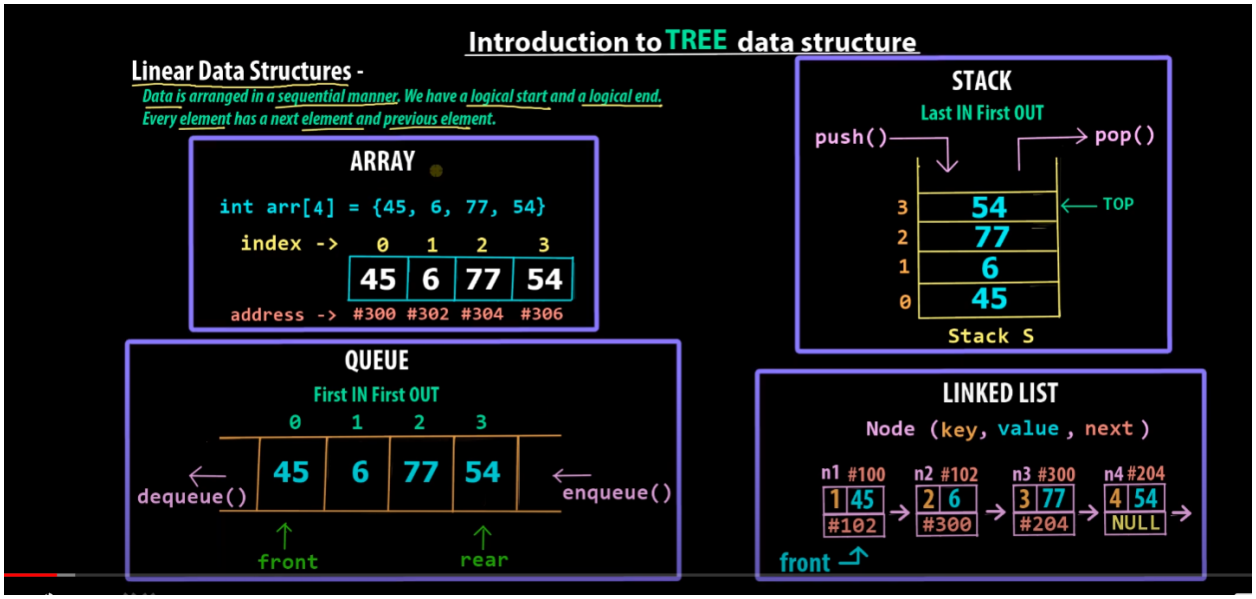
Tree

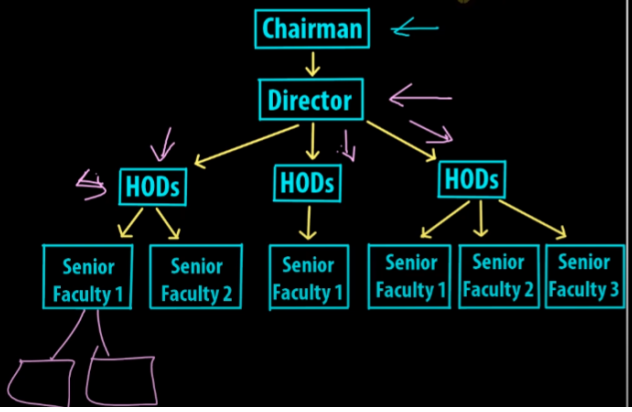

Introduction to TREE data structure
What is a Data Structure -
DS is way of storing/organising data in the memory, in such a way that access, management & modification becomes efficient.
Which DS to use -
1) What type of data needs to be stored
2) Cost of operations (read, write, update etc)
3) Memory usage & efficiency
4) Easy of implementation
5) Maintainability
* Every Data Structure has its own PROS & CONS !


