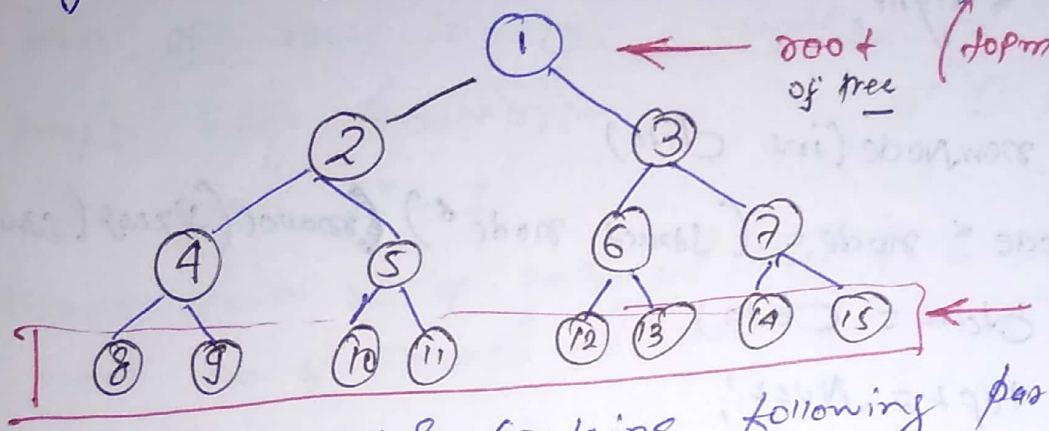
Binary Tree

A tree whose elements have at most 2 children is called binary tree. Name as left and right child.



Binary tree nodes contains following parts :-

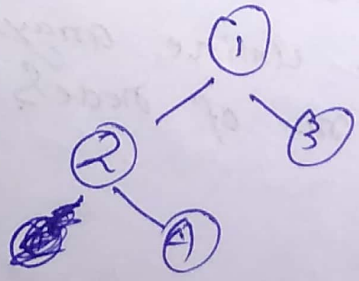
→ Data
→ pointer to left child
→ pointer to right child.

Unlike, array, linked list, stack, queues, Trees are hierarichical data structures rather than linear data structures.

→ Trees provides moderate access/search (quicker than linked list and slower than arrays)
→ provides moderate insertion/deletion (quicker than arrays and slower than linked list)
→ Like linked list unlike arrays it do not have upper limits on no. of nodes (as it linked through pointers).

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

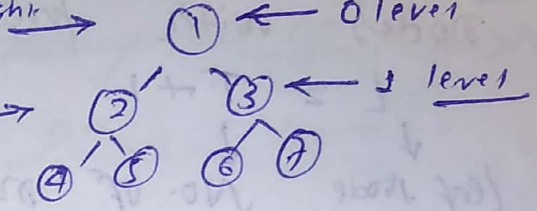
```
Struct node
{ int data;
  Struct node * left;
  struct node * right;
};
Struct node * newNode (int data)
{ Struct node * node = (struct node *) malloc (sizeof (struct node));
  node -> data = data;
  node -> left = NULL;
  node -> right = NULL;
  return (node);
}
int main ()
{ Struct node *root = newNode (1);
  root -> left = newNode (2);
  root -> right = newNode (3);
  root -> left -> right = newNode (4);
  return 0;
}
```



Properties :-

1) maximum number of nodes at level 'l' of binary tree is 2^l.

level of root is 0.



Proof :- Using induction :-

for, $l=0$ $\quad 2^0 = 1$

Assume max^m no. of nodes at level $l = 2^l$

then no. of nodes at level $l+1 = 2^{l+1} = 2 * 2^l$ satisfies since if nodes have max^m 2 children.

2) max^m number of nodes in binary tree of height 'h' is $2^h - 1$.

→ **Proof :-**

max^m no. of nodes $= 2^0 + 2^1 + 2^2 + \cdots + 2^{(h-1)}$

$= \frac{2^0(2^h - 1)}{2-1} = 2^h - 1.$

3) Binary tree with N nodes, min^m possible height or min^m no. of levels is $\log_2(N+1)$

Proof :- $2^h - 1 = N$

$\Rightarrow h = \log_2(N+1)$, $l = \log_2(N+1) - 1$.

level

4) Binary tree with L Leaves has at least levels :-

$2^{l-1} >= L$

$l >= \log_2 L + 1$

↓

min^m number of levels.

(5) In binary tree where every node has 0 or 2 children number of leaf nodes is always one more than nodes with two children.

$L = T + 1$

L → leaf node, T → No. of nodes with two children.

