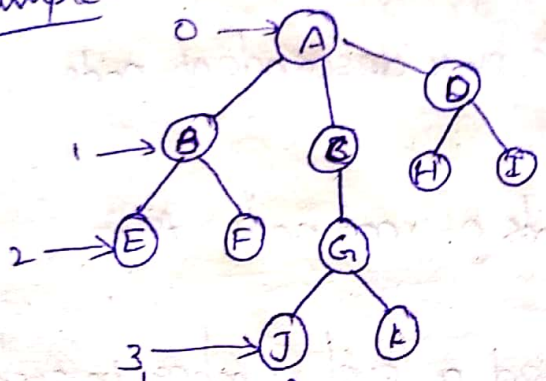
UNIT-4

TREES

Tree:
A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes are divided into number of subtrees.

Example:



Terminology of Tree:

1. **Root node:**
   It is the top most node in the tree if root is NULL then the tree is empty.

2. **Sub trees:**
   It root is not NULL then the trees are $t_1, t_2, t_3, \ldots$ are subtrees of the root node. Here B, C, D are three subtrees.

3. **Leaf node:**
   A node that has no children is called a leaf node (or) terminal node.
   Here E, F, J, K, H, I are leaf nodes.

4. **Path:**
   A sequence of consecutive edges is called a path.
   Example: Path from A to J is (A-C-G-J)

5. **Ancestor node:**
   An ancestor of a node is any predecessor node on the path from root to that node. A root node do not have ancestor node.

Example
For G the ancestors nodes are A, C

Descendant node:

It is any successor node or any path from node to a leaf node.

A leaf node do not have any descendent node

Example
C G J are descents of root node A.

Level:
Every node is assigned a level number in such a way that the root node is at level zero. And children of root node are at level '1'. All child nodes have a level number given by parent's level number + 1

The above tree has level 0, 1, 2 and 3

Degree:
Degree of a node is equal to the number of children a node has.

Example
Degree of A is 3 and B is 2. C is 1 and D is 2

Indegree:
Indegree of a node is the number of edges arriving at that node.

Example indegree of B is 1

Note: Indegree of root node is always zero

outdegree:
It is the number of edges leaving that node.

Example
Out degree of B is 2

Height of a node:
It is the number of edges from root node to that node.

Example: Height of node J is 3
Height of node E is 2

siblings:
If all the nodes are at same level and share the same parents then those nodes are called siblings

Examples: B, C, D are siblings of A
E, F are siblings of B
H, i are siblings of D
J, K are siblings of G

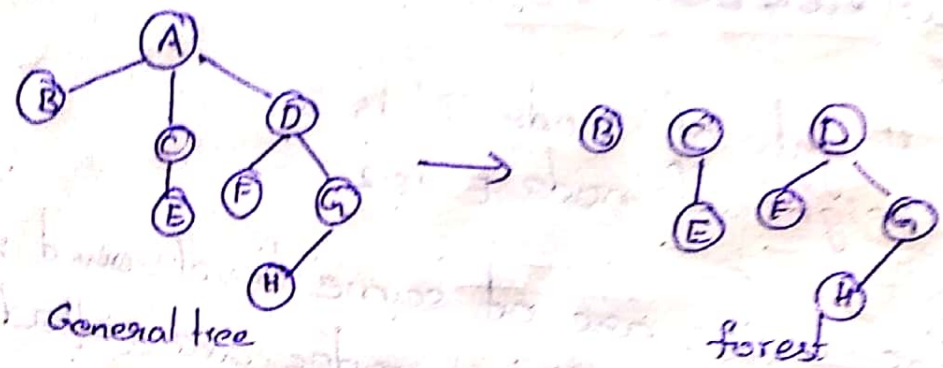
TYPES OF TREES:
There are 5 types of trees.

1. General tree
2. Forest
3. Binary tree
4. Binary search tree
5. Tournament tree.

Forest:

A forest is a disjoint union of trees it is obtained by deleting the root and edges connecting the root node to nodes at level 1.

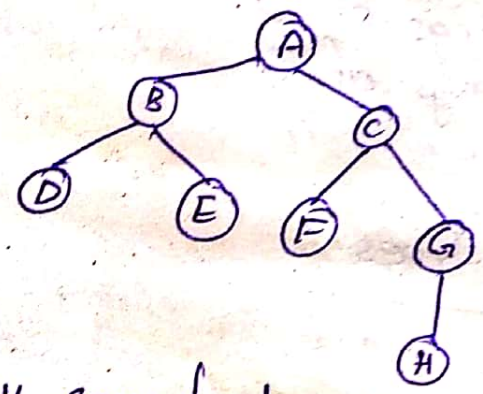
Ex:



Binary tree

A binary tree is a data structure in which each node has 0, 1, and at most 2 children

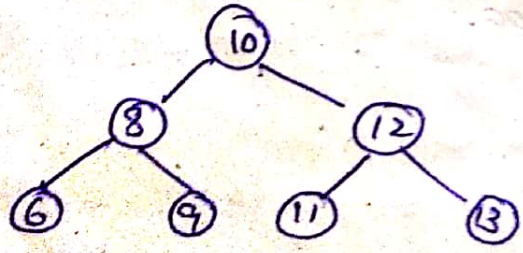
Ex:



Binary search tree:

It is a binary tree in ordered form. It is a variation of binary tree in which the nodes are arranged in order

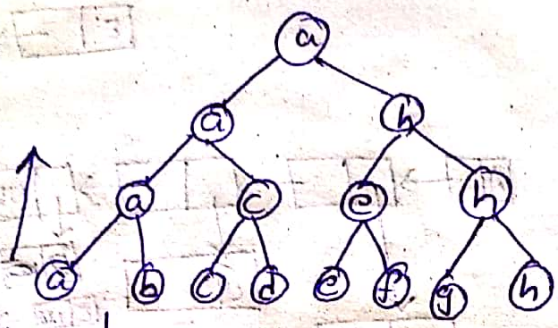
Ex:



Tournament tree:

It is also called as selection tree each external node (one without child branches) represents a player and each internal node (has atleast one child branch) represents the winner of the match played between players represented by its child nodes.

Example

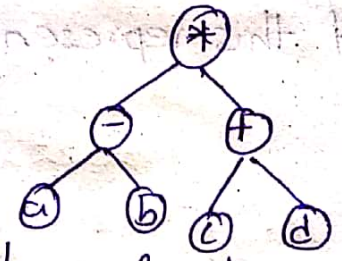


Expression tree:

It is used to store algebraic expressions.

Example

The expression (a-b) *(c+d) can be Converted to



Representation of a tree in memory:

A tree can be represented in two ways

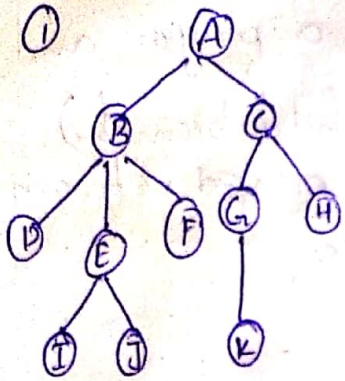
1. List Representation
2. Left child - Right sibling representation.

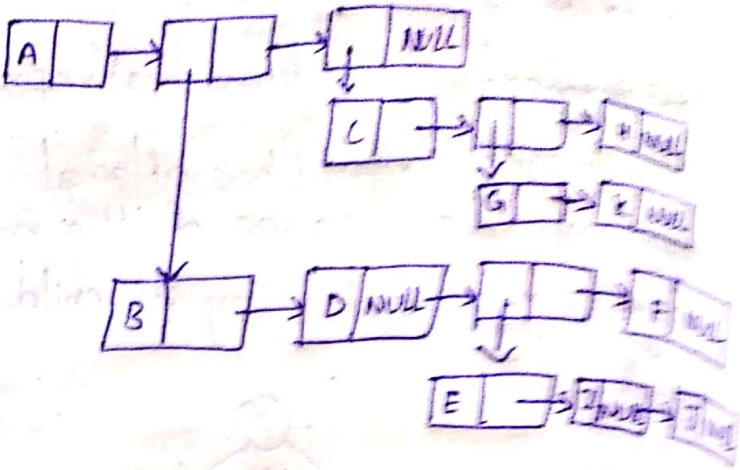
List Representation:

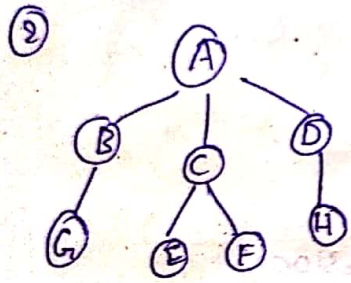
In this representation there are two types of nodes. One node for representing data called data node and another for representing only references called reference node.

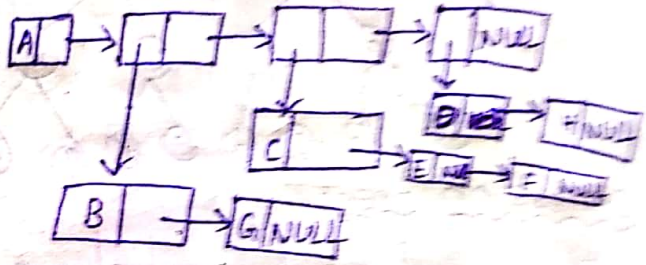
We start with a data node from the root node in a tree then it linked to an internal node to any other node directly. This process repeats for all the nodes in the given tree.

Example

① 

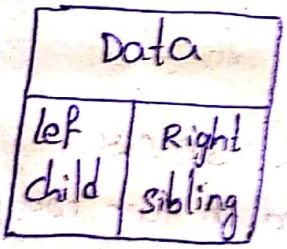


② 



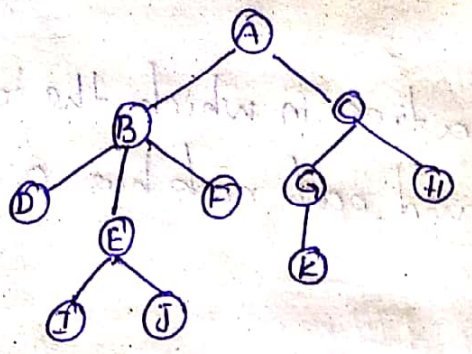
Left child - Right sibling representation:

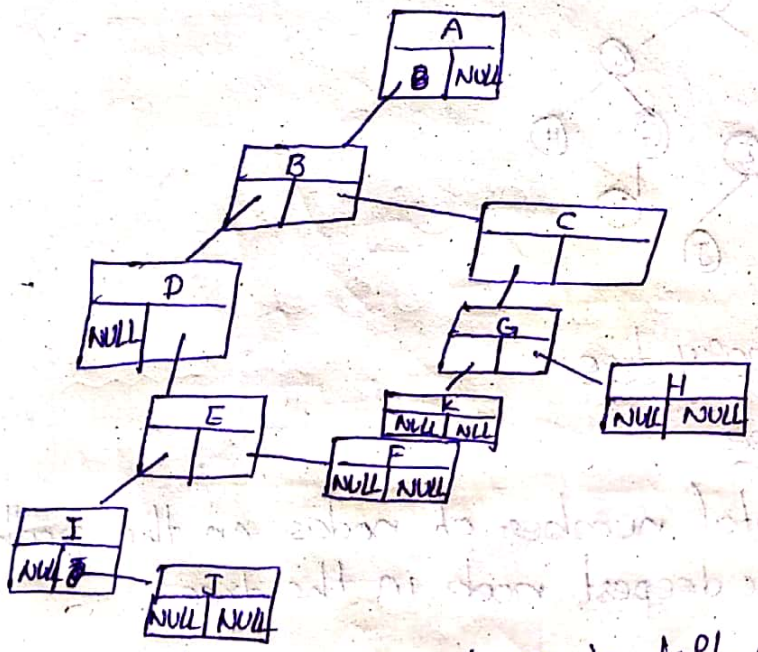
The node structure of this representation is



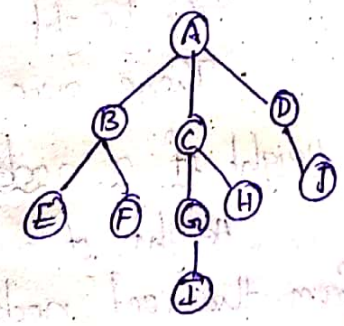
Every nodes data field contains the actual value of that node if that node has a left child then left reference field stores the address of left child node otherwise NULL. If that node has right sibling then right reference field stores the address of right sibling node otherwise NULL

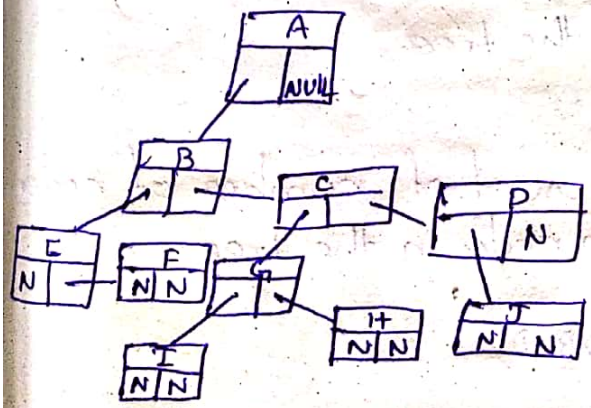
Example





Represent the given tree in left child right sibling notation.

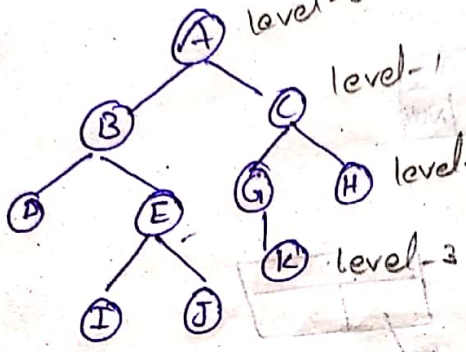




Binary tree:

Binary tree is a tree in which the top most element is called root node and each node has 0, 1 and or atmost 2 child nodes.

Example



Terminology of Binary tree:

1. Height of the tree

It is the total number of nodes on the path from the root node to the deepest node in the tree.

Example

The height of the above tree is 4 A-B-E-I

There are 3 levels in the tree.

2. Height of a node:

Height of a node is the count of number of nodes from the root node to the given node in the tree

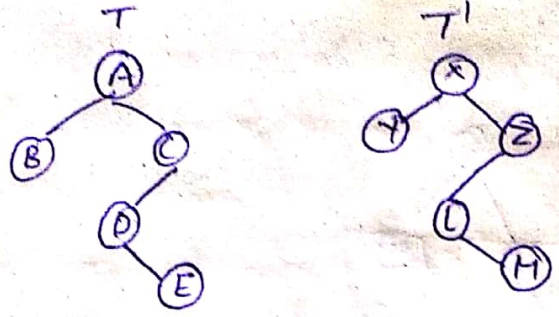
Ex: Height of node D is 3 (A,B,D)

* A binary tree of height 'h' has atleast h nodes and atmost $2^h - 1$ nodes

* similar Binary trees:

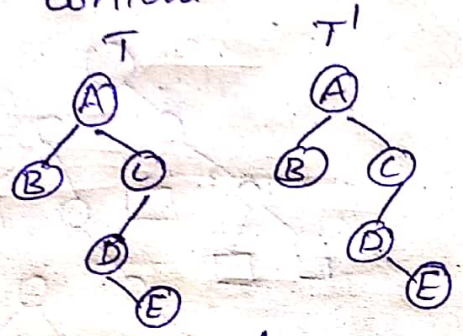
Two binary trees T and T' are said to be similar if both follow the same structure.