Introduction to TREE data structure
Linear Data Structures -
Data is arranged in a sequential manner. We have a logical start and a logical end.
Every element has a next element and previous element.
ARRAY
int arr[4] = {45, 6, 77, 54}
index -> 0 1 2 3
45 6 77 54
address -> #300 #302 #304 #306
STACK
Last IN First OUT
push()
pop()
TOP
3 54
2 77
1 6
0 45
Stack S
QUEUE
First IN First OUT
0 1 2 3
45 6 77 54
dequeue()
enqueue()
front
rear
LINKED LIST
Node (key, value , next)
n1 #100 n2 #102 n3 #300 n4 #204
1 45 #102
2 6 #300
3 77 #204
4 54 NULL
front

Introduction to TREE data structure

TREE - *A tree is a non linear data structure that simulates a hierarchial tree structure with a root value & subtrees of children with parent node, represented as set of linked nodes.*

Real world example of hierarchial tree

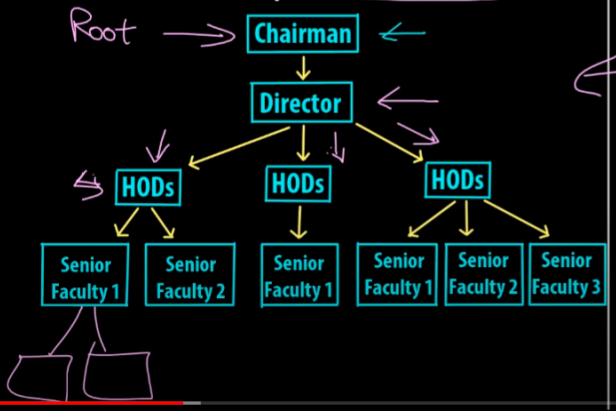


Company Hierarchy -

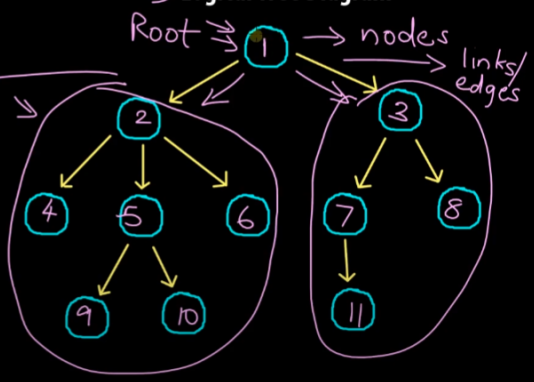
Introduction to TREE data structure

TREE - *A tree is a non linear data structure that simulates a hierarchial tree structure with a root value & subtrees of children with parent node, represented as set of linked nodes.*

Real world example of hierarchial tree



Logical Tree Diagram



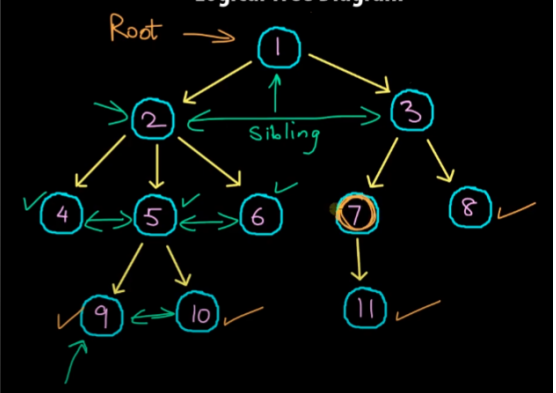
Introduction to TREE data structure

TREE - *A tree is a non linear data structure that simulates a hierarchial tree structure with a root value & subtrees of children with parent node, represented as set of linked nodes.*

Important TREE terms -

1. **Root** - Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent.
2. **Parent Node** - Parent node is an immediate predecessor of a node.
3. **Child Node** - All immediate successors of a node are its children.
4. **Siblings** - Nodes with the same parent are called Siblings.
5. **Leaf** - Last node in the tree. There is no node after this node.
6. **Edge** - Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.
7. **Path** - Path is a number of successive edges from source node to destination node.
8. **Degree of Node** - Degree of a node is equal to number of children, a node have.