Assume height h :-

total nodes $= 2^h - 1$

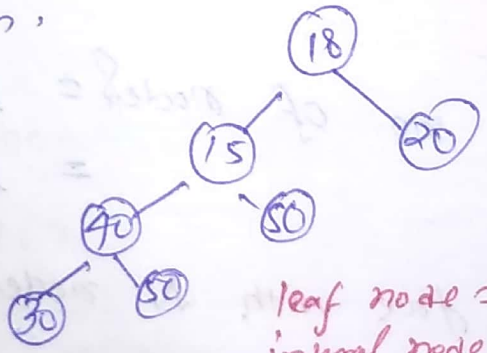
leaf nodes $= 2^{h-1}$

nodes with two child $= 2^{h-1} - 1$

Types of binary tree

i) Full Binary tree :- Binary tree is full if every node has 0 or 2 children.

(Means node full)



In full binary tree :-

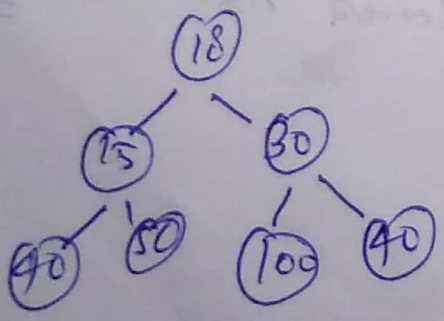
leaf node = internal node + 1

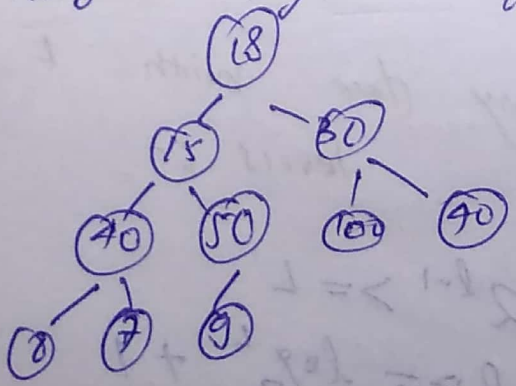
leaf node = 4 = 3 + 1

internal node = 3

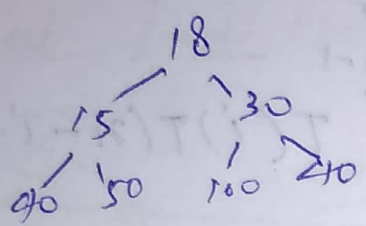
ii) Complete Binary Tree :-

if all levels are completely filled except last level and last level has all keys as left as possible.



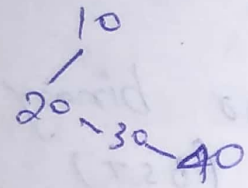


(3) **Perfect Binary tree:-** All internal nodes have two children and all leaf nodes at same level.



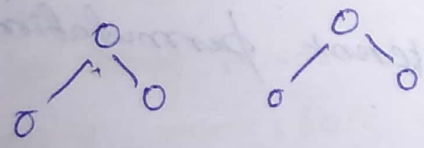
Perfect binary tree of h height have $2^h - 1$ nodes.

(4) **degenerate tree:-** where every internal node has one node child. Such trees are performance wise same as linked list.



∴ Binary tree is labeled if every node is assigned label.

two same unlabeled trees



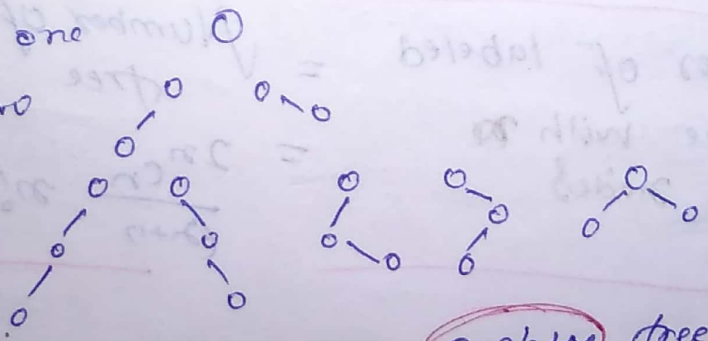
two different labeled trees

Number of different Unlabeled tree with n nodes

for n=1 → only one

for n=2 → two

n=3 → five



T(0) = 0 → There is only one empty tree.

T(1) = 1 → 5

T(2) = 2

T(3) = 5 = T(0) * T(2) + T(2) * T(0) + T(1) * T(1) = 5

T(4) = 14

$T(n)$ = C_n

$T(n)$ → no. of different Unlabeled free.