Example



* copies two binary trees T and T' are said to be copies if they have same structure and they can have same content

Example:



*There are two different types of binary trees.

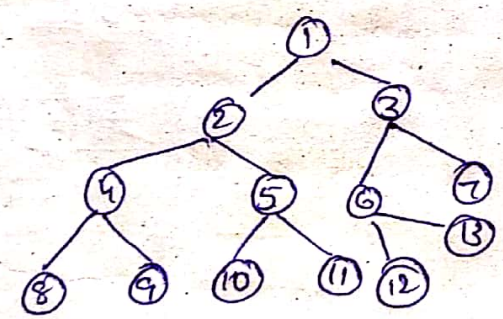
1. Complete binary tree:

It is a binary tree that satisfies two properties

a) In a complete binary tree every level except possibly the last is completely filled.

b) All nodes appear as far left as possible

Example



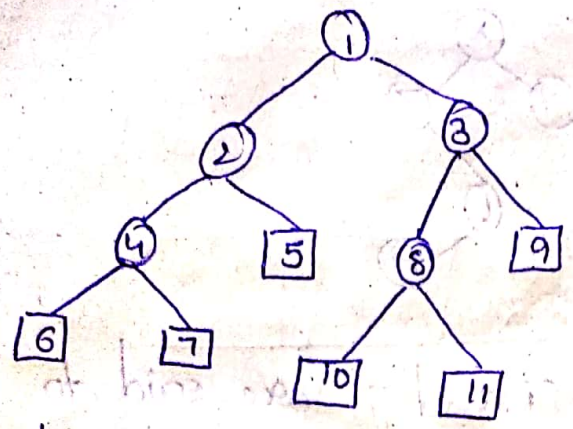
2. Extended binary tree (or) full binary tree:

In this type of tree each node can have either zero child nodes or two child nodes.

Here, the nodes are called as internal nodes (nodes having exactly two children) and external nodes (nodes having no children).

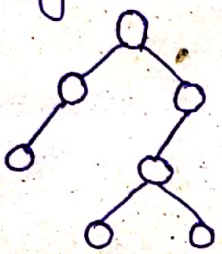
Internal nodes are represented by circles and external nodes are represented by squares.

Ex:

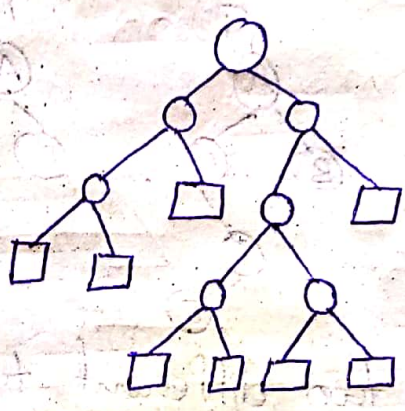


Converting a binary tree into extended binary tree.

Binary Tree

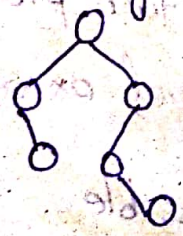


Extended binary tree.

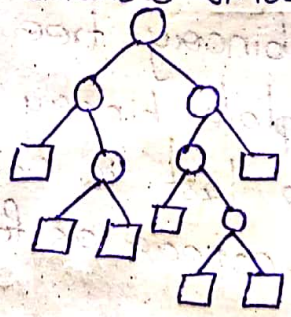


Convert the following binary tree to extended binary tree

Binary tree



extended binary tree



Representation of binary tree in memory:

A binary tree is represented in two ways

1. Array
2. Linked list

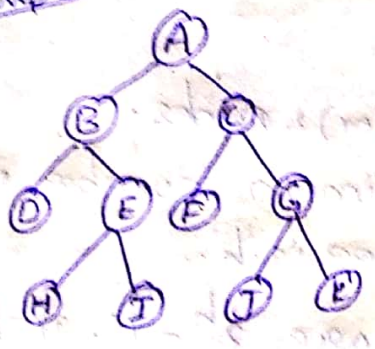
Sequential representation of binary trees.

It is done by using one dimensional arrays.

The rules for representing binary tree in array is

1. The root of the tree will be stored in the first location of the array Tree[1]
2. If the node is left child then the node will be stored at index $2*k$ where k is the index of the parent node.
3. If the node is a right child then the node will be stored at index $(2*k)+1$ where k is the location of the parent node.

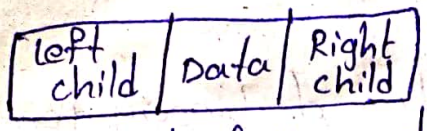
Example



Tree[0]	11
1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	
9	
10	H
11	I
12	
13	
14	J
15	K

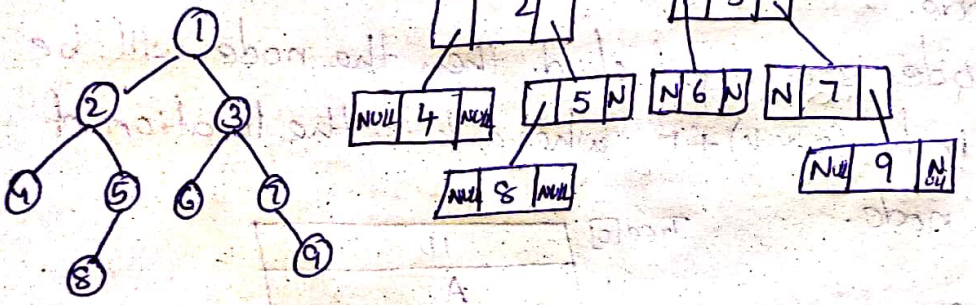
Representation of binary tree using linked list:

In linked list representation each node contains 3 fields as



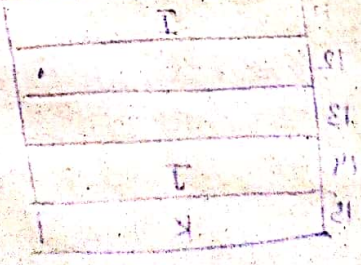
The middle part of the node is used to store data and the first part points the left child and the second part points the right child. Binary tree uses double linked list representation for every node. If there is no child node (left or) right) then it is represent as "NULL"

Example



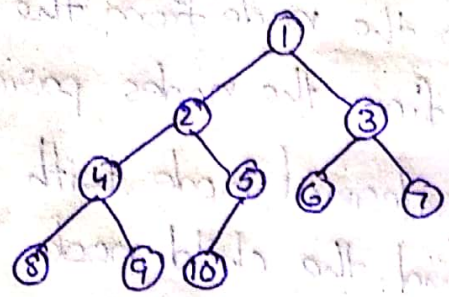
Properties of binary tree:

1. Every node contains 0 (or) 1 (or) 2 nodes.
2. If Every binary tree contains minimum number of nodes of h then the height of the tree is 'h'.
3. maximum number of nodes are $2^h - 1$. If h starts from '1'



Construct a binary tree for the given number of nodes

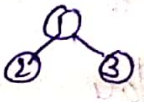
1 2 3 4 5 6 7 8 9 10



Procedure

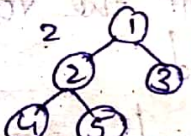
step 1 Insert 1
1 becomes the root

step 2 Insert 2
2 becomes the left child for 1

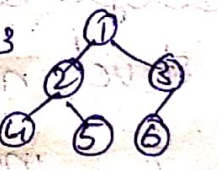


step 3 Insert 3
3 becomes the right child for 1

step 4 Insert 4
4 becomes the left child for 2

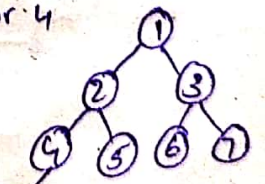


step 5 Insert 5
5 becomes the right child for 2

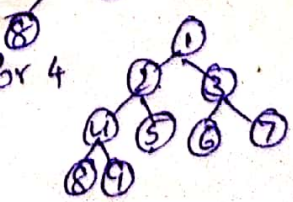


step 6 Insert 6
6 becomes the left child for 3

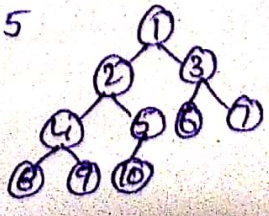
step 7 Insert 7
7 becomes the right child for 3



step 8 Insert 8
8 becomes the left child for 4



Step 9 Insert 9
9 becomes the right child for 4



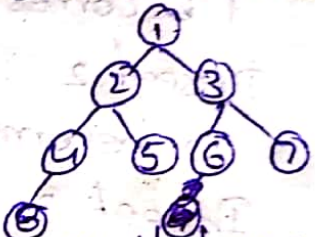
step 10 Insert 10
10 become the left child for 5

Deletion in the Binary tree:

1. If the node to be deleted is a leaf node then we can directly delete the node from the binary tree by disconecting the line from the nodes parent.

2. If the node is an internal node with one child then delete the node and the child node becomes the parent node.

3. we cannot delete a node with 2 child nodes in a binary tree Ex: Delete 8
It is directly deleted



Deleting 3
It is not possible because it has 2 children

Display

To display the nodes in a given binary tree then printf the root node then left node then right node (or) child.

Creation of a node in a binary tree:

```
struct node
{   int data;
    struct node *lchild, *rchild;
};
```

Tree traversals in a binary tree:

Traversing is the process of visiting each node in the tree exactly once in a systematic way. There are three different traversal techniques.

1. Pre order traversal
2. In order traversal
3. post order traversal

Pre order traversal:

To traverse a non-empty binary tree the following operations are performed recursively at each node.

1. visit the root node
2. Traverse left subtree
3. Traverse right subtree.

Algorithm for recursive preorder traversal:

step 1 : Repeat steps 2 to 4 until TREE becomes null
step 2 : print TREE → Data
step 3 : preorder (TREE → left)
step 4 : preorder (TREE → Right)
step 5 : stop.

In order traversal:

The procedure for in order traversal is

1. Traverse the left subtree.
2. visit the root node
3. Traverse the right subtree.

Algorithm for inorder transeraval:

step1: Repeat steps 2 to 4 until TREE becomes NULL
step2: inorder (TREE → left)
step3: print TREE → data
step4 : inorder (TREE → Right)

step 5 : stop.

Post order traversal

Procedure for post traversal is

1. Traverse the left subtree
2. Traverse the right subtree
3. visit the root.

Algorithm for post order traversal:

step 1 : Repeate 2 to 4 steps until TREE become NULL

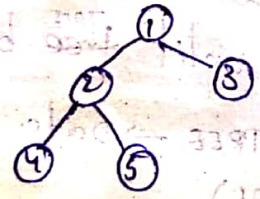
step 2 : Postorder (TREE → left)

step 3 : Postorder (TREE → Right)

step 4 : Print TREE → data

step 5 : Stop.

* For a given binary tree find preorder, inorder, postorder traversals.

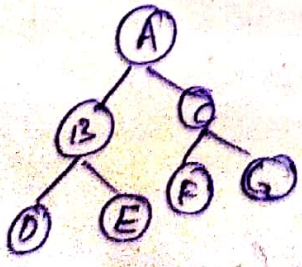


Pre order traversal : 1, 2, 4, 5, 3

inorder traversal : 4, 2, 5, 1, 3

Post order traversal : 4, 5, 2, 3, 1

* Consider the given binary tree and write the traversals

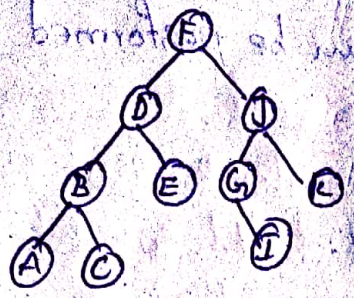


Pre traversal : A, B, D, E, C, F, G

In traversal : D, B, E, A, F, C, G

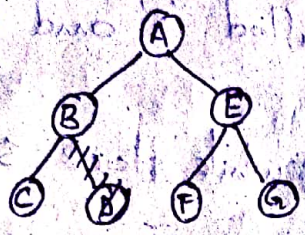
Post traversal : D, E, B, F, G, C, A

*



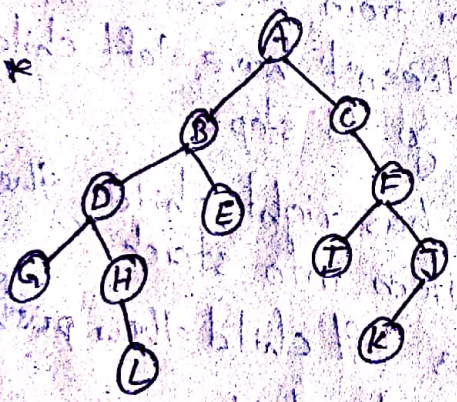
Pre order traverse - F, D, B, A, C, E, J, G, I, K
in order - A, B, C, D, E, F, G, I, J, K
post order - A, C, B, E, D, I, G, K, J, F

*



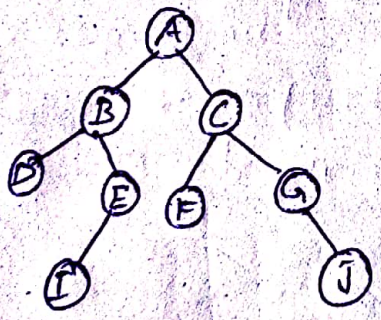
preorder - A, B, C, D, E, F, G
in order - C, B, D, A, F, E, G
post order - C, D, B, F, G, E, A

*

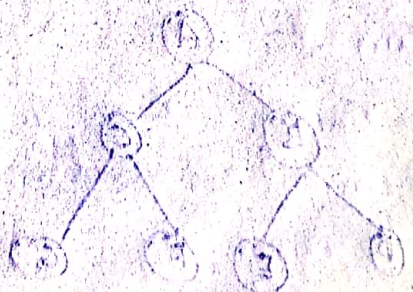


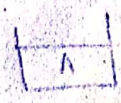
Pre-order - A, B, D, G, H, L, E, C, F, I, J, K
in order - G, D, H, L, B, E, A, C, I, F, K, J
post order - G, L, H, D, E, B, I, K, J, F, C, A

*



Pre-order - A, B, D, E, I, C, F, G, J
in-order - D, B, I, E, A, F, C, G, J
post-order - D, I, E, B, F, G, C, A





The non-recusive traversals can be performed using loops and a stack.

Algorithm for pre-order traversal :

step 1 : creat an empty stack called 'S' and push root node on to the stopck.

step 2 : Do the following steps while stack S is not empty.

step 2.1 : pop the top most item from stack and print it

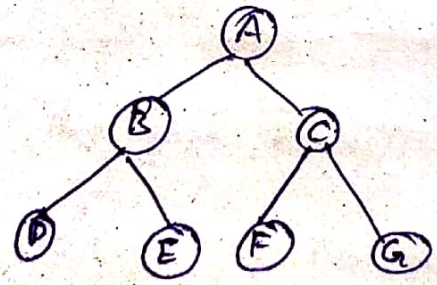
step 2.2 : If the poped item doesn't have left child and right child then go to step 2.

step 2.3 : If the poped item has right child then push right child of poped item into stack.