Logical Tree Diagram



Introduction to TREE data structure

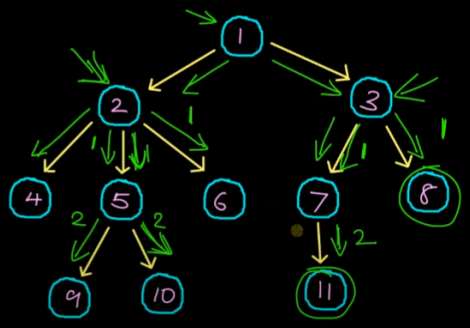
TREE - *A tree is a non linear data structure that simulates a hierarchial tree structure with a root value & subtrees of children with parent node, represented as set of linked nodes.*

Important TREE terms / properties -

1) *Tree can be termed as a RECURSIVE data structure.*
2) *In a valid tree for N Nodes we have N-1 Edges/Links.*
3) **Depth of Node** - Depth of a node represents the number of edges from the tree's root node to the node.
4) **Height of Node** - Height of a node is the number of edges on the longest path between that node & a leaf.
5) **Height of Tree** - Height of tree is the height of its root node.

Depth (6) = 2 Height (3) = 2
Depth (9) = 3 Height (2) = 2

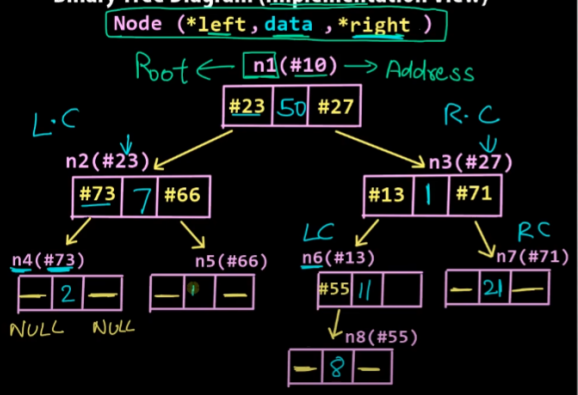
Logical Tree Diagram



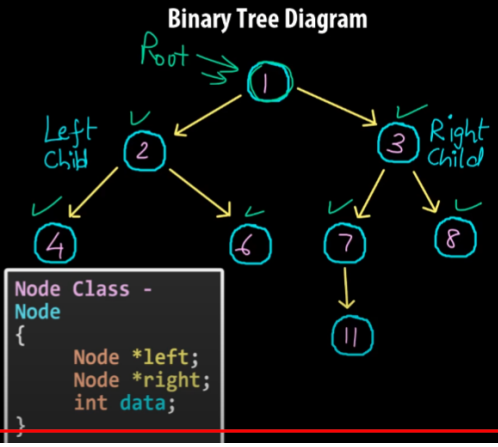
Introduction to TREE data structure

TREE - *A tree is a non linear data structure that simulates a hierarchial tree structure with a root value & subtrees of children with parent node, represented as set of linked nodes.*

Binary Tree Diagram (Implementation View)



Binary Tree Diagram




```
Node Class -
Node
{
    Node *left;
    Node *right;
    int data;
}
```

Introduction to TREE data structure

TREE - *A tree is a non linear data structure that simulates a hierarchial tree structure with a root value & subtrees of children with parent node, represented as set of linked nodes.*

Types of Trees -

1. General Tree
2. Binary Tree
3. Binary Search Tree
4. AVL Tree
5. Spanning Tree
6. B-Tree
7. B+ Tree
8. Heap

Applications of Tree Data Structure -

1. Store hierarchical data, like folder structure, organization structure data.
2. Binary Search Tree is a tree that allows fast search, insert, delete on a sorted data.
3. Heap is a tree data structure which is implemented using arrays and used to implement priority queues.
4. B-Tree and B+ Tree are used to implement indexing in databases.
5. Used to store router-tables in routers.
6. Used by compilers to build syntax trees.
7. Used to implement expression parsers and expression solvers.

Binary Tree

Binary Tree Data Structure

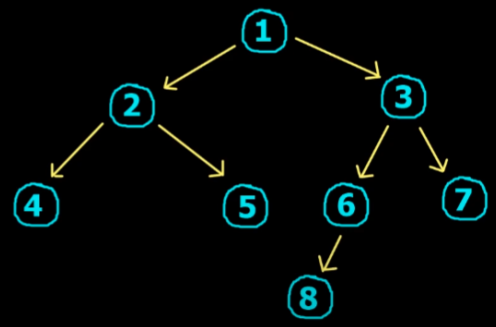
Binary Tree - *A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child (LC) and the right child (RC).*