C_n → Catalan numbers

$$T(n) = \sum_{i=1}^{n} T(i-1)\,T(n-i) = \sum_{i=0}^{n-1} T(i)\,T(n-i-1)$$

$= T(0)T(n-1) + T(1)T(n-2) + \cdots + T(n-1)T(0)$

$T(i-1)$:- no. of nodes in left side

$T(n-i-1)$:- no. of " " right side.

$$C_n = \frac{2n!}{(n+1)!\,n!} = \frac{(2n C_n)}{(n+1)}$$ → Same for binary search tree (BST)

Number of ~~binary~~ labeled binary tree with n nodes

Every unlabeled tree with n nodes can create $n!$ different labeled tree by assigning different permutation of labels to all nodes.

Number of labeled tree with n nodes = Number of unlabeled tree $\times\ n!$

$$= \frac{2n C_n}{(n+1)}\,n! = \frac{2n!}{(n+1)\,n!}$$

Given tree

11

as le

node

new

tree un

is comp

using no

Struct N

int k

Node * l

Node (

{ key =

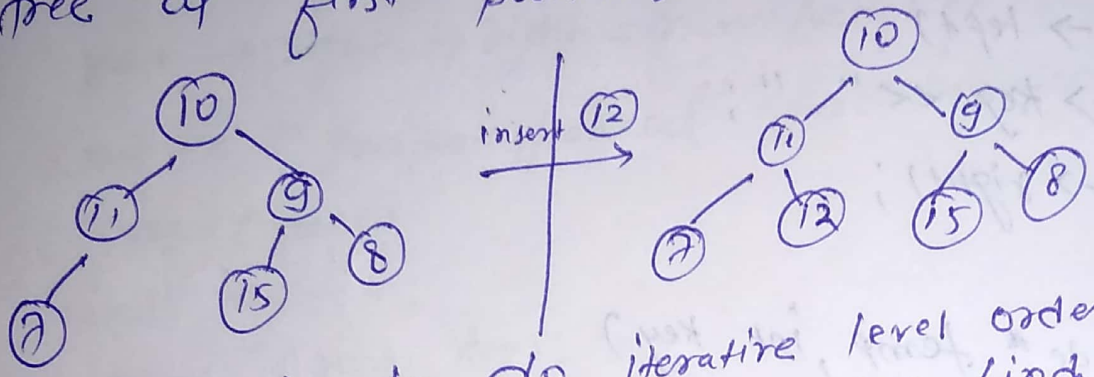
left =

3

3;

Insertion in a Binary Tree in level order

Given binary tree and key, insert key into binary tree at first position available in level order.

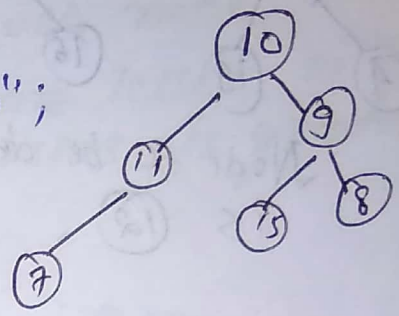


The idea is to do iterative level order traversal of given tree using queue. If we find a node whose left child is empty, we make new key as left child of node. Else if we find a node whose right child is empty, we make new key as right child. We keep traversing tree until we find a node whose left or right is empty.

```
#include <bits/stdc++.h>
using namespace std;
struct Node {
  int key;
  Node *left, *right;
  Node (int x)
  { key = x;
    left = right = NULL;
  }
};
```

```
Void inorder (Node * temp)
{ if (!temp)
      return;
  inorder (temp -> left);
  Cout << temp -> key << " ";
  inorder (temp -> right);
}
Void insert (Node * temp, int key)
{ queue < Node * > q;
  q.push (temp);
  While (! q.empty ())
  { Node * temp = q.front ();
    q.pop();
    if (! temp -> left)
      { temp -> left = new Node(key);
        break;
      }
    Else  q.push (temp -> left);
  if (! temp -> right)
    { temp -> right = new Node (key);
      break;
    }
  else q.push (temp -> right);
}
  int main ()
{ Node * root = new Node (10);
  root -> left = new Node (11);
```

```
root -> left -> left = new Node(7);
root -> right = new Node(9);
root -> right -> left = new Node(15);
root -> right -> right = new Node(8);
Cout << "inorder traversal before insertion";
inorder(root);
int key=12;
insert (root, key);
Cout<<endl;
Cout << "inorder traversal after insertion";
inorder(root);
return 0;
```

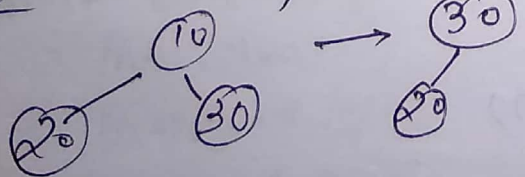


3 Output:- inorder traversal before insertion : 7 11 10 15 9 8
" " after " : 7 11 12 10 15 9 8

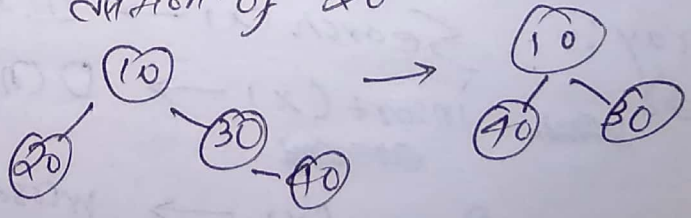
Deletion in Binary Tree.

