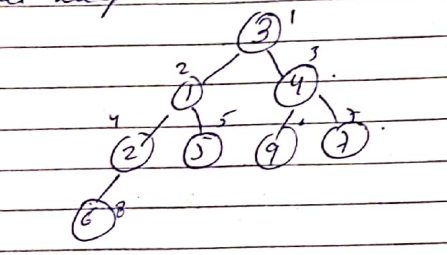
quick sort on left array — quick sort on right array.



Heap Sort: < MAX heap → all nodes are greater than its child.
MIN heap

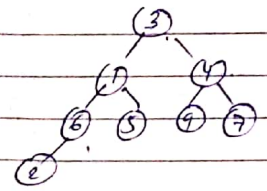
3, 1, 4, 2, 5, 9, 7, 6.

1) Create heap.

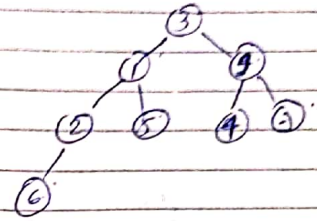


0	1	2	3	4	5	6	7	8
8	3	1	4	2	5	9	7	6

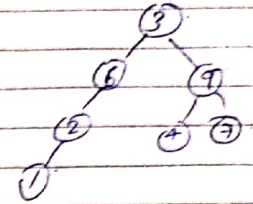
2) Step2: upadjust on node at 4321 (8/2=4)
upadjustment on 4



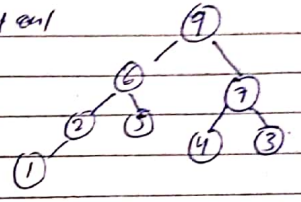
upadjustment on 3.



upadjustment 2



upadjustment on 1



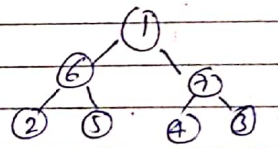
max heap

max no

0	1	2	3	4	5	6	7	8
8	9	6	7	2	5	4	3	1

min no

replace max with min

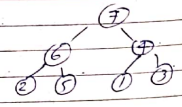


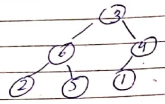
sorted

0	1	2	3	4	5	6	7	8
1	6	7	2	5	4	3	9	

now it does not have max heap
now down adjustment.

Step IV





6	3	6	4	2	5	1	7	9

Q. Merge Sort :- is based on divide and conquer approach in which the array of n-elements is split around its centre. Producing two smaller array, this process will continue till single elements of array is not formed After that sorting can be done on unique Elements of array by comparing both.

Eg: Divide the following list of elements using merge sort

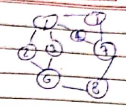
24 11 9 2 6 5 4 3

(24 11 9 2) (6 5 4 3)

((24,11) (9 2)) ((6,5) (4 3))

DFS(G, 1)

① visited(1)
DFS(G, 2)
DFS(G, 3)
DFS(G, 4)
DFS(G, 5)
} Adjacent Node



② visited(1)
visited(2)
└ adjacent nodes of 2 (G,1) (G,5) (G,6)

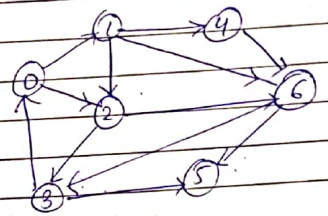
1 2 3 4 5 6 7 8
1 0 0 0 0 0 0 0

Algorithm of DFS:

```
DFS (vertex i)
{
    Vertex w,
    stack s,
    init (&s)
    push i in the stack s
    while (stack is not empty),
    {
        i = pop(&s)          // vertex vertex becomes 0 if adjacent vertex are pushed.
        if (!visited[i])
        {
            visited[i]=1,
            for each w. adjacent to i)
            if (!visited[w]
                push w into stack,
        }
    }
}
```

DATE : / 20
PAGE No

show the working of DFS algorithm on the following graph



Stack contents		Vertex visited
	0 1 2 3 4 5 6 0 0 0 0 0 0 0	push(0)
1). 0		
2) pop(0) → adj. 1, 2	1	0
1 2 ↑top		
2). pop(2) → 6, 3	1 0 1	0, 2
1 3 6 ↑top		
3) pop(6) → 3, 5	1 0 1 0 0 0 1	0, 2, 6
1 3 3 5 ↑top		
4) pop(5)	1 0 1 0 0 1 1	0, 2, 6, 5
1 3 3 ↑top		

DATE : / 20
PAGE No

pop(3) → already visited

pop(1) → 2 NV, 6, 4 NV

1 0 1 1 0 1 1

1 1 1 1 1 1 1

2 4

Hash-fl

BFS: Breadth first search:-

```
void BFS ( int v)
{
    q a queue variable.
    initialize (q);
    visited [v] = 1;
    and the vertex v to queue q
    for all vertex w adjacent v
        if ( ! visited [w]
```

```
void BFS (int v)
{
    q: a queue variable.
    initialize q;
    visited [v] = 1;
    and the
```

DATE : / 20
PAGE No

Store through hash f(n)

5, 4, 3, 15, 25

Add	data
1	
2	
3	

DIVISION - $H(k) = k \pmod m$

m → prime No

$H = 5 \bmod 3 \Rightarrow 2$

$\Rightarrow 2$

5 stored at index 2

$4 \bmod 3 \Rightarrow 1$

if index is from 0

$\therefore p \bmod m + 1$

$3 \bmod 3 = 0$

$15 \bmod 3 = 0$ → collision

Linear Probing

$h+1, k+1$

~~i) Division method~~

~~ii) mid square~~

iii) Folding

(iv) Quadratic probing.

24 11 9 2 65 4 3

(11, 24) (2, 9) (5, 6) 3 4