Important Binary Tree Terms & Properties -

1) A binary tree is called **STRICT/PROPER** binary tree, when each node has 2 or 0 children.
2) A binary tree is called **COMPLETE** binary tree if all levels except the last are completely filled and all nodes are as left as possible.
3) A binary tree is called **PERFECT** binary tree of all levels are completely filled with 2 children each.
4) Max number of nodes at level 'x' = 2^x
5) For a binary tree, maximum no of nodes with height 'h' = $2^0 + 2^1 + ... + 2^h$

$= 2^{(h+1)} - 1$

Binary Tree Diagram



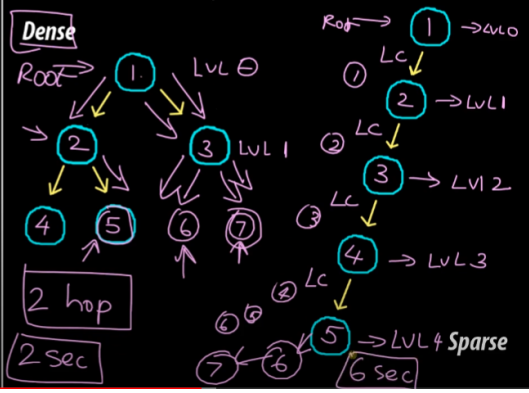
Binary Tree Data Structure

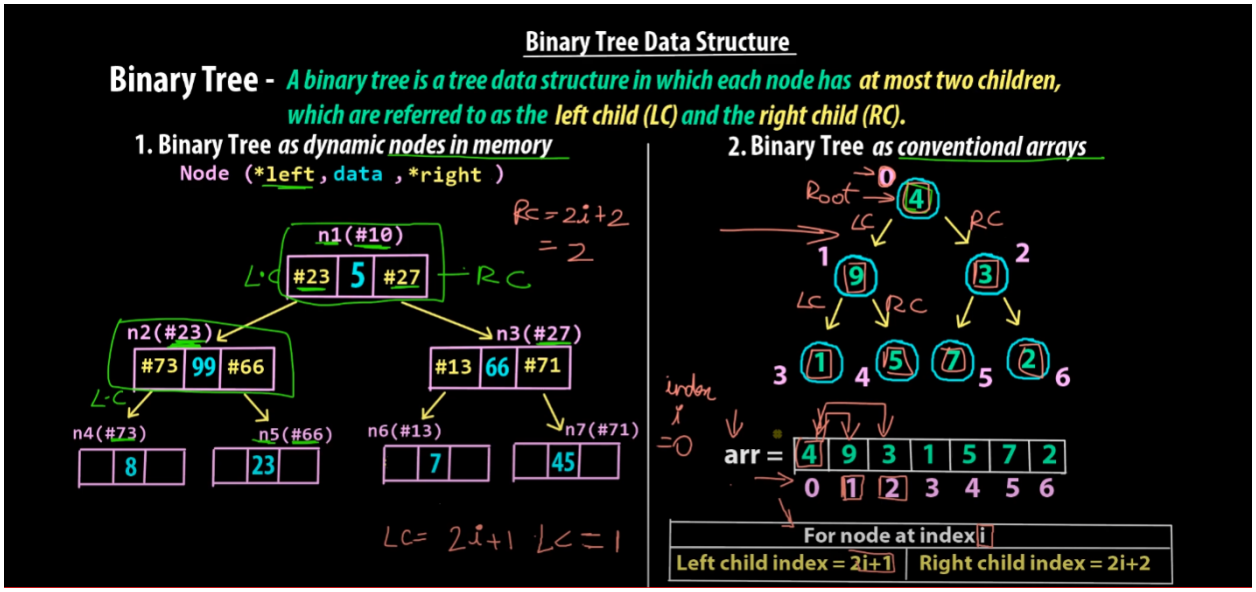
Binary Tree - *A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child (LC) and the right child (RC).*