This deletion is different from BST deletion. Here, we do not have any order among elements, so deleted element replace by last element. (bottommost ~~element~~ node and rightmost node)

Ex:- deletion of 10



deletion of 20

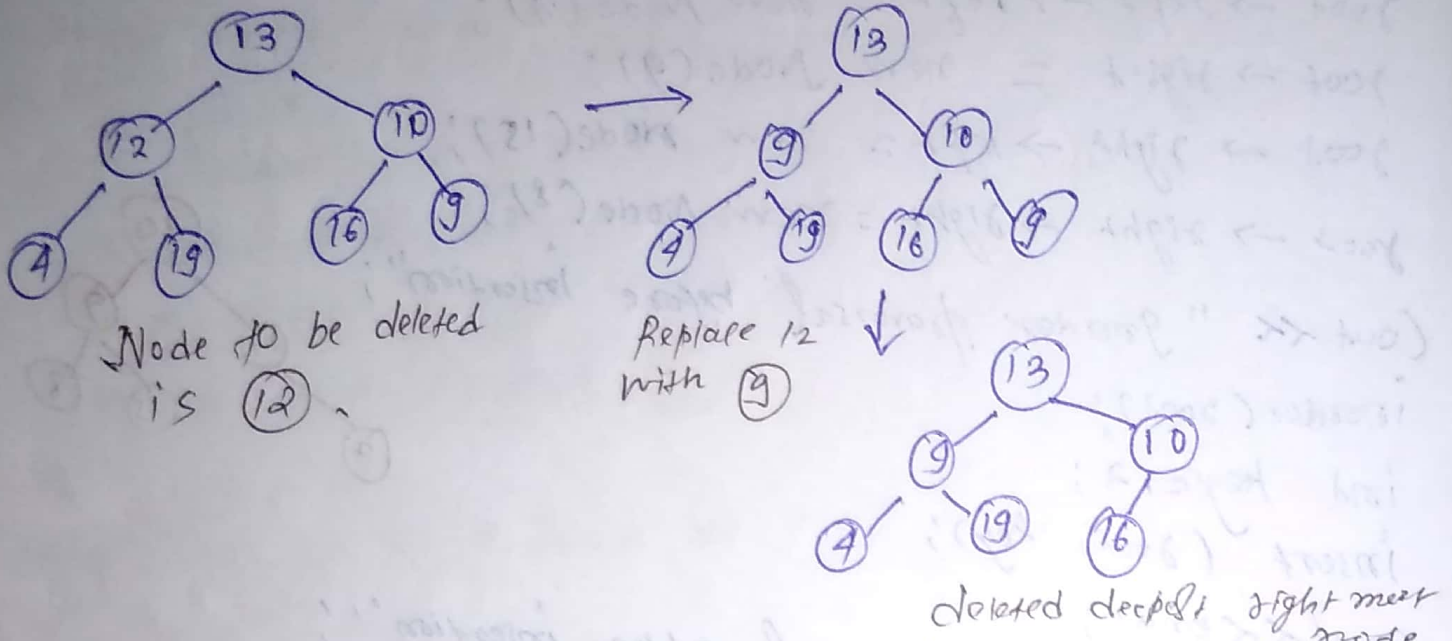


Algorithm:-

i) Starting from root, find deepest and rightmost node in binary tree and node which we want to delete.

ii) Replace deepest rightmost node's data with node to be delete.

iii) Delete deepest rightmost node.



Depth of x :-
No. of edges in path from root to x

Height of x :-
No. of edges in longest path from x to leaf.

Height of tree :- Height of root is equal to Height of tree.

Binary Search Tree

Quick Search, deletion, insertion.