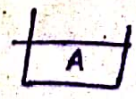
step 2.4 : If the poped item has left child then push left child of poped item into stack.

step 3 : stop.

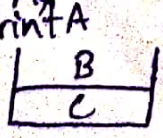
Example



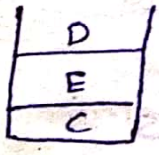
step 1 :



step 2 : Print A



step 3 : print A, B



step4 pop and print D

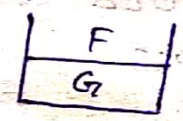
D has no child nodes A, B, D

step5 pop and print E

E has no child nodes

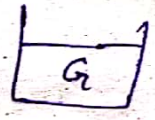
A, B, D, E

step 6 Pop and print C



A, B, D, E, C

step 7 pop and print F



F has no child nodes

A, B, D, E, C, F

step 8 pop and print G

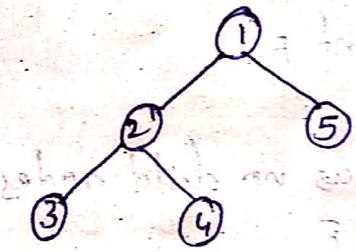
now stack is empty

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

Algorithm for inorder traversal:

step 1 : Create an empty stack 'S'
step 2 : Initialize current node as root
step 3 : push the current node to stack and set
current = current → left until current is NULL
step 4 : If the current is NULL and stack is not
empty then
step 4.1 : pop the top item from the stack.
step 4.2 : print the poped item and
set current = poped item → right
step 4.3 : Go to step 3
step 5 : If the current is NULL and stack is empty
then stop the process.

Example



Step -1

i, Current = root = 1

ii, Current = Current → left

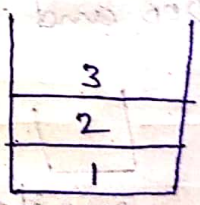
Current = 2

iii, Current = Current → left

Current = 3

iv, Current = Current → left

Current = NULL



if (current == NULL)

pop 3 and print 3

3, Set Current = poped item → right

Current = NULL

if current = NULL

pop the top most item 2

and print 2

set current = Current → right

Current = 4

3, 2

if Current ! = NULL

push current to stack and

set current = Current → left

Current = NULL

pop the top item and

set current = Current → right

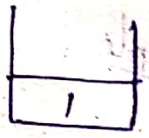
Current = NULL

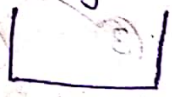
3, 2, 4

pop the top item and print it

set current = current → right

Current = 5



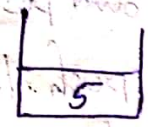


3, 2, 4, 1

if Current ! = NULL

push current to stack and set ~~curret~~

Current = Current → left = NULL



if current = = NULL

pop the top item and print it

3, 2, 4, 1, 5

set current = current → right

current = NULL

if current = = NULL and stack is empty

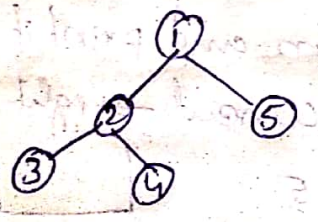
then stop the process.

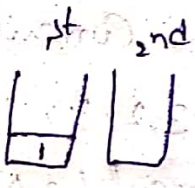
∴ The in-order traversal is 3, 2, 4, 1, 5

Algorithm for post order traversal:

step 1: push root node first stack.
step 2: Repeat the following steps while first stack is not empty.
step 2.1: pop a node from first stack and push it into second stack.
step 2.2: push left and Right children of the poped node to first stack.
step 3: print contents of second stack
step 4: stop.

Example

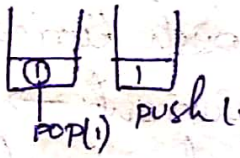




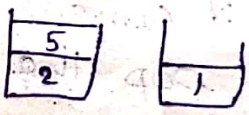
step 1: push root to first stack

step 2: pop top item from 1st stack and push to 2nd

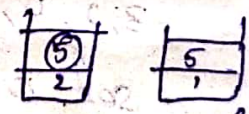
push left and right child of the poped item



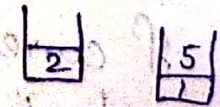
pop top item from 1st and push to 2nd



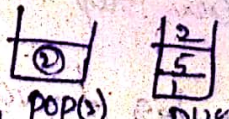
push left and right child of 5

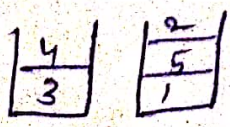


pop top item from 1st and push it to 2nd stack

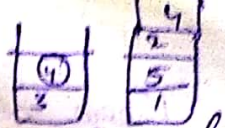


push left and right child of 2

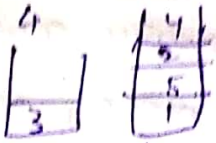




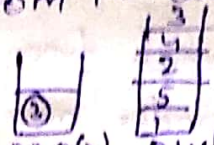
pop top item from 1st and push it to 2nd



push the left & right of 4

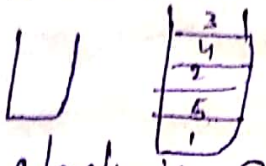


pop top item from 1st and push it to 2nd



push the left and right child of 3

There is no child nodes for three



Now, 1st stack is empty ×

step3: print the elements in the stack

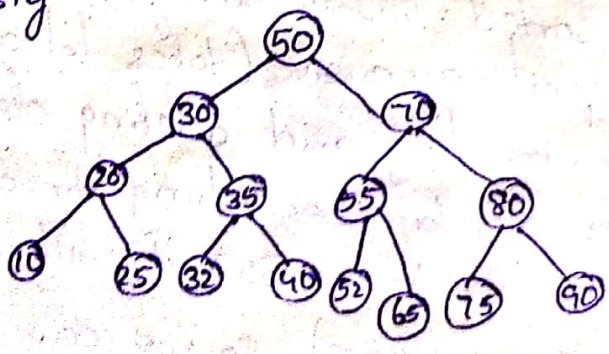
3, 4, 2, 5, 1

Binary search tree:

It is an ordered tree (or) one variant of binary tree in which nodes are arranged in order.

The left subtree of a node N contains values that are less than N's value and right sub tree of a node N contains value that are greater than N's value. Both the left subtree and right subtree have to satisfy the above two conditions.

Example



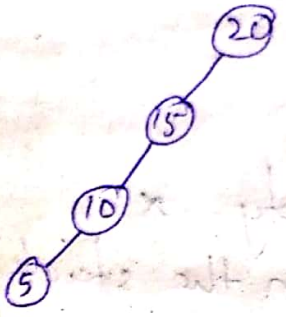
There are two types of binary search trees

1. Left skewed binary search tree
2. Right skewed binary search tree

Left skewed Binary search tree:

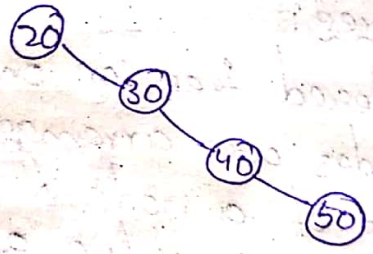
In this tree all the nodes are present in the left of each node.

Example



Right skewed Binary search tree:

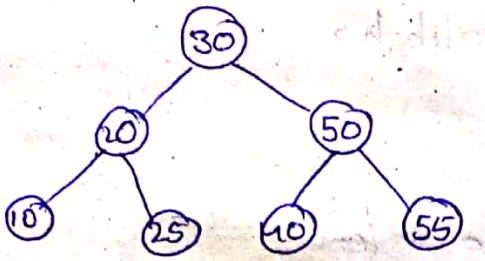
Each node contains right childs until ~~tof~~ leaf n



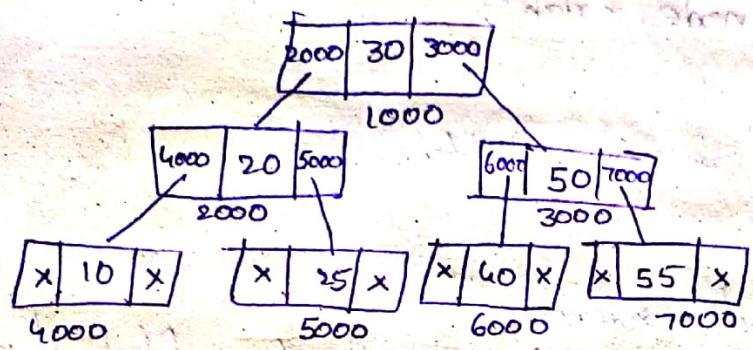
A binary search tree is considered to be an efficient data structure when compared to arrays and linked list. Because, even though searching can be done easily in sorted linear array insertions and deletions are quite expensive (takes lot of time)

In contrast inserting and deleting elements in a linked list is easier but searching for an element is O(n) time. However in a BST it will perform all the operations in $O(\log_2 n)$. So we can say that BST has more advantages than linked list

representation of a binary search tree in memory:



(x represents NULL)



operations on binary search tree:

1. Insertion
2. Traversals
3. search
4. Deletion

Insertion:

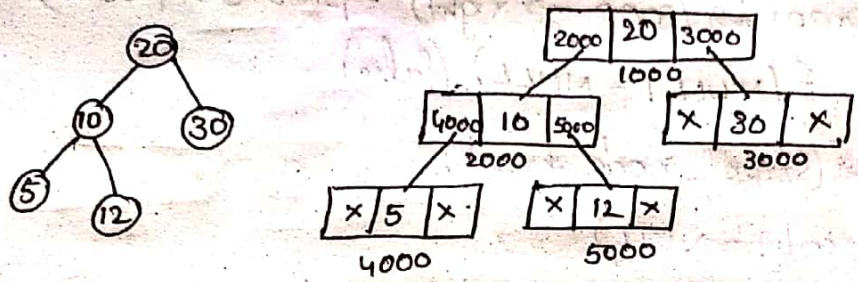
For inserting a node

1. Create an empty node called temp
2. Accept the data of the node
3. If the tree is empty then assign temp to root node. (∵ root = temp)
4. if temp → data is less than root then insert temp in left sub tree.
3. if temp → data is more than root then insert temp in right subtree.

Program

```
#include<stdio.h>
#include<stdlib.h>
struct node
{  int data;
   struct node *left;
   struct node *right;
};
struct node *root = NULL;
void insert()
{  struct node *temp;
   temp = (struct node *) malloc(sizeof(struct node));
   printf("enter data");
   scanf("%d",&temp->data);
    temp->left = NULL;
    temp->right = NULL;
    if(root == NULL)
    {  root = temp;
    }
    else
    {  struct node *Curr, *parent;
       Curr = root;
       while(Curr != NULL)
       {  parent = Curr;
          if(temp->data < Curr->data)
             Curr = Curr->left;
          else
             Curr = Curr->right;
       }
       if(temp->data < parent->data)
          Curr->left = temp;
       else
          Curr->right = temp;
    }
}
```

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



Preorder(1000 root)
if(1000 != NULL)
print(1000 -> data) = print 20
20
preorder(1000 -> left) = preorder(2000)
if(2000 != NULL)
print(2000 -> data) = print 10
20, 10
preorder(2000 -> left) = preorder(4000)
if(4000 != NULL)
~~preorder~~ print(4000 -> data) = print 5
20, 10, 5
preorder(4000 -> left) = preorder(NULL)
if(NULL != NULL) (False)
preorder(4000 -> right) = preorder(NULL)
if(NULL != NULL) (False)
preorder(2000 -> right) = preorder(5000)
if(5000 != NULL)
print 5000 -> data = print 12
20, 10, 5, 12
preorder(5000 -> left) = preorder(NULL)
if(NULL != NULL) (F)
preorder(5000 -> right) = preorder(NULL)
if(NULL != NULL) (F)

preorder(1000->right) = preorder(3000)

if(3000 != NULL)

print(3000->data) = print 30

20, 10, 5, 12, 36

preorder(3000->left) = preorder(NULL)

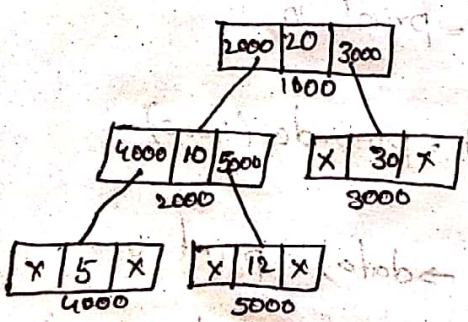
if(NULL != NULL) (f)

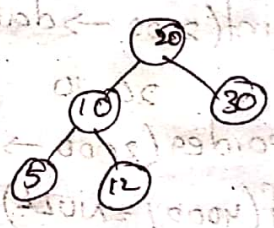
preorder(3000->right) = preorder(NULL)

if(NULL != NULL) (false)

```
Void inorder(struct node *root)
{   if(root != NULL)
    {   inorder(root->left);
        printf("%d", root->data);
        inorder(root->right);
    }
}
```

5, 10, 12, 20, 30





inorder(1000)

inorder if(1000 != NULL)

inorder(1000->left) = inorder(2000)

if(2000 != NULL)

inorder(2000->left) = inorder(4000)

if(4000 != NULL)

inorder(4000->left) = inorder(NULL)

if(NULL != NULL) (false)

print(4000->data) = print 5

inorder(4000->right) = inorder(NULL)

if(NULL != NULL) (false)

print (3000 → data) = print 10

5, 10

inorder (2000 → right) = inorder (5000)

if (5000 ! = NULL)

inorder (5000 → left) = inorder (NULL)

if (NULL ! = NULL) false

print (5000 → data) = 12

5, 10, 12

inorder (5000 → right) = inorder (NULL)

if (NULL ! = NULL) false

print (1000 → data) = print 20

5, 10, 12, 20

inorder (1000 → right) = inorder (3000)

if (3000 ! = NULL)

print inorder (3000 → left) = inorder (NULL)

if (NULL ! = NULL)

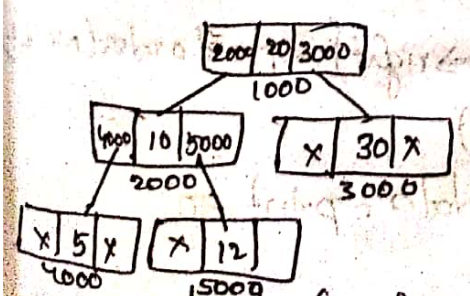
Print 3000 → data = print 30

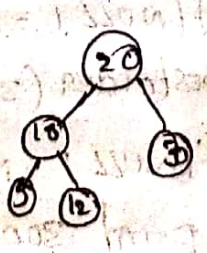
5, 10, 12, 20, 30

inorder (3000 → right) = inorder (NULL)

if (NULL ! = NULL) false

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





5, 12, 10, 30, 20

postorder (1000)

if (1000 ! = NULL)

postorder (1000 → left) = postorder (2000)

if (2000 ! = NULL)

postorder(2000 → left) = postorder(4000)

if(4000 != NULL)

postorder(4000 → left) = postorder(NULL)

if(NULL != NULL) (false)

postorder(4000 → right) = postorder(NULL)

if(NULL != NULL) (false)

print(4000 → data) = print 5

5,

postorder(2000 → right) = postorder(5000)

if(5000 != NULL)

post order(5000 → left) = postorder(NULL)

if(NULL != NULL) (false).

post order(5000 → right) = postorder(NULL)

if(NULL != NULL) (false)

print 5000 → data = print 12

5, 12

print 2000 → data = print 10

5, 12, 10

postorder(1000 → right) = postorder(3000)

if(3000 != NULL)

postorder(3000 → left) = postorder(NULL)

if(NULL != NULL)

postorder(3000 → right) = postorder(NULL)

if(NULL != NULL)

print 3000 → data = print 30

5, 12, 10, 30

print 1000 → data = print 20

5, 12, 10, 30, 20

Search operation:

1. Accept the key value
2. if the key element is equal to root then return that root
3. if the tree is empty then print NULL
4. if key is less than root then search in the left subtree continuously until we find the key
5. if the key is greater than root then search in the right subtree continuously until we find the key.

```
struct node * search(struct node *root, int key)
{   if(key == root || root == NULL)
        return root;
    else if(key < root -> data)
    {   return search(root -> left, key);
    }
    else if(key > root -> data)
        return search(root -> right, key);
    else
        printf
}
```

Deletion of a node in Binary search tree

To delete a node we must consider to follow 3 cases

Case 1: if the node is a leaf node then we can directly remove it from the tree.