Important Binary Tree Terms & Properties -

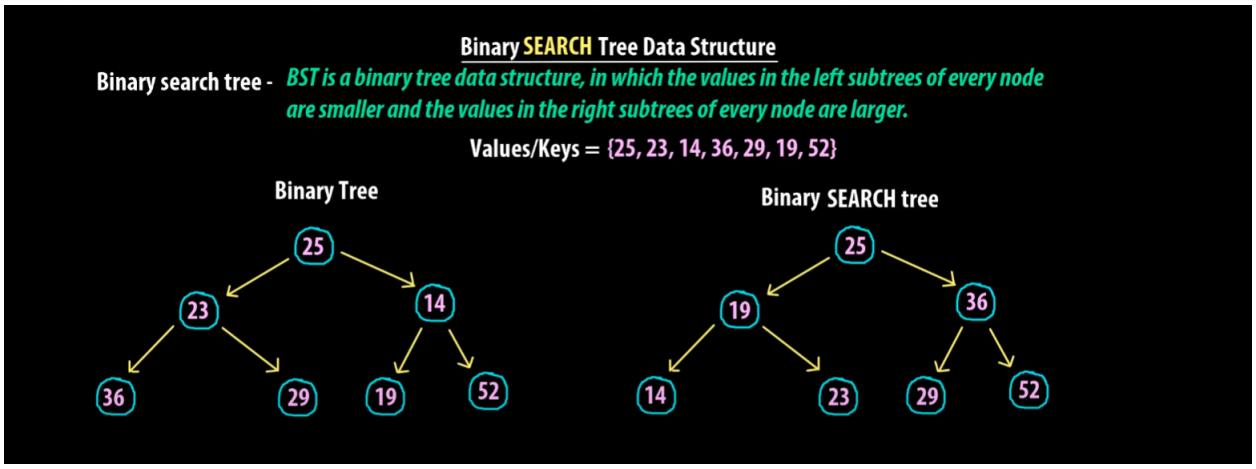
6) A binary tree is called BALANCED binary tree, if the difference between the height of left and right subtree for every node is not more than k (ususally 1)

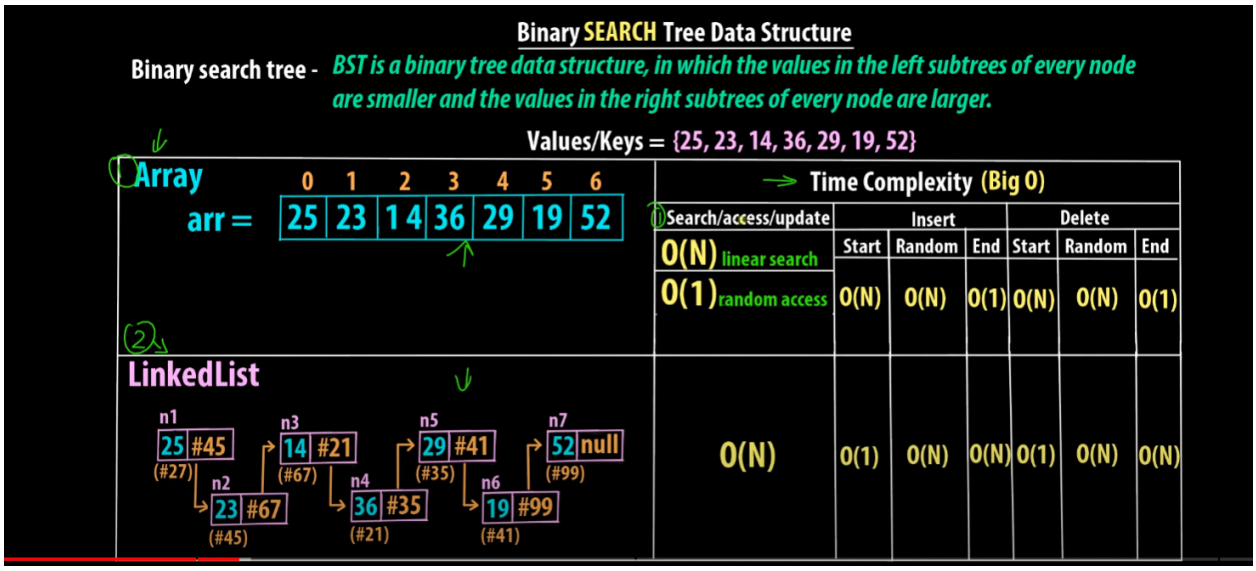
Binary Tree Variations





Binary Search Tree





Binary SEARCH Tree Data Structure

Binary search tree - *BST is a binary tree data structure, in which the values in the left subtrees of every node are smaller and the values in the right subtrees of every node are larger.*