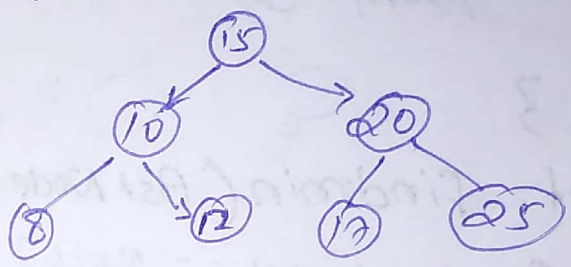
Array :- Search (x) → O(log n)
Unsorted insert (x) → O(n)
Remove (x) → worst O(n)

linked list → Search (x) → O(n)
unsorted insert (x) → O(1)
remove (x) → O(n)

Binary search tree (balanced) → search O(logn)
insert O(logn)
delete O(logn)

↳ A binary tree in which for each node, value of all nodes in left subtree are lesser or equal and value of all nodes in right subtree is greater.

binary search O(logn)



binary search tree

```
Struct BstNode {
    int data;
    BstNode * left;
    BstNode * right;
};  int main() {
BstNode* rootptr;
  rootptr = NULL;
  insert (root, 15);
```

BstNode* Get

Find min and max element in a BST

```
Struct BstNode {
    int data;
    BstNode* left;
    BstNode* right;
};
```

```
int Findmin( BstNode* root)
{ BstNode * Current = root;
  if (root == NULL)
      return -1;
  while (current -> left != NULL)
     { Current = Current -> left; }
  return Current -> data;
}
```

Iterative

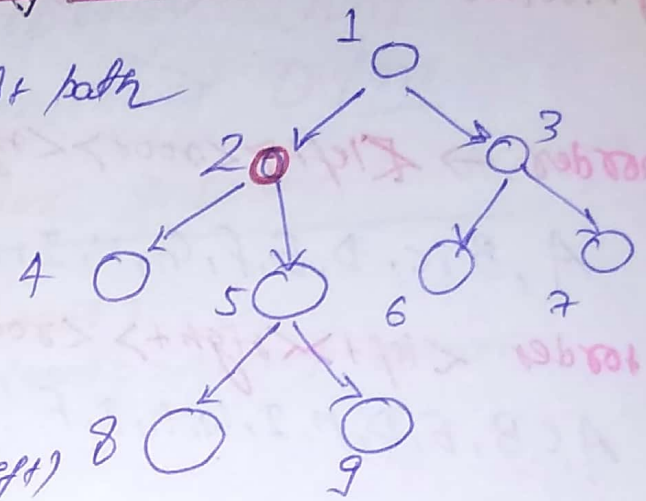
```
int Findmin( BstNode *root)
{ if ( root == NULL)
   { return -1; }
  Else if ( root -> left == NULL)
        return root -> data;
  Else return Findmin(root -> left);
}
```

Recursive

```
int Findmax ( BstNode * root)
{ BstNode * Current = root;
  while ( current -> right != NULL)
        { current = current -> right ; }
  return current -> data;
}
```

Find height of Binary tree

→ no. of edges in longest path from root to leaf.



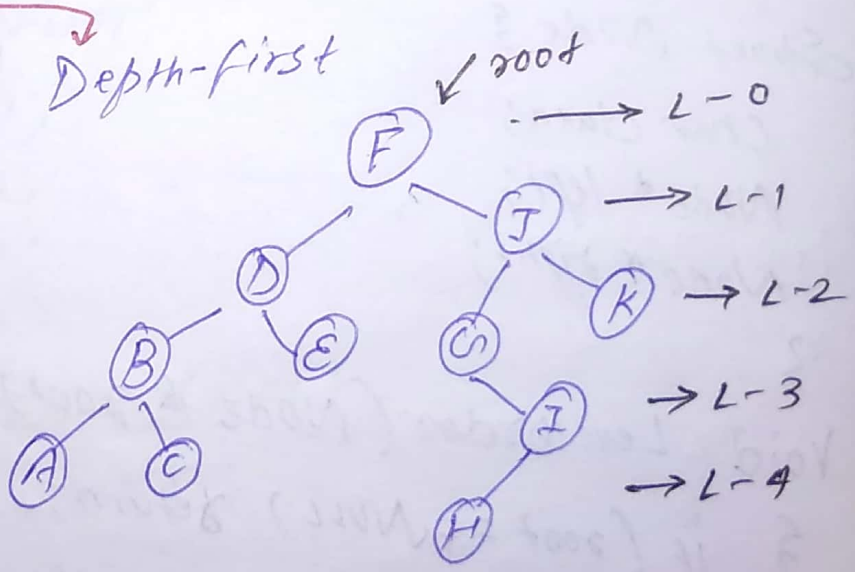

```
Find Height (root)
{  if (root == NULL)
       return -1;
   left height = Find Height (root -> left)
   Right height = Find Height (root -> right)
   return max (left height, right height) + 1;
```

3 Binary tree traversal

→ Breadth-first (Level order)