Case 2: if the node contains only one child
- i) if the child is left child then replace the childs data with the roots data and remove the child
- ii) if the child is right child then replace the right child's data with the root data and remove the right child.

Case 3: if the node contains two children then we may consider any one option in the following two options.
- Find minimum node in the right subtree and replace the root data with the minimum node data and delete the node.

(ii) Find maximum node in the left subtree and replace the root data with the maximum node data and remove the maximum node

```
struct node * minimum (struct node *root)
{
    struct node * curr = root;
    while(curr & curr->left != NULL)
    {
      curr = curr->left;
    }
    return curr;
}
struct node *delete(struct node *root, int key)
{  if(root == NULL)
     return root;
   if(key < root->data)
     root->left = delete(root->left, key)
   else if (key > root->data)
     root->right = delete(root->right, key);
   else
   {  if(root->right == NULL)
      {  struct node *temp = root->left;
         free(root);
         return temp;
      }
      else if(root->left == NULL)
      { struct node * temp = root->right;
        free(root);
        return temp;
      }
      else
      {   struct node *temp = minimum(root->right);
          root->data = temp->data;
          root->right = delete(root->right, temp->data);
      }
```

```
    return root;
}
```

UNIT-5

AVL TREES

AVL Tree: AVL tree is a self balancing binary search tree invented by three scientists known as GM Adelson, Velski, EM Landis. The tree is named AVL in honour of its inventers.

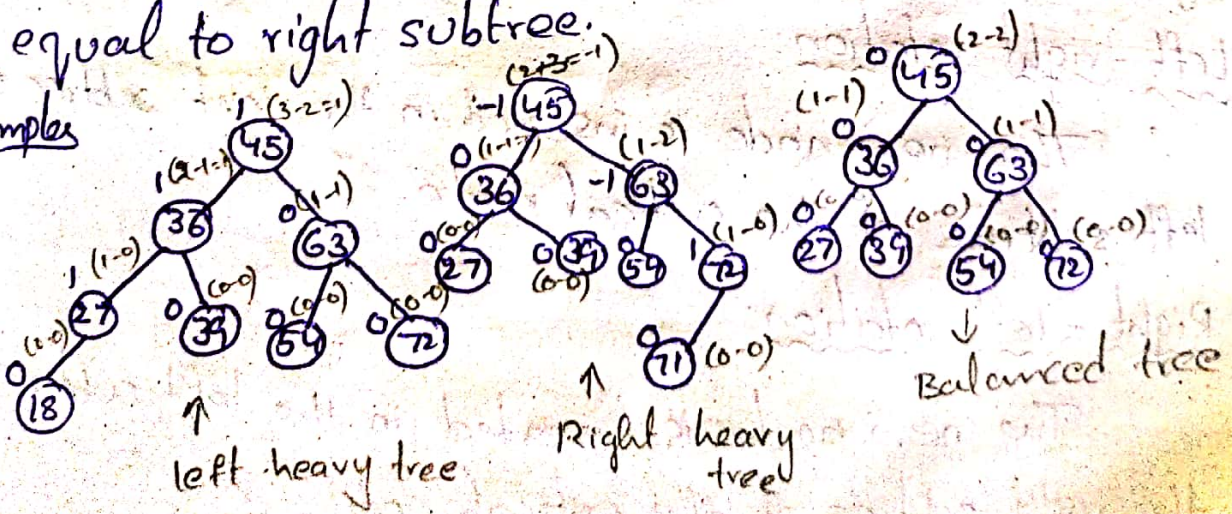
In an AVL tree the height of the left subtree and right subtree of a node differ by atmost 1. Hence this tree is also known as height balanced tree.

The structure of AVL tree is same as BST but with a little difference that it stores an addition al variable called balance factor given by

Height of left subtree - Height of right subtree

* If the balance factor is '1' then the left subtree is considered to be one level heighier than the right subtree. such a tree is called left heavy tree.
* If the balance factor is '-1' then it means that the left subtree is one level lower than the right subtree.
* If the balance factor is 'zero' then left subtree is equal to right subtree.

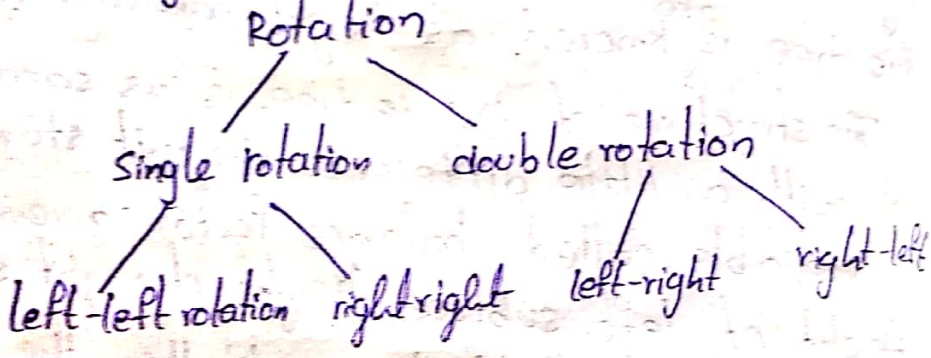
Examples



Insertions and deletions on an AVL tree may disturb the balance factor of the node and thus rebalancing of the tree has to be done.

The tree is rebalanced by performing rotation at the critical node (it is a nearest ancestor node on the path from the inserted node to the root whose balance factor is neither -1, 0, nor 1

There are 4 types of rotations.



Left-left rotation:

The new node is inserted in the left subtree of the left subtree of the critical node.

Right-Right rotation:

The new node is inserted in the right subtree of the right subtree of the critical node.

Left-right rotation:

The new node is inserted in the right subtree of the left subtree of the critical node.

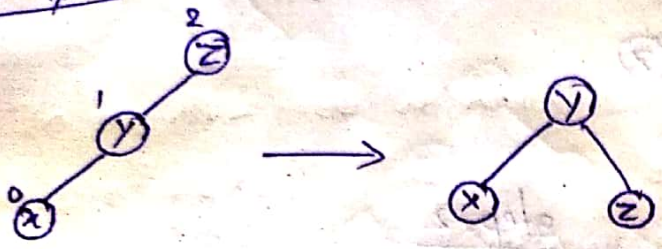
Right-Left rotation:

The new node is inserted in the left subtree of the right subtree of the critical node.

Left - Left rotation:

Pull the median node to the top of the tree and assign it as root and take the root node and push it down to the right side

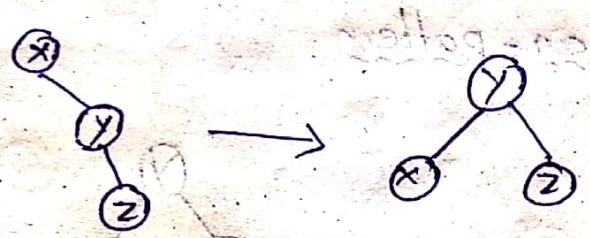
Example



Right - Right rotation:

Pull the middle node to the top of the tree and push down the root node to the left side.

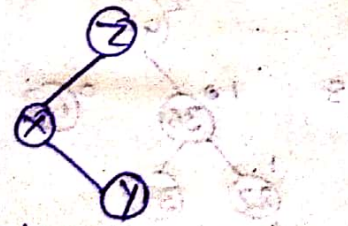
Example



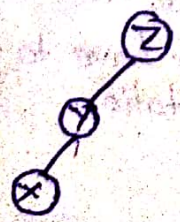
Left - Right Rotation:

From the given tree obtain the left skewed tree and then apply the Right rotation

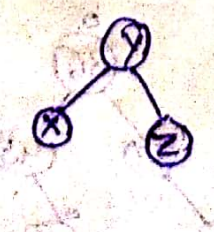
Example



Step - 1



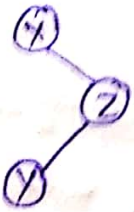
Step - 2



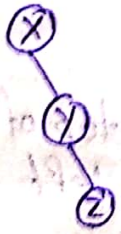
Right - left rotation:

From the given tree obtain the right skewed tree. Then apply the left rotation

Example

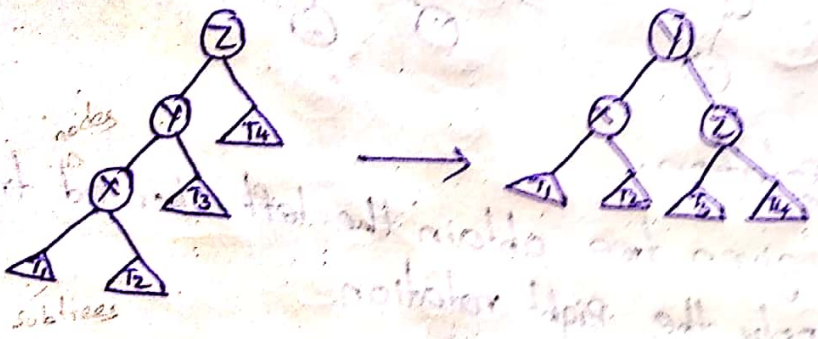


step-1

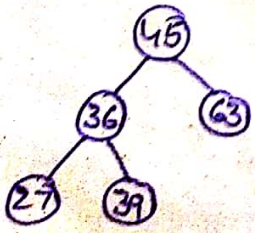


step-2

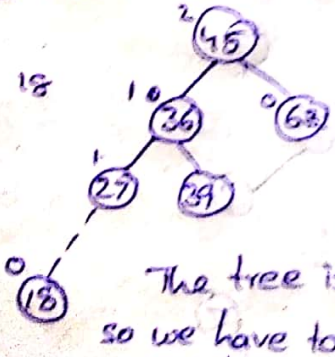
Left - left rotation - pattern:



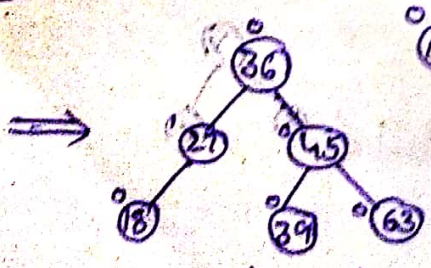
Example consider the given tree insert 18 in the given



Add insert 18

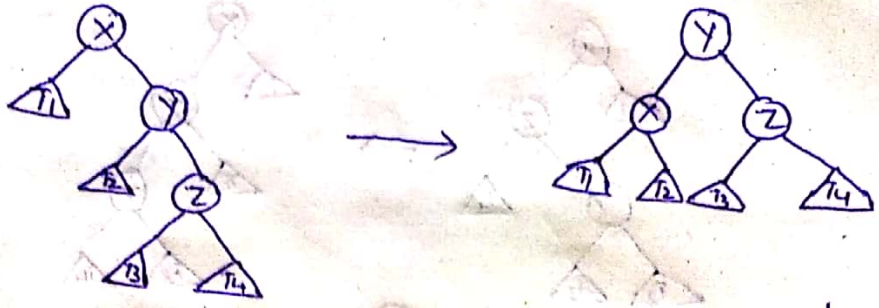


The tree is inbalan
so we have to apply LL
Rotation.

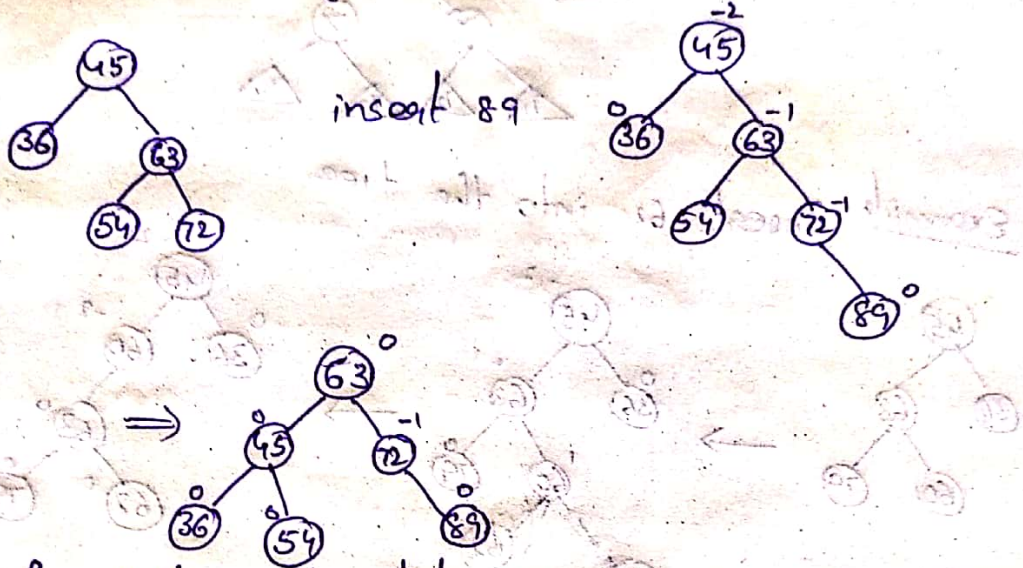


Balanced tree

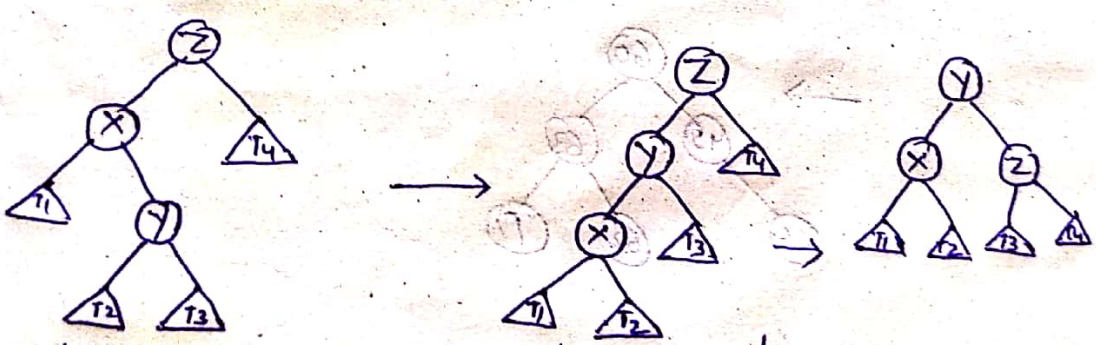
Pattern for Right-Right Rotation



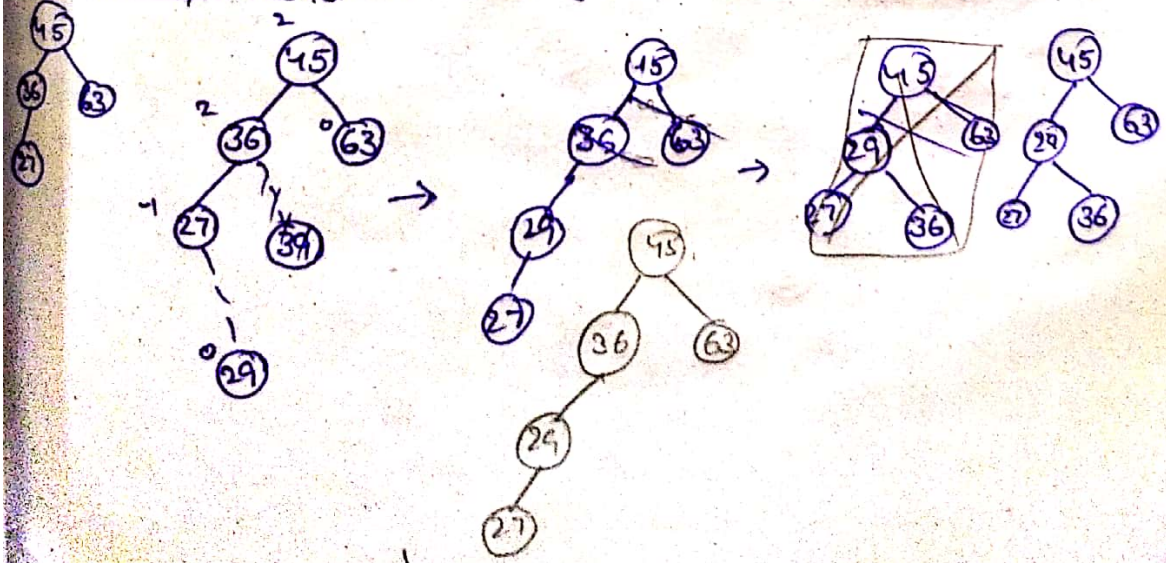
Example Consider the given tree and insert 89