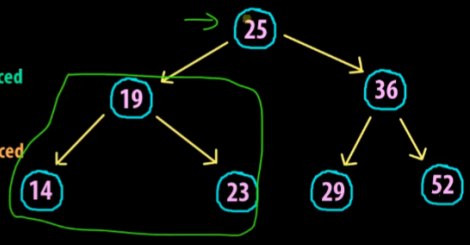
→ Values/Keys = {25, 23, 14, 36, 29, 19, 52}

	Time Complexity (Big O)		
	Search/Access/Update	Insert/Add	Delete/Remove
Avg case	O(log N)	O(log N)	O(log N)
Worst case	O(N)	O(N)	O(N)

balanced

not balanced

Binary SEARCH tree balanced



	0	1	2	3	4	5	6
sorted_arr =	14	19	23	25	29	36	52

Binary SEARCH Tree Data Structure

Binary search tree - *BST is a binary tree data structure, in which the values in the left subtrees of every node are smaller and the values in the right subtrees of every node are larger.*

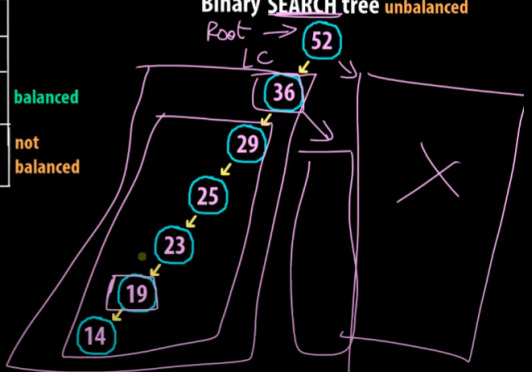
Values/Keys = {25, 23, 14, 36, 29, 19, 52}

	Time Complexity (Big O)		
	Search/Access/Update	Insert/Add	Delete/Remove
Avg case	O(log N)	O(log N)	O(log N)
Worst case	O(N)	O(N)	O(N)

balanced

not balanced

Binary SEARCH tree unbalanced



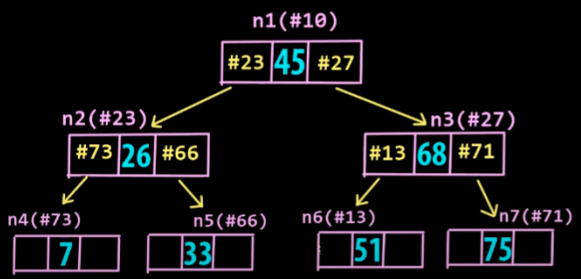
	0	1	2	3	4	5	6
sorted_arr =	14	19	23	25	29	36	52

Implementation

Binary SEARCH Tree Data Structure (Implementation)

1. Binary *SEARCH* Tree *as dynamic nodes in memory*

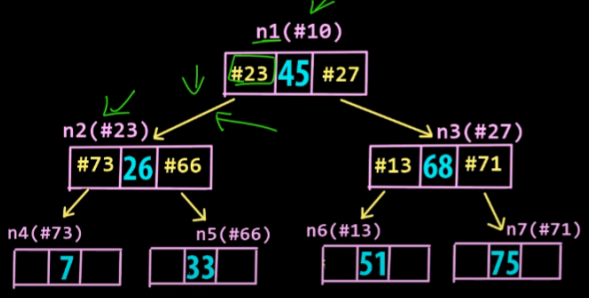
```
Node (*left, data ,*right )
```



Binary SEARCH Tree Data Structure (Implementation)

1. Binary *SEARCH* Tree *as dynamic nodes in memory*

Node (*left, data, *right)