F, D, J, B, E, S, K, A, C, I, H

→ Depth-first



Depth-first

i) <root> <left> <right> → preorder

ii) <left> <root> <right> → inorder

iii) <left> <right> <root> → postorder

Preorder → <root><left><right>

Inorder → <left><root><right>

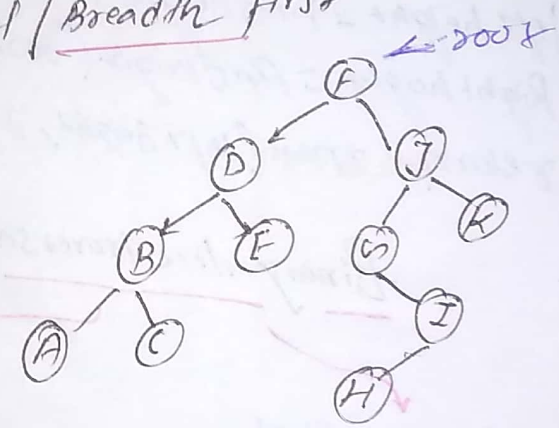
A, B, C, D, E, F, G, H, I, J, K

Postorder <left><right><root>

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

level-order traversal / Breadth first

F D J B E G K A C I H




```
#include<iostream>
Using namespace Std;
Struct Node {
   char data;
   Node * left;
   Node * right;
}
Void LevelOrder(Node * root)
{ if (root == NULL) return;
   queue<Node*> Q;
   Q.push(root);
   while (!Q.empty())
   { Node * current = Q.front();
   Cout << current -> data << " ";
   if (current -> left != NULL) Q.push(current -> left);
   if (current -> right != NULL) Q.push(current -> right);
```

}
Void
{ Co
Pr
Pr
}
Void
{ if
gr
co
gr
}

Q.pop();

}

}

Time-complexity O(n)

Space-Complexity O(n)

Depth-first (preorder, inorder, postorder)

```
Void preorder(struct Node *root)
{ Cout << root->data;
  preorder(root->left);
  preorder(root->right);
}
```

if (root==NULL) return;

Extra space O(h) → height of tree for stack function call.

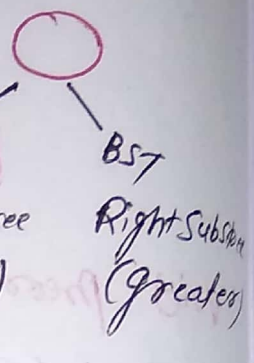
```
Void inorder(struct Node *root)
{ if (root==NULL) return;
  inorder(root->left);
  cout << root->data;
  inorder(root->right);
}
Void postorder(struct Node *root)
{ if (root==NULL) return;
  inorder(root->left);
  inorder(root->right);
  Cout<< root->data;
}
```

Time complexity O(n)

Space " O(h)

Check if a given binary tree is BST

BST :- binary tree in which for each node value of all node in leftsubtree should be less or equal and value of all node in rightsubtree should be greater.




```
bool IsBinarySearchTree(Node *root)
{ if(root==NULL) return True;
  if(IssubtreeLesser(root->left, root->data) &&
     Issubtree Greater(root->right, root->data) &&
     IsBinarySearchTree(root->left) && IsBinarySearchTree
     (root->right))
          return true;
  else    return false;
}
bool IssubtreeLesser(Node *root, int value)
{   if(root==NULL) return true;
  if(root->data<=value && IssubtreeLesser(root->left, value)
     && IssubtreeLesser(root->right, value))
          return true;
  else    return false;
}
bool Issubtreegreater(Node *root, int value)
{ if(root==NULL return true;
  if(root->data>value && issubtreegreater(root->left, value)
     && Issubtreegreater(root->right, value))
          return true;
  else    return false; }
```

method-2

```
· IsBstUtil (Node *root, int minValue, int maxValue)
{ if (root == NULL) return true;
  if (root -> data > minvalue && root -> data < maxvalue
     && IsBSTUtil(root->left, minvalue, root -> data)
  && IsBSTUtil (root -> right, root->data, maxvalue)
      return true;
  else return false;
}
bool IsBinarysearchTree (Node* root)
{ return IsBSTUtil (root, INT_MIN, INT_MAX);
}
```

Delete a node from BST

```
Struct Node* Delete (struct Node *root, int data)
{   if (root == NULL) return root;
    else if (data < root -> data)
            root->left = Delete (root -> left, data);
    else if (data > root -> data)
            root -> right = Delete(root -> right, data);
    else
    { if (root -> left == NULL && root -> right == NULL)
      { delete root;
        root = NULL;
        return root;
      }
```