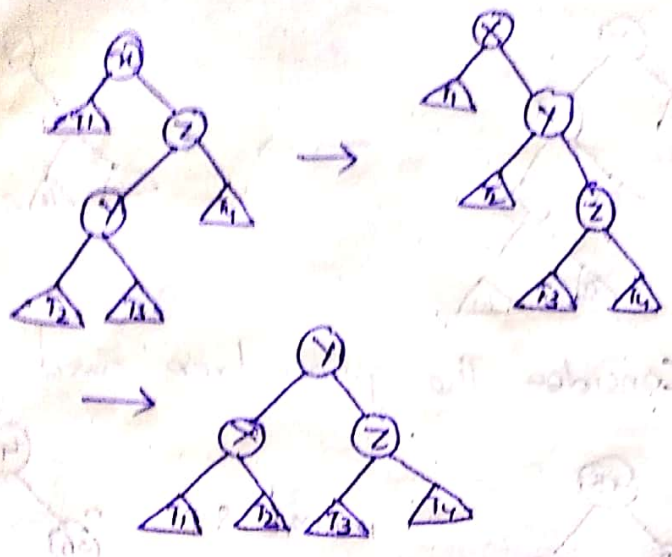
Pattern for left-Right rotation:



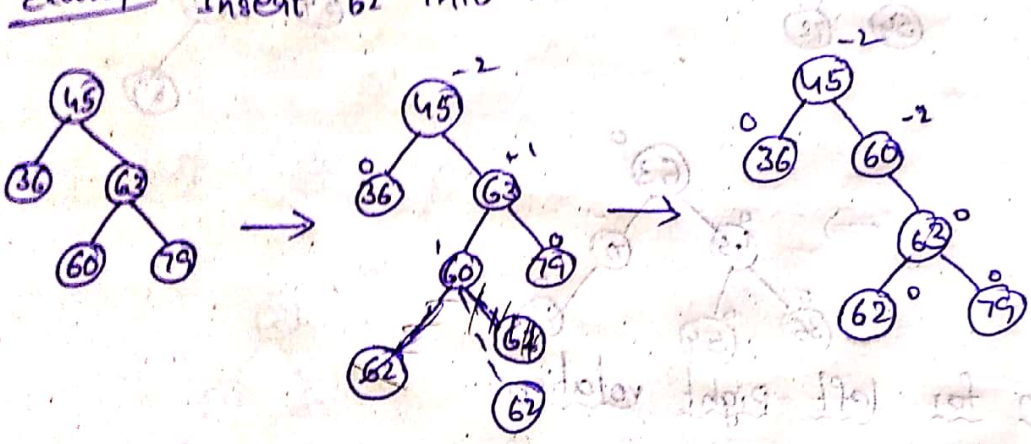
Example Consider the given tree insert 29

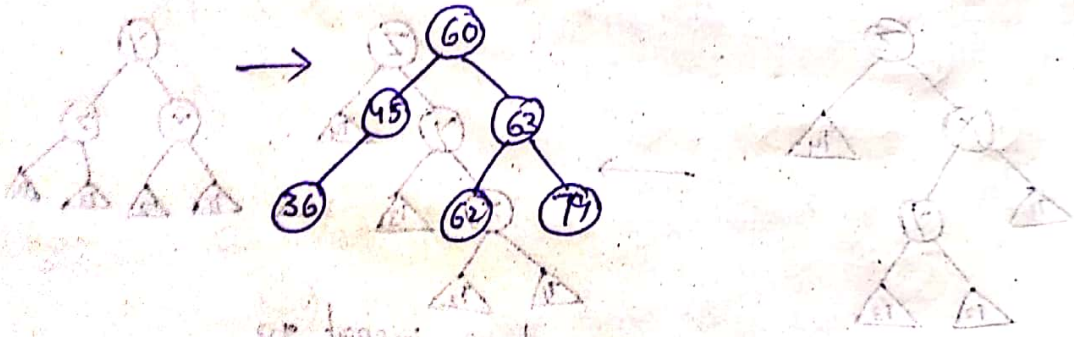


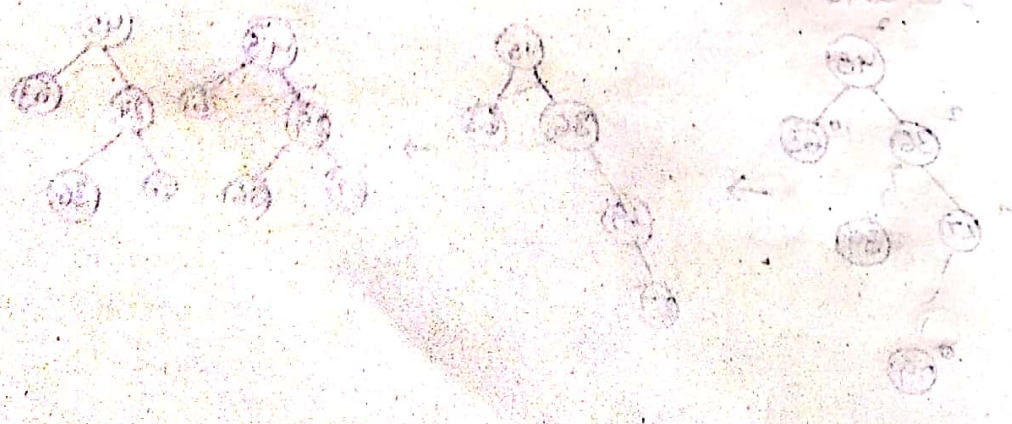
Right left rotation:



Example Insert 62 into the tree







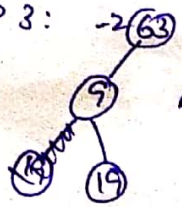
Scanned by CamScanner

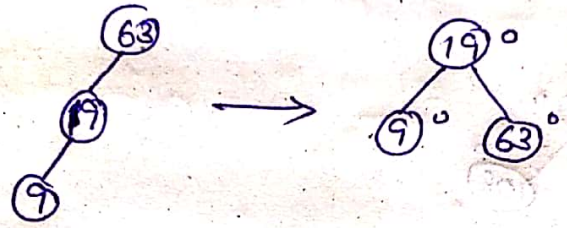
*Construct an AVL tree from the given list of nodes

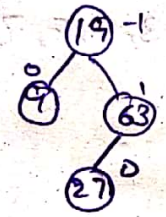
63, 9, 19, 27, 108, 108, 99, 81

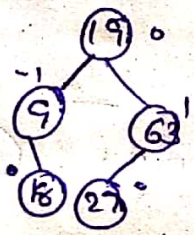
Sol step 1 :  Make 63 as root

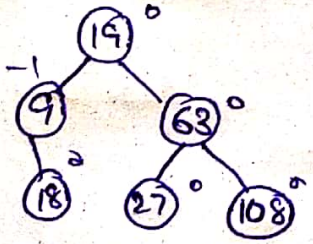
step 2 : 

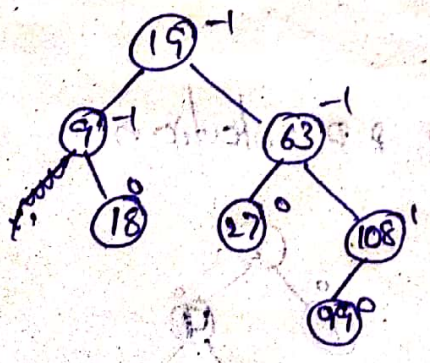
step 3:  Apply LR rotation

3.1 

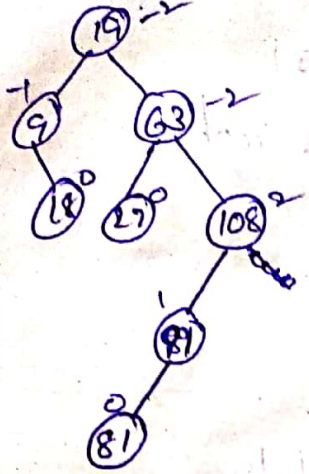
step 4: 

step 5: 

step 6: 

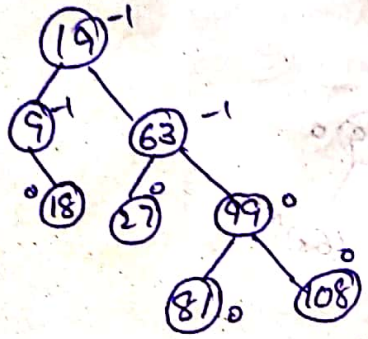
step 7: 

step 8:



Apply L-L Rotation

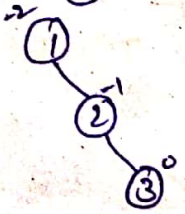
8-1

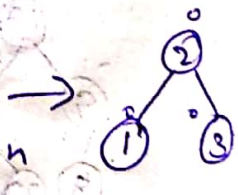


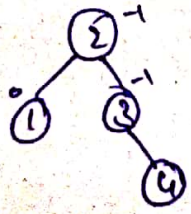
Construct an AVL tree for 1, 2, 3, 4, 8, 7, 6, 5, 11, 10, 12

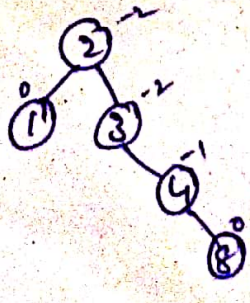
step 1  Make 1 as root

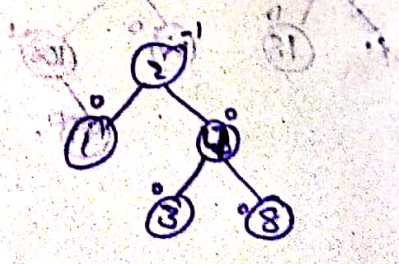
step 2 

step 3 

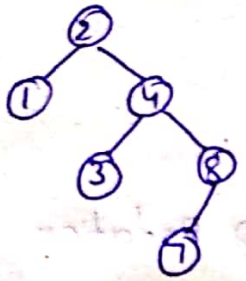
Apply R-R rotation 

step 4 

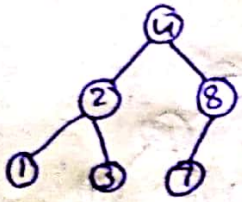
step 5 

Apply R-R rotation 

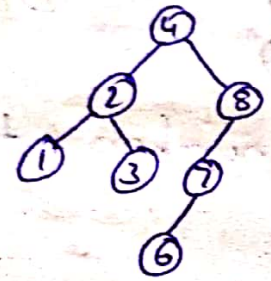
step-6



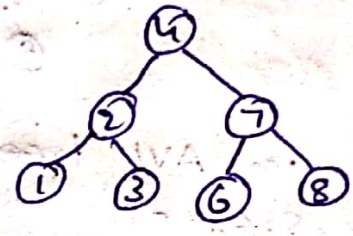
Apply RR Rotation.



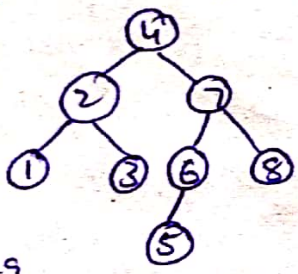
step-7



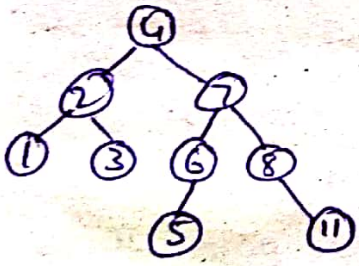
LL Rotation



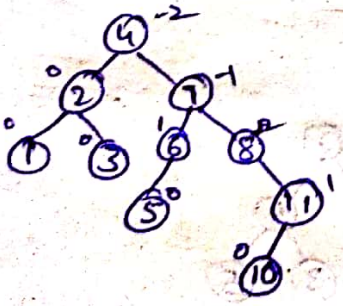
step-8



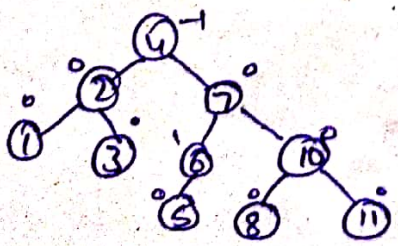
step-9

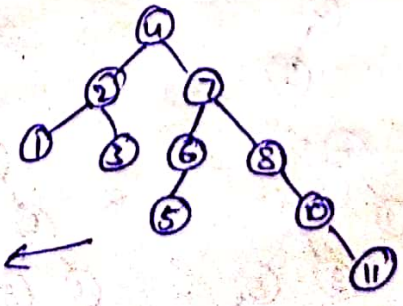


step-10

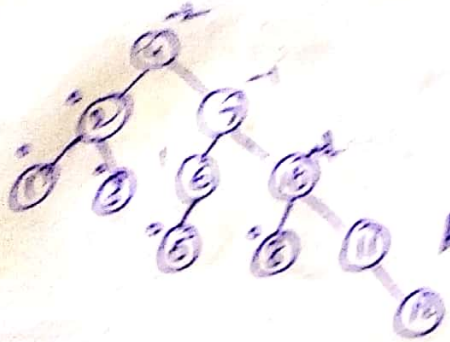


R-L Rotation





Step-1



RR Rotation

→

Construct an AVL tree from node numbers 1 to 8
(1, 2, 3, 4, 5, 6, 7, 8)

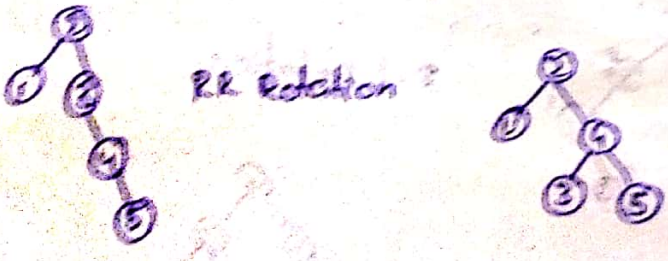
Step-1 make '1' as root

Step-2

Step-3

Step-4

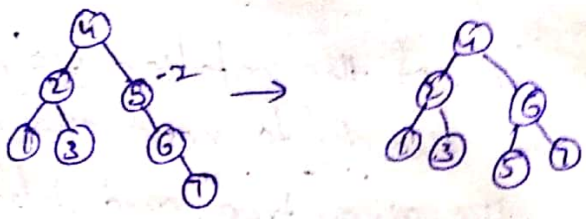
Step-5



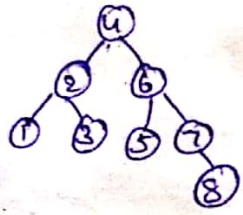
RR Rotation :

Step-6

step-7



step 8



Deletion of a node in an AVL tree.