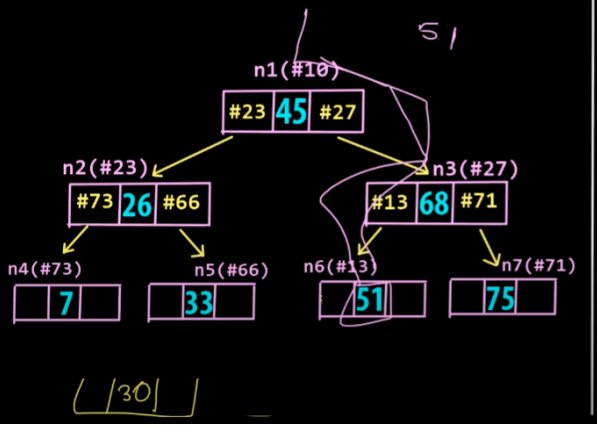
Tree Node Class -

```
class TreeNode
{
 Data Members -
  1.int value
  2.TreeNode * left   // pointers in C++
  3.TreeNode * right  // pointers in C++
 Member Functions -
  1.TreeNode()        // default constructor
  2.TreeNode(int v)   // parameterized constructor
}
```

Binary SEARCH Tree Data Structure (Implementation)

1. Binary *SEARCH* Tree *as dynamic nodes in memory*

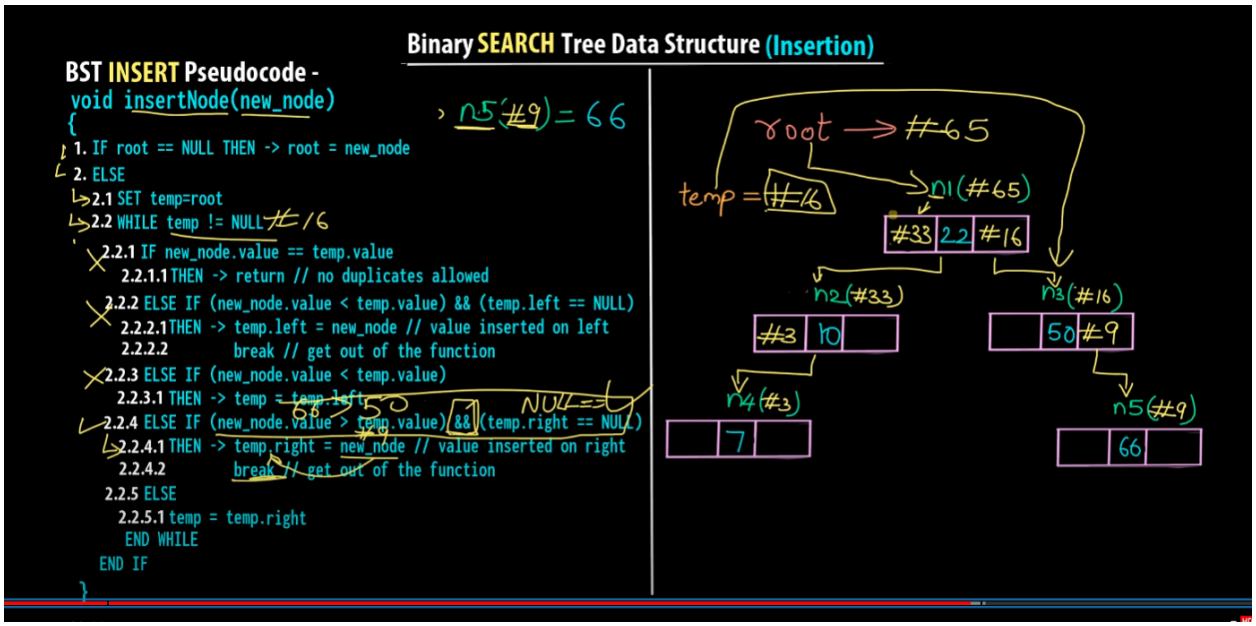
Node (*left, data, *right)