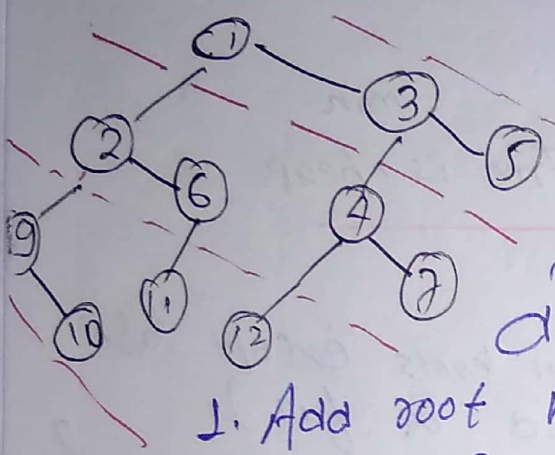
```
else if (root->left == NULL)
  { struct Node *temp = root;
    root = root->right;
    delete temp;
    return root;
  }
else if (root -> right == NULL)
    { struct Node *temp = root;
      root = root -> left;
      delete temp;
      return root;
    }
Else {
    struct Node *temp = find min (root -> right);
    root -> data = temp -> data;
    root -> right = delete (root -> right, root -> data);
    return root;
    }

} Node * findmin (Node * root)
  { if (root -> left == NULL)
      return root;

    else
      return find min (root -> left);

  }
```

Inorder Successor in BST

```
Struct Node * GetSuccessor (Struct Node * root, int data)
{ Struct Node * current = Find (root, data);
  if (current == NULL) return NULL;
// case I : Node has right subtree
 if (current -> right != NULL)
    { Struct Node * temp = current -> right;
    while (temp -> left != NULL) temp = temp -> left;
     return temp;
    }
Else
{ Struct Node * sucessor = NULL;
  Struct Node * ancestor = root;
  while (ancestor != current)
    { if (current -> data < ancesstor -> data)
        { successor = ancestor;
          ancestor = ancestor -> left;
        }
      else
        ancestor = ancestor -> right;
    }
  return sucessor;
}
}
```

```
root -> right = ConvertExpression(str, i);
return root;
}
Else return root;
}
```

Diagonal Sum of Binary Tree



Algorithm:-

idea is to keep track of vertical distance from top diagonal passing through root. We increment vertical distance we go down to next diagonal.

1. Add root with vertical distance as 0 to queue
2. Process the sum of all right child and right of right child and so on.
3. Add left child current node into queue for later preprocessing. Vertical distance of left child is vertical distance of current node plus 1.
4. keep doing 2nd, 3rd and 4th step till queue is empty.

```
Vector<int> Diagonal_sum(Node* root)
{ Vector<int> v; map<int, int> mp;
  int x=0;
  Solve(root, mp, x);
  for(auto it = mp.begin(); it != mp.end(); it++)
      v.push_back(it->second);
  return v;
}
```

```
void Solve(Node* root, map<int,int>& mp, int x)
{
    if(root==NULL)
        return;
    mp[x]+= root->data;
    Solve(root->left, mp, x+1);
    Solve(root->right, mp, x);
}
```

Check if given binary Tree is max heap

For max heap two condition satisfy:-

i) It should be Complete tree - (i.e all levels except last should be full).

ii) Every node's value should be greater than or equal to its child node.

```
int CountNodes(struct Node* root)
{
    if(root==NULL)
        return 0;
    return 1+CountNodes(root->left)+CountNodes(root->right);
}
bool isCompleteUtil(Node* root, int index, int number_nodes)
{
    if(root==NULL)
        return true;
    if(index >= number_nodes)
        return false;
```

```
    return (isCompleteUtil(root->left, 2*index+1, number_nodes)
        && isCompleteUtil(root->right, 2*index+2, number_nodes));
}
bool isHeapUtil(Node* root)
{
    if (root->left == NULL && root->right == NULL)
        return true;
    if (root->right == NULL)
    {
        return (root->data >= root->left->data);
    }
    else {
        if (root->key >= root->left->key &&
            root->key >= root->right->key)
            return ((isHeapUtil(root->left)) &&
                    (isHeapUtil(root->right)));
        else
            return false;
    }
}
bool isHeap(struct Node* root)
{
    int count = countNodes(root);
    if (isCompleteUtil(root, 0, count) &&
        isHeapUtil(root))
        return true;
    return false;
}
```