Like in binary search tree we have to remove a node by considering any one of the following 3 cases

① If the node is a leaf node then we can directly remove the node.

② If the node contains only one child then replace the nodes data with the childs data.

③ If the deleted node contains both children then we can replace it with inorder predecessor nodes data or inorder successor nodes data.

while performing deletion the balance factor of any node changes then we have to apply rotations like in insertion. There are two types of rotations

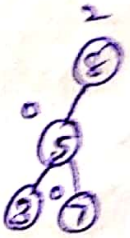
Left Rotation	Right Rotation
L(0), L(-1) $\Rightarrow$ RR	R(0), R(1) $\Rightarrow$ LL
L(1) $\Rightarrow$ RL	R(-1) $\Rightarrow$ LR

If the deleted node is a left child to the parent node (critical node) then we have to apply the left rotation otherwise we have to apply the Right Rotation.

To select the type of the rotation we must check balance factor of the delete node sibling.

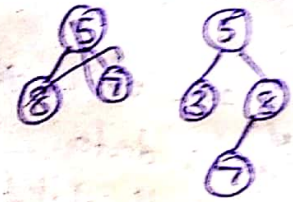
Example ① Consider the given tree and delete 10 from it

Delete 10

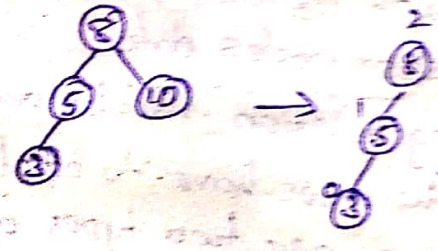


Since 10 is right child of the present node & we have to apply the rotation on the sibling of 10 is 5 5's balance factor is 0. So rotation is of type R(0)

R(0) → we have to apply LL Rotation



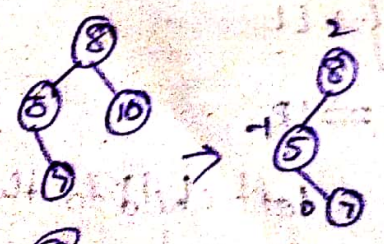
② Consider the given tree and delete 10



sibling balance factor is 1

∴ R(1) → LL Rotation

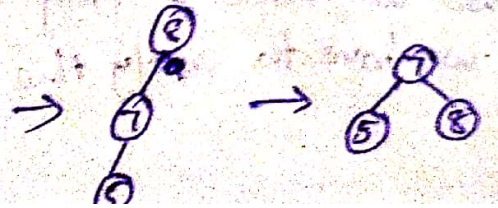
②



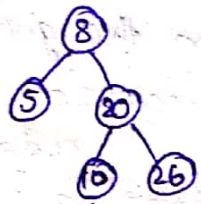
Here sibling balance factor is -1

R(-1)

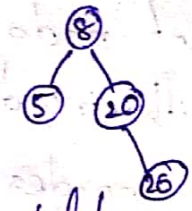
LR rotation



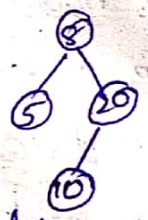
(1) delete 5



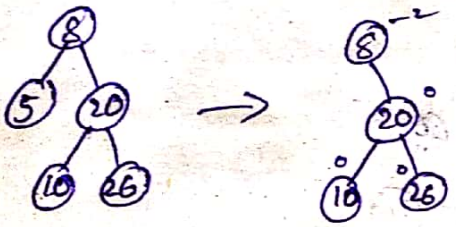
(2) delete 5



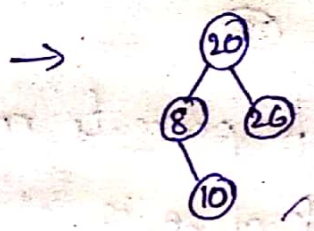
(3) delete 5.



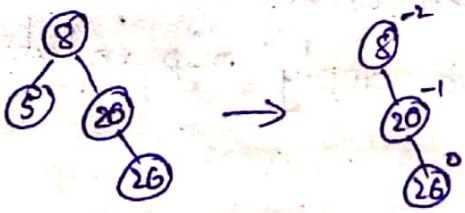
(1)



L(0) = RR rotation

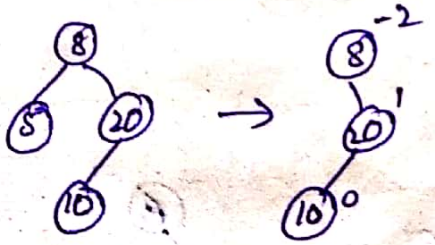


(2)

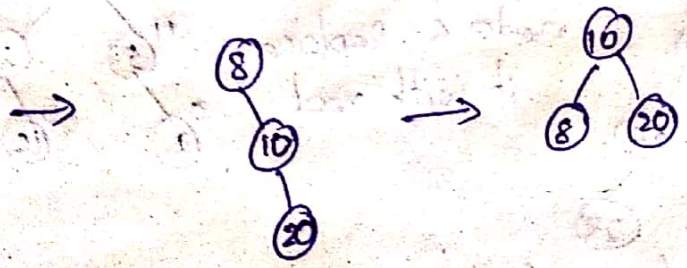


L(-1) = RR rotation

(3)



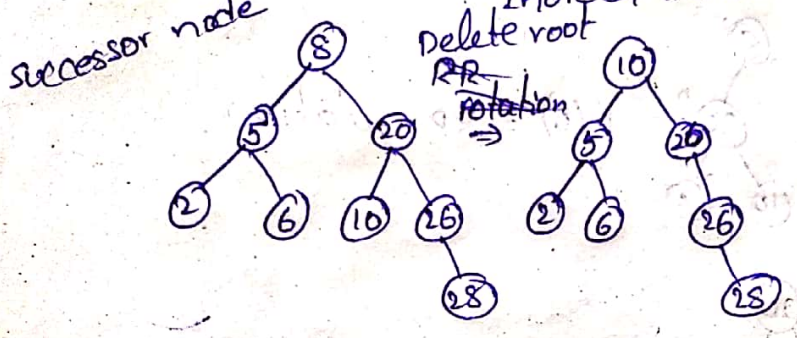
L(1) = RL rotation



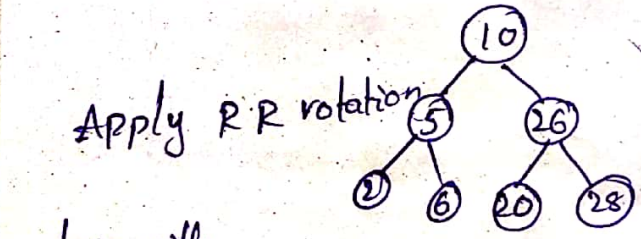
Delete root node:

Inorder to delete a root node from the given tree we have to replace it with inorder successor or inorder predessessor node. The inorder successor is 10 and predeccessor node is 6.

Inorder traversal- 2, 5, 6, 8, 10, 20, 26, 28

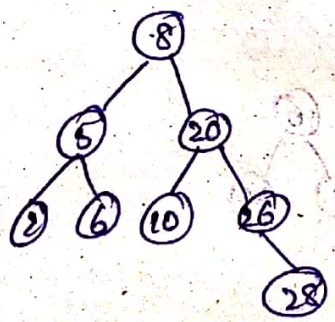


Here 10 is the left child of its parents node 20 and 10's sibling is 26 and bf of 26 is -1. So the rotation of L(-1) i.e we have apply R-R rotation

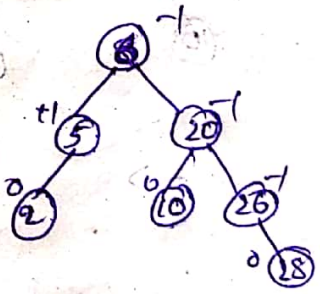


2, 5, 6, 8, 10, 20, 26, 28

replace with Predecessor node



predeccessor node 6. Replace it with root



Here the balance factor of every node is lies b/w -1, 0 and 1. Therefore we need not apply any rotation.

Heap:

Heap is a special tree based data structures in which tree is a complete binary tree.

A binary heap is a type of heap following the properties of binary tree.

1. It should be a complete tree All (continue)
   - All levels are completely filled except possibily the last level. The last level has all the keys as left as possible
2. A binary heap is either min heap or maxheap

Representation of binary heap in memory:

It is stored in an array if the root element is stored at zeroth position then we can obtain the left child by A[(2*i)+1] where 'i' is the index of the root element. Right child is obtained by A[(2*i)+2] where 'i' is the index of the root element.

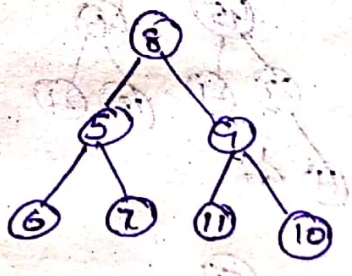
If the index of the root element is 'i' then we can obtain the left child as l=2*i (or) A[2*i] and we can obtain the right child as r=(2*i)+1 (or) A[(2*i)+1]

root index=0

a[0]	1	2	3	4	5	6
8	5	9	6	7	11	10

root index=1

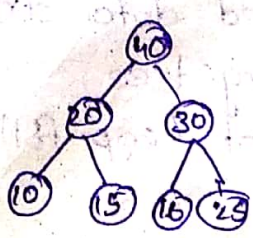
A[]	2	3	4	5	6	7
8	5	9	6	7	11	10



There are two types in binary heaps:

1. Max heap: In a max heap the value present at root node must be greater than its left child and right child. The same property is applied for all subtrees in that binary tree.

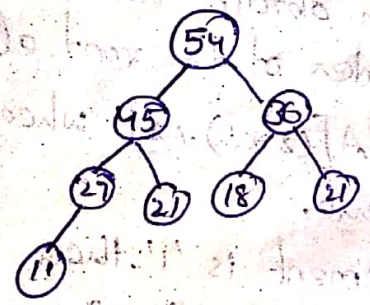
Example



2. Min heap: In min heap the value present at root node must be minimum among the values present at all of its children. This property must be recursive applied for all the subtrees in the binary tree.

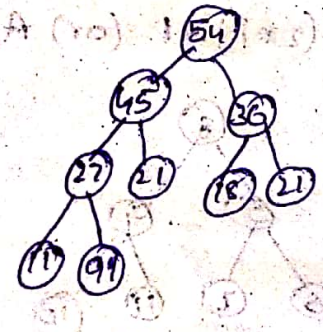
Insertion of an element in max heap:

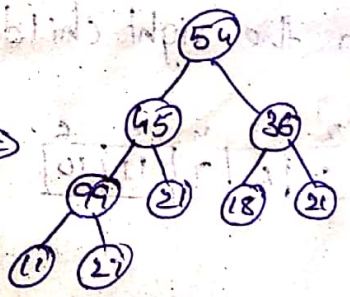
Consider the given tree and insert 99 int it

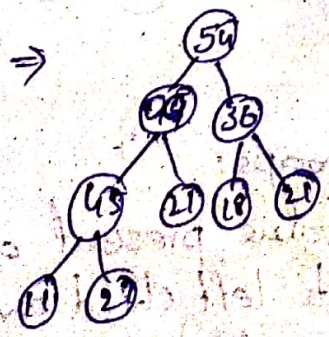


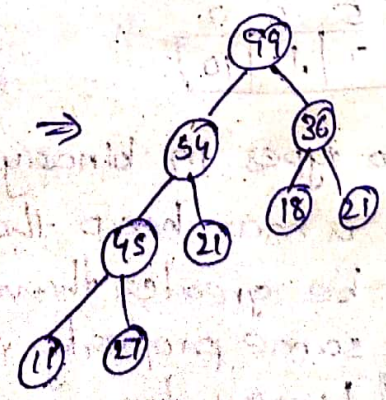
insert 99

step1 Reconstruct max heap









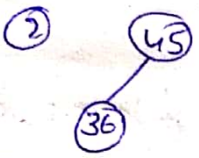
Max heap.

Construct a max heap from the given list of nodes.

45, 36, 54, 27, 63, 72, 61, 18

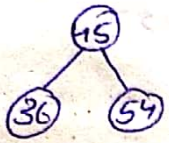
Step 1 (45) Assign 45 as root

(2)

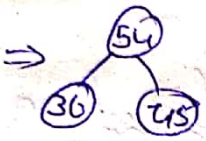


36 < 45 so it is assigned as left child

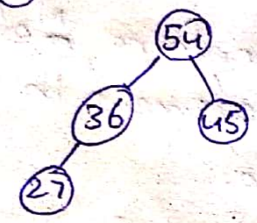
(3)



54 > 45. So, swap 45, 54 then the tree will be

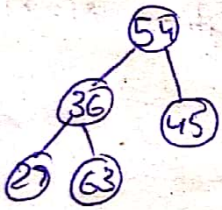


(4)

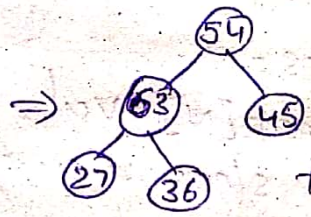


27 < 36. So it is assigned as left child for 36

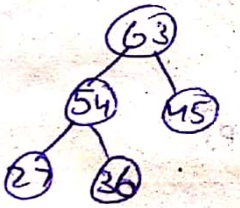
(5)



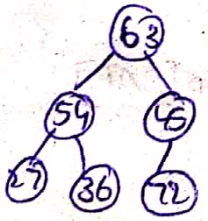
63 > 36. So swap 36, 63 then the tree will be



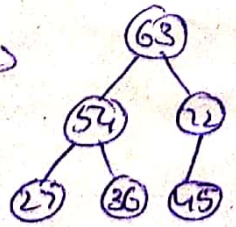
Here 63 > 54 So swap 63 and 54 then the tree will be



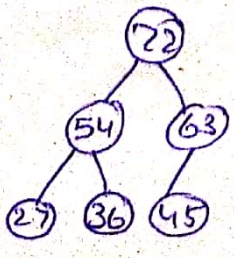
(6)



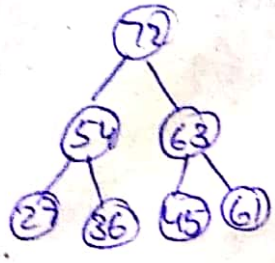
Here 72 > 45. So swap 72 and 45 then the tree will be



Here 72 > 63. So swap 63 and 72 then the tree will be

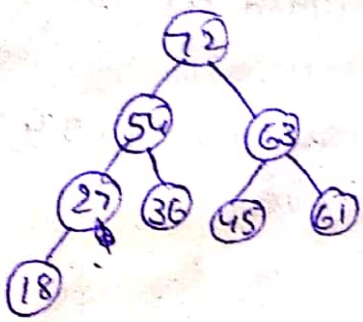


(7)



Here 61 < 63 . So it is assigned as right child of 63

(8)



Here 18 < 27 . so it is assigned as leftchild for 27

Algorithm for insertion of an element in max heap:

Assume heap is an array which contains N elements. POS is position of a last inserted element in the array. PAR denotes the parent node.

step 1: Add new value and set its position N = N + 1 and POS = N

step 2: set HEAP[N] = VAL

step 3: Repeat steps 4 and 5 while POS ≥ 1

step 4: set PAR = POS/2

step 5: if HEAP[POS] ≥ HEAP[PAR] then go to step 6

else swap HEAP[POS] and HEAP[PAR]

set POS = PAR

step 6: Return

(Insert an element into mean heap)

Build min heap with the value 35, ~~33, 42,~~ 22, 78, 10, 56, 12, 7

Sol

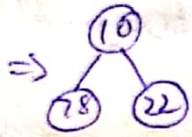
(1) (22) Assign 22 as a root

(2)

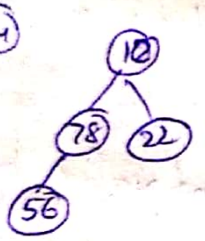
78 > 22 · so it is assigned as left child for 22

(3)

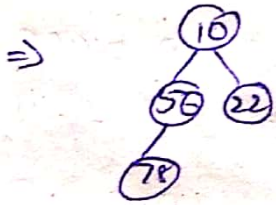
10 < 22 swap 10 and 22



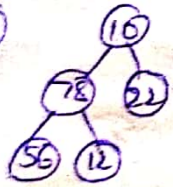
(4)



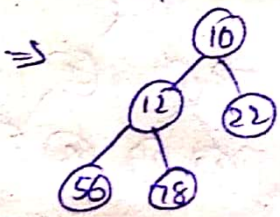
56 < 78, swap 56 and 78



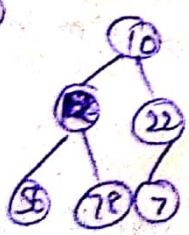
(5)



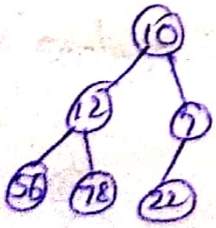
12 < 78 · swap 12 and 78



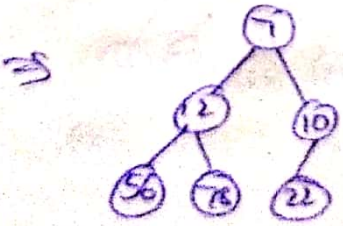
(6)



7 < 22 Swap 7 and 22



7 < 10 · Swap 7 and 10



Deletion of an element from maxheap:

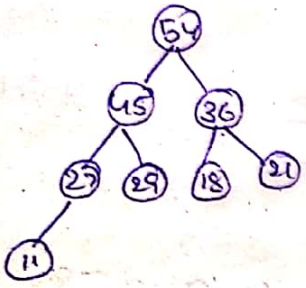
Inorder to delete an element the required 3 steps

Step 1: Replace the root node value with the last node value so that the tree is still a complete binary tree but not necessarily a heap.

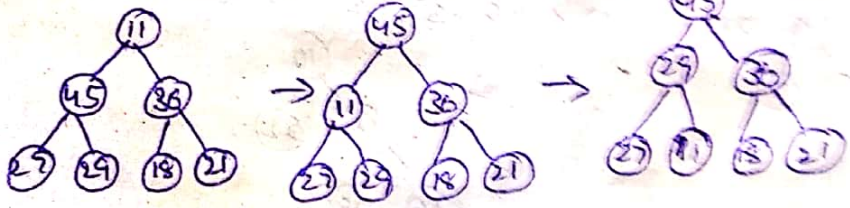
Step 2: Delete the last node

Step 3: sink down the new roots value so that the tree satisfies the heap property.
In this step interchange the root nodes value with its child value.

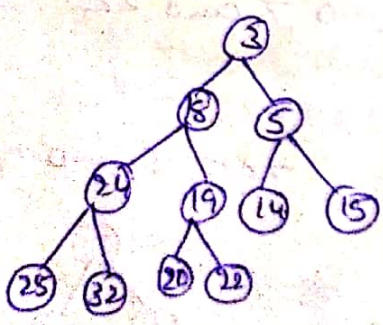
Consider the given tree and delete the root node 54



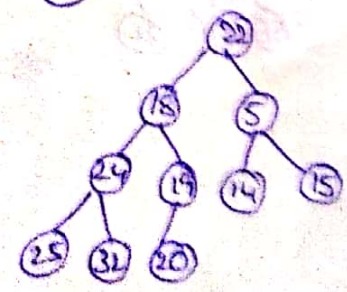
Replace root value by 11

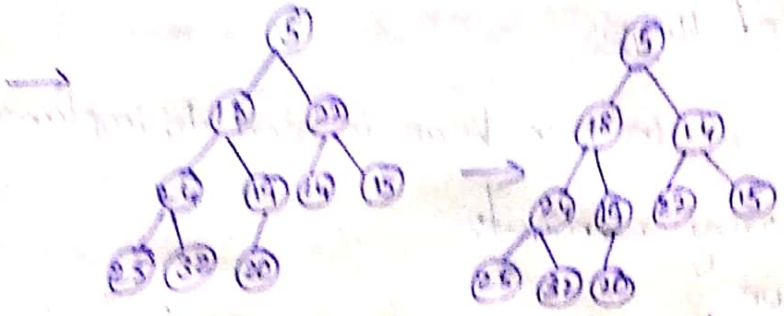


Consider the given tree and delete 3 from the maxheap



Replace root value by 32





Algorithm for deletion of an element from max heap:

step1: Remove the last node from the heap
set last = HEAP[N] and N = N - 1

step2: set PTR = 1, LEFT = 2, RIGHT = 3

step3: set HEAP[PTR] = last

step4: Repeat steps 5 to 8 while left <= N

step5: if (HEAP[PTR] >= HEAP[LEFT] && HEAP[PTR] >= HEAP[RIGHT])
then goto step 9

step6: if HEAP[PTR] < HEAP[LEFT]

step6.1: swap HEAP[PTR] and HEAP[LEFT]

step6.2: PTR = LEFT

step7: if HEAP[PTR] < HEAP[RIGHT]

step7.1: swap HEAP[PTR] and HEAP[RIGHT]

step7.2: PTR = RIGHT

step8: set LEFT = 2 * PTR,
set RIGHT = 2 * PTR + 1,

Applications of Heaps:

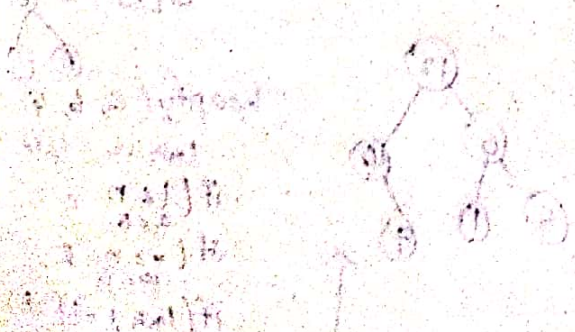
A max heap or in min heap is used to implement

1. sorting of array elements
2. priority queue

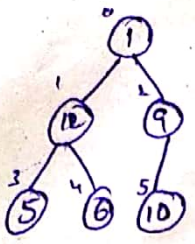
Heap sort

```
#include<stdio.h>
void heapify(int a[], int n, int i)
{
    int largest = i;
    int left = 2*i+1;
    int right = 2*i+2;
    if(left<n && a[left] > a[largest])
        largest = left;
    if(right<n && a[right] > a[largest])
        largest = right;
    if(largest != i)
    {
        int t;
        t = a[largest];
        a[largest] = a[i];
        a[i] = t;
    }
    heapify(a, n, largest);
}
}
void heapsort(int a[], int n)
{ int i;
  for(i = n/2-1; i>=0; i--)
  { heapify(a,n,i);
  }
```

```
for(i=n-1;i>=0;i--)
 {   int t;
     t=a[0];
     a[0]=a[i];
     a[i]=t;
      heapify(a,i,0)
   }
}
void printarray(int a[],int n)
{    int i;
     for(i=0;i<n;i++)
        printf("%d ",a[i]);
}
void main()
{  int a[20],i,n;
    printf("enter n");
    scanf("%d",&n);
    printf("enter elements");
    for(i=0;i<n;i++)
    { scanf("%d",a[i]);
    }
    heapsort(a,n);
    printf("elements after sorting\n");
     printarray(a,n);
}
```



Example



a[0]	1	2	3	4	a[5]
1	12	9	5	6	10

i = 2

heapify(a, n, i) = heapify(a, 6, 2)

largest = 2

left = 5

right = 6

if(left < n && a[l] > a[largest])
(5 < 6 && 10 > 9) T

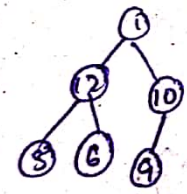
largest = left = 5

if(right < n)
false

if(largest != i)
5 != 2

swap a[2] and a[5]

a[5] = 9 a[2] = 10



heapify(a, 6, 5)

largest = 5

left = 11

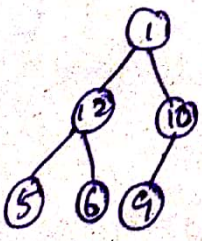
right = 12

if(left < 6) F

if(right < 6) F

if(largest != i)

5 != 5 F



i = 1

heapify(a, n, i)

= heapify(a, 6, 1)

largest = 1

left = 3

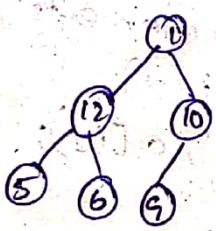
right = 4

if(left < n && a[l] > a[largest])
3 < 6 5 > 12
F

if(right < n && a[r] > a[largest])
4 < 6 6 > 12 F

if(largest != i)

if(1 != 1) F



i = 0

heapify(a, n, i) =

heapify(a, 6, 0)

largest = 0

left = 1

right = 2

if(left < n && a[l] > a[largest])
1 < 6 12 > 1

largest = 1

if(right < n && a[r] > a[largest])
2 < 6 && 10 > 12
F

if(largest != i)

1 != 0 (T)

Swap a[0] & a[1]

a[0] = 12

a[1] = 1

heapify(a, n, i)

= heapify(a, 6, 1)

largest = 1

left = 3

right = 4

if(l < n && a[l] > a[largest])
3 < 6 5 > 1

largest = 3

if(r < n && a[r] > a[largest])
4 < 6 6 > 5

largest = 4

if(largest != i)

Swap a[4] and a[1]

a[4] = 1

a[1] = 6

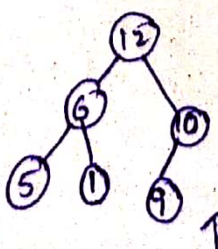
heapify(a, 6, 4)

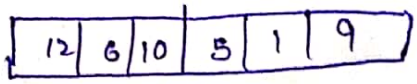
largest = 4, left = 9, r = 10

if(l < n
9 < 6 F

if(r < n
10 < 6 F

if(largest != i) F
4 4

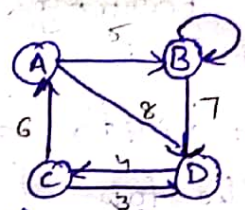




Unit-6

Graphs

(1)

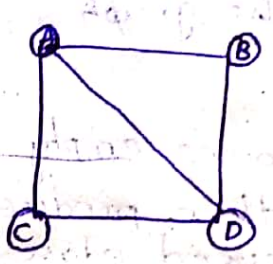


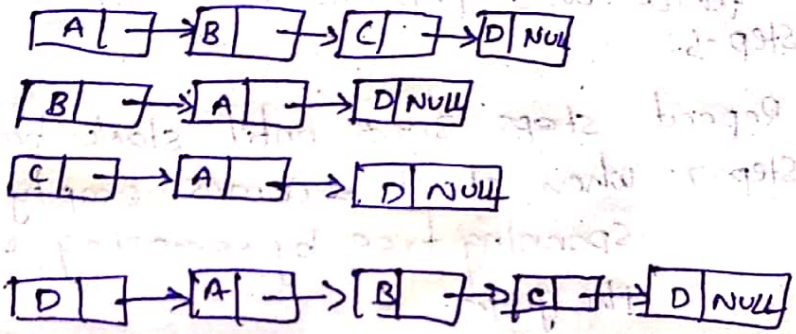
	A	B	C	D
A	0	1	0	1
B	0	1	0	1
C	1	0	0	1
D	0	0	1	0

	A	B	C	D
A	0	5	0	8
B	0	9	0	7
C	6	0	0	4
D	0	0	3	0

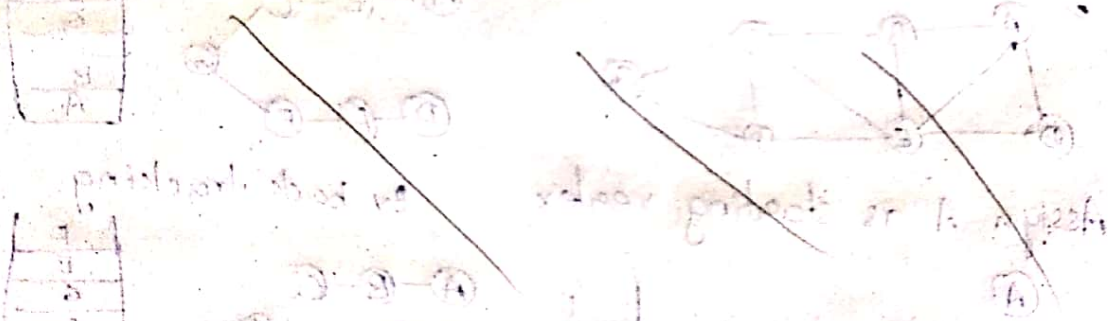
Adjustancy array

(2)





Adjustancy linked list



Graph traversal algorithms:

* DFS- Depth first search / traversal (stack)
* BFS- Breadth first seaach/ traversal (queue)

Algorithm for DFS:

step 1: Define a stack size total number of vertices in the graph.

step 2: Select any vertex as starting point for traversal visit that vertex and push it on the stack

step 3: visit any one of the non-visitor adjacent vertex of a stack which is at the top of the stack and push it on the stack.