3

DP On Trees

→ General Syntax

→ How DP can be applied to trees

→ Diameters of binary trees

→ Maximum path Sum from any node to any node

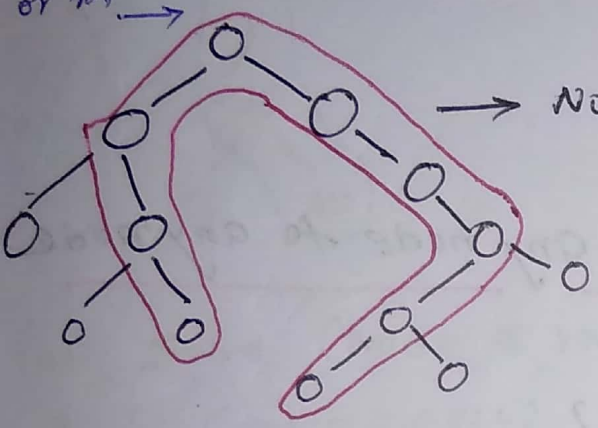
→ maximum path Sum from leaf to leaf

→ diameter of N-array trees.

Diameter of tree :- longest distance / path b/w two leaf.

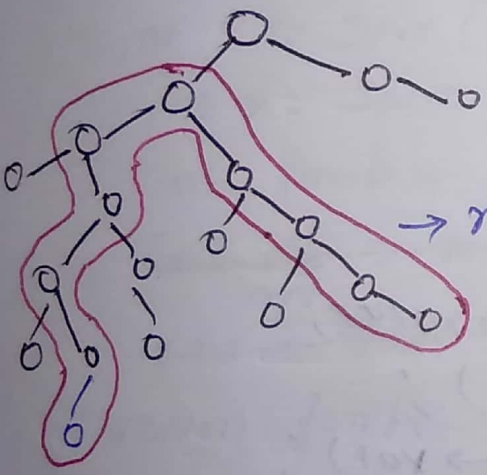
Roots included or not →

Diameter of binary trees



→ No. of Nodes in path = 9

O/P = 9



→ no. of nodes in path = 10.

O/P = 10

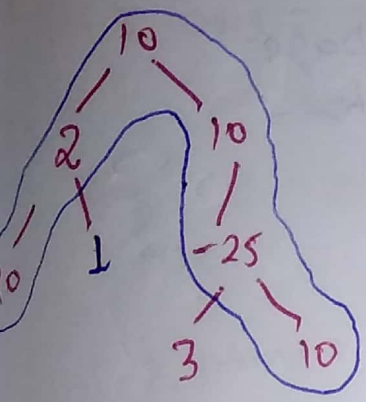
```
int Solve (Node * root, int *res)
{   if (root == NULL)
        return 0;
    int l = solve (root -> left, &res);
    int r = Solve (root -> right, &res);

int temp = max (l, r) + 1;
int ans = ~~max(temp~~, 1 + l + r);
res = max(res, ans);
return temp;
}
```

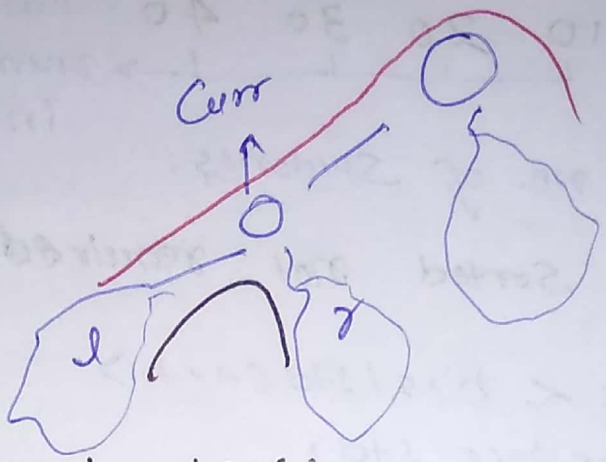
```
int main ()
{
  int res = INT_MIN;
  Solve (root, &res);
  Cout<< res <<endl;
}
```

Maximum path sum from any node to any node

```
int Solve (Node * root, int *res)
{ if (root == NULL)
        return 0;
  int l = Solve (root->left, &res);
  int r = Solve (root->right, &res);
  int temp = max( max(l, r) + root->value,
                  root->value );
  int ans = max(temp, l+r+root->val);
  res = max(res, ans);
  return temp;
}
    int main ()
  { int res = INT_MIN;
    Solve (root, &res);
    Cout<< res <<endl;
  }
```

maximum path sum from leafs to leafs






```
int Solve (Node * root, int * res)
{ if (root == NULL)
      return 0;
  int l = Solve (root -> left, &res);
  int r = Solve (root -> right, &res);
  int temp = max( l, r) + root -> value;
  int ans = max( temp, l+r+root -> value);
  res = max(res, ans);
  return temp;
}
int main()
{ int res = INT_MIN;
  Solve (root, &res);
  Cout << res << endl;
}
```

```
if (root -> left && root -> right)
    res = max(res, l+r+root -> value);
```