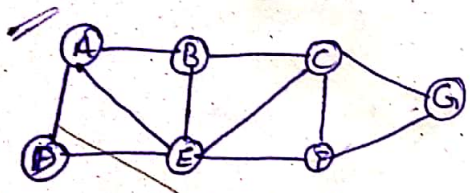
step 4: Repeat step 3 until there is no new vertex to be visited from the ~~circuit~~ vertex which is at the top of the stack.

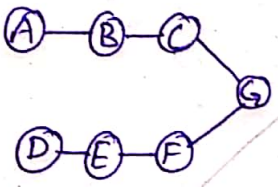
step-5:
when there is no new vertex to visit then use back tracks and pop one vertex from the graph

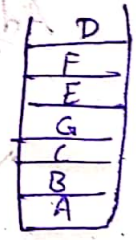
step-6:
Repeat steps 3,4,5 until stack becomes empty

step-7: when stack becomes empty then produce final spanning tree by removing unused edges from the graph.

Example

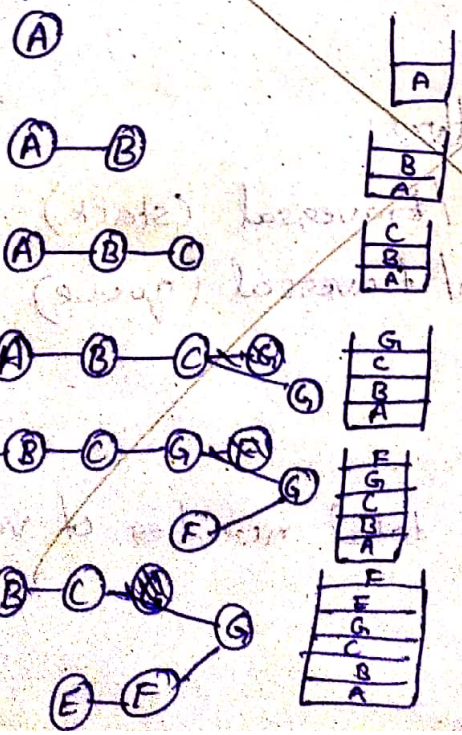


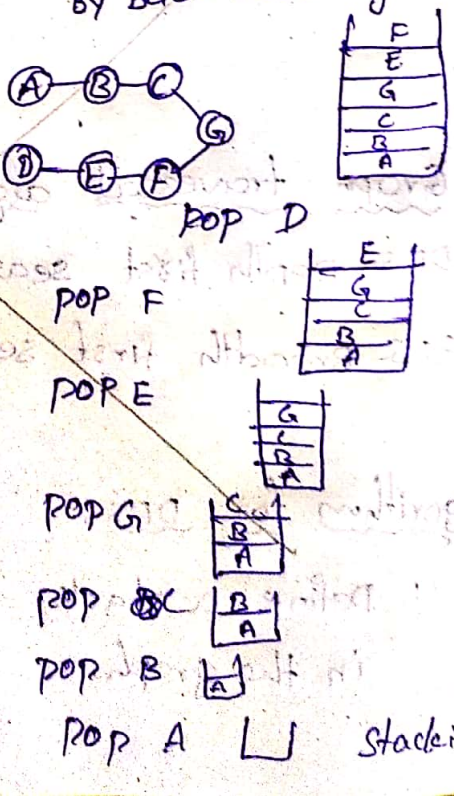




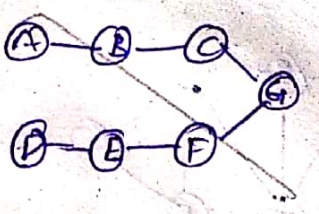
Assign A as starting vertex

By back tracking

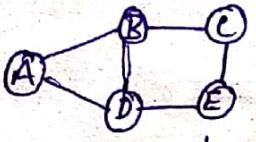




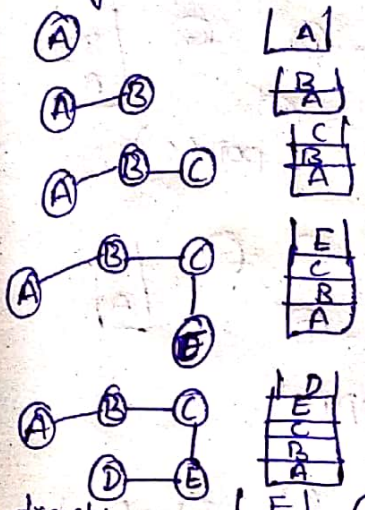
spanning tree obtained is



Example

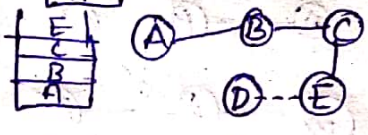


Assign A as starting vertex



Back tracking

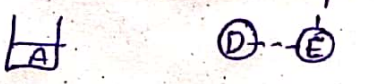
POP D



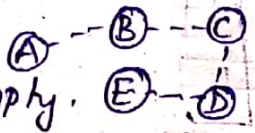
POP E

POP C

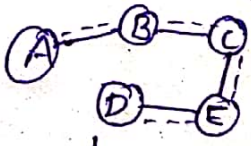
POP B



POP A ⌊ stack is empty.

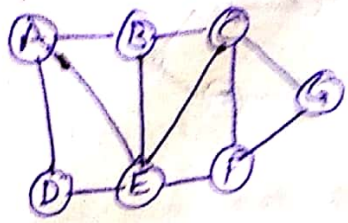


The final spanning tree is

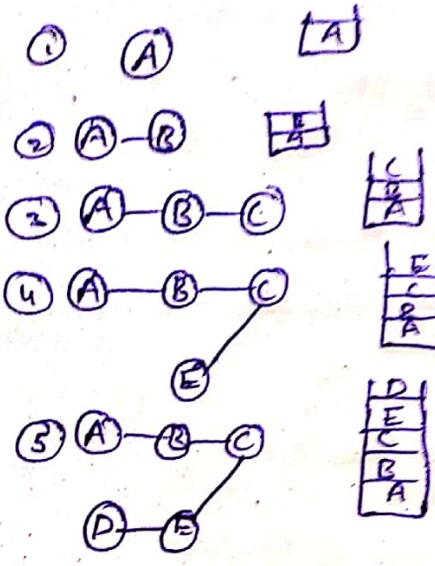


Adjacent matrix

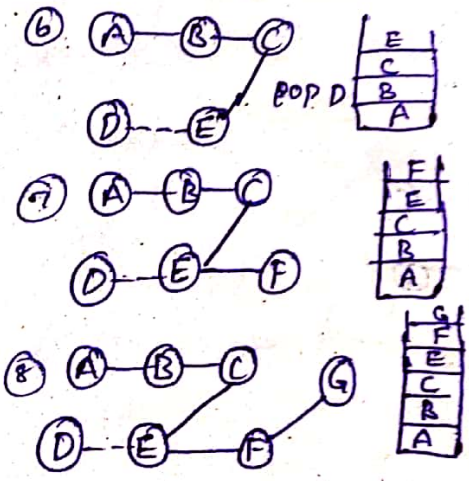
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0



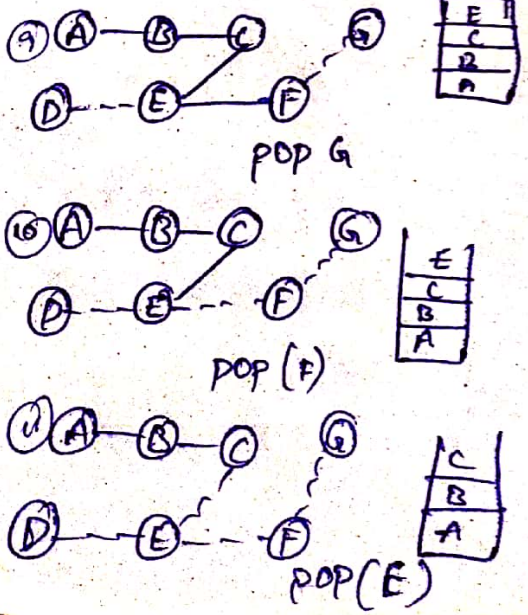
Assign A as starting vertex

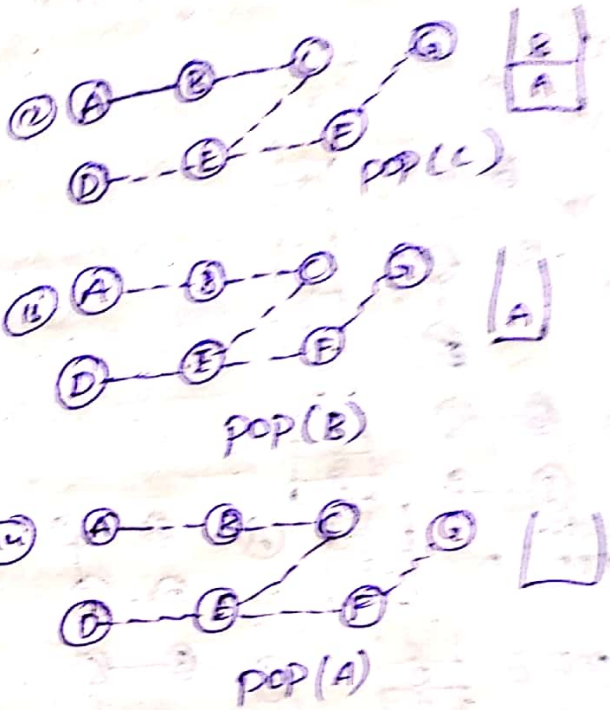


By back tracking



By backtracking





shortest path algorithms:

There are three different algorithms to calculate the shortest path between the vertices of a graph

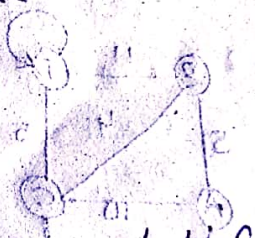
1. Minimum spanning tree
2. Dijkstra's
3. Warshall's

Minimum spanning tree:

A spanning tree of a connected graph with no cycles or loops. A minimum spanning tree is defined as a spanning tree with weight less than or equal to every other spanning tree.

There are two algorithms that are used to calculate the minimum cost of the spanning tree

1. Prims algorithm
2. Kruskals algorithm



Prims algorithm:

It is a greedy algorithm that is used to form minimal cost spanning tree. This algorithm builds a tree that includes every vertex and a subset of edges in such a way that the total weight of all the edges in the tree is minimise.

There are three type of vertices in this algorithm

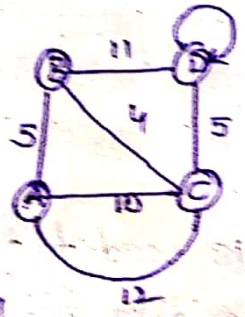
1. Tree vertices: vertices that are part of minimum spanning tree
2. Fringe vertices: vertices that are currently not a part of tree but are adjacent to some tree vertex
3. Unseen vertices: vertices that are neither tree nor fringevertices are unseen vertices.

Prims algorithm procedure:

step 1: choose a starting vertex A
step 2: Add the fring vertices (that are adjacent t
step 3: select an edge connecting the tree vertex and the fringe vertex that has minimum weight and add the selected edge and vertex to the minimum spanning tree.
Repeat the about steps until we find all the number of vertices in the original graph

Note:

First remove loops and parallel edges in the given graph



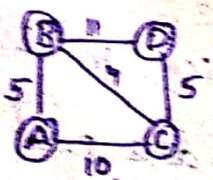
step-1

Remove loops and parallel edges

① D
② A → C

step-2

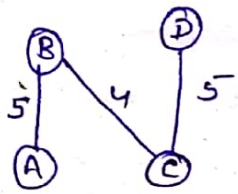
Then obtained graph will be



step3: Adjacent matrix for the graph

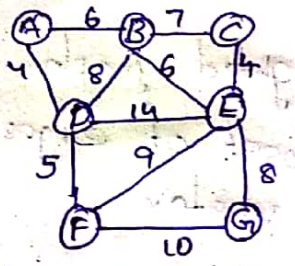
	A	B	C	D
A	-	5	10	∞
B	5	-	4	11
C	10	4	-	5
D	∞	11	5	-

step 4 spanning tree



sum = 5+4+5 = 14

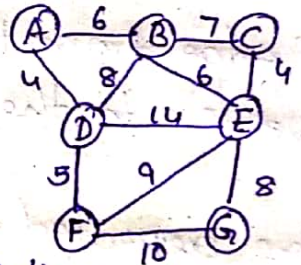
②



step-1 Remove loops and parallel edges

There is no loops and parallel edges.

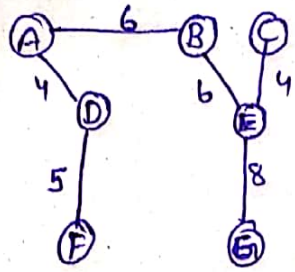
step-2 Then the graph will be:



step-3 Adjacent matrix for the graph

	A	B	C	D	E	F	G
A	-	6	∞	4	∞	∞	∞
B	6	-	7	8	6	∞	∞
C	∞	7	-	∞	4	∞	∞
D	4	8	∞	-	14	5	∞
E	∞	6	4	14	-	9	8
F	∞	∞	∞	5	9	-	10
G	∞	∞	∞	∞	8	10	-

step 4 -



Kruskals algorithm

This algorithm is also used to find the minim
Spanning tree for a connected weighted graph.

This algorithm aims to find subset of edges that forms a tree that includes every vertex

Procedure:

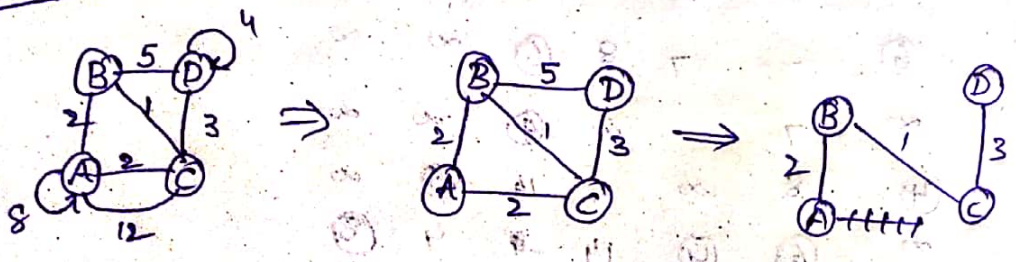
step -1 start from a vertex and find the minimu
Cost edge from that vertex

step-2 Find the minimum cost edge from the starting vertex and adjacent vertex to the starting vertex.

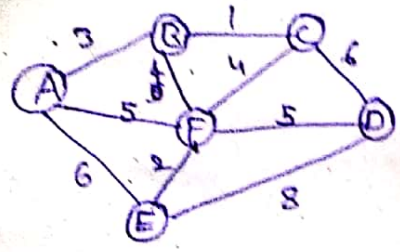
step-3 The above two steps are repeated for every vertex in the graph.

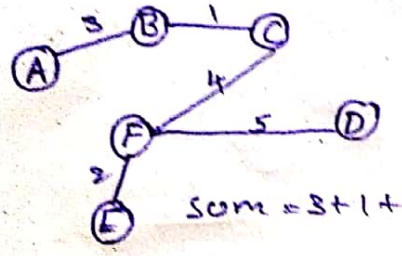
step-4 If we obtain all the vertices in the graph tha we can stop the process.

Example



a)





som = 3+1+4+5+2 = 15