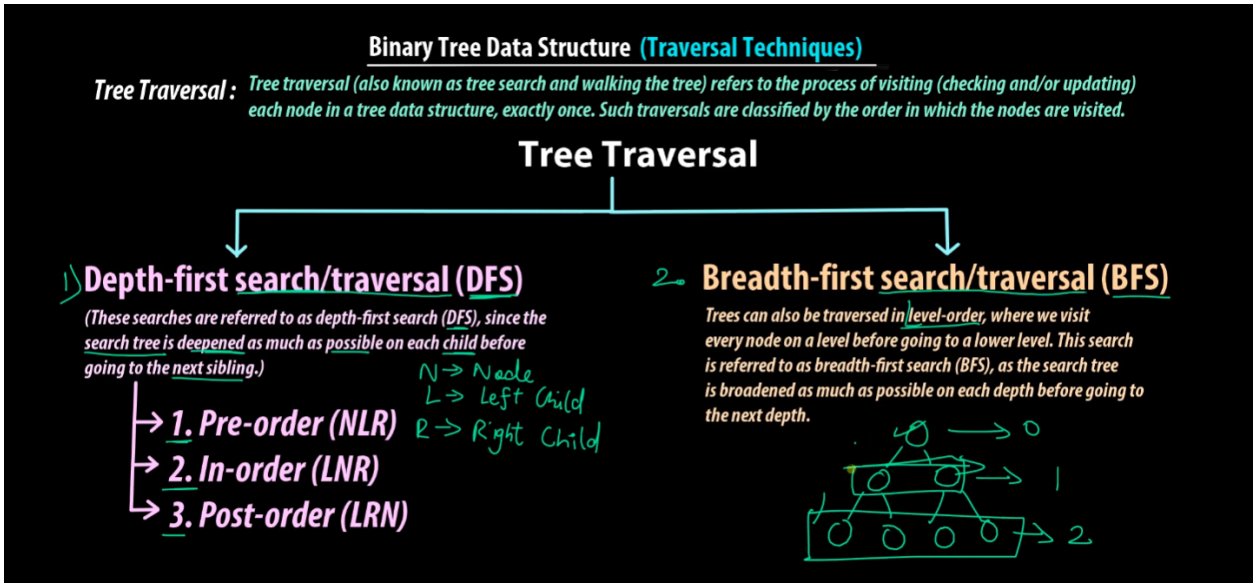
BST Class -

```
class BST
{
 Data Members -
   1.TreeNode *root
 Member Functions -
  1.bool isTreeEmpty() // check if empty
  2.void insertNode(TreeNode *new_node)//insertion
  // depth first traversal approach
  3a.void printPreorder(TreeNode* r)//print & traverse
  3b.void printInorder(TreeNode* r) //print & traverse
  3c.void printPostorder(TreeNode* r)//print & traverse
  3d.void print2D(TreeNode *r,int space)//print & traverse
  // breadth first traversal approach
  3e.void printLevelOrder(TreeNode *r)//print & traverse

  4.TreeNode* search(TreeNode* r, int v)// search

  5a.TreeNode* minValueNode(TreeNode* node)//delete
  5b.TreeNode* deleteNode(TreeNode* r, int v)//delete
}
```

BST INSERTION



Tree Traversal

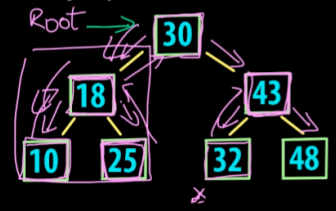


Binary Tree Data Structure (Traversal Techniques)

→ *Depth-first search/Traversal of binary tree (DFS)* – *These searches are referred to as depth-first search (DFS), since the search tree is deepened as much as possible on each child before going to the next sibling.*

1. Pre-order (NLR) *(node - left - right)*

1) - Access the data part of the current node.
2) - Traverse the left subtree by recursively calling the pre-order function.
3) - Traverse the right subtree by recursively calling the pre-order function.

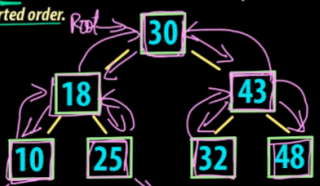


30, 18, 10, 25, 43, 32, 48

2. In-order (LNR) *(left - node - right)*

- Traverse the left subtree by recursively calling the in-order function.
- Access the data part of the current node.
- Traverse the right subtree by recursively calling the in-order function.

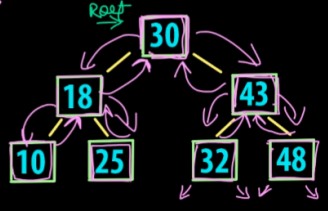
In BST in-order traversal retrieves the keys in ascending sorted order.



10, 18, 25, 30, 32, 43, 48

3. Post-order (LRN) *(left - right - node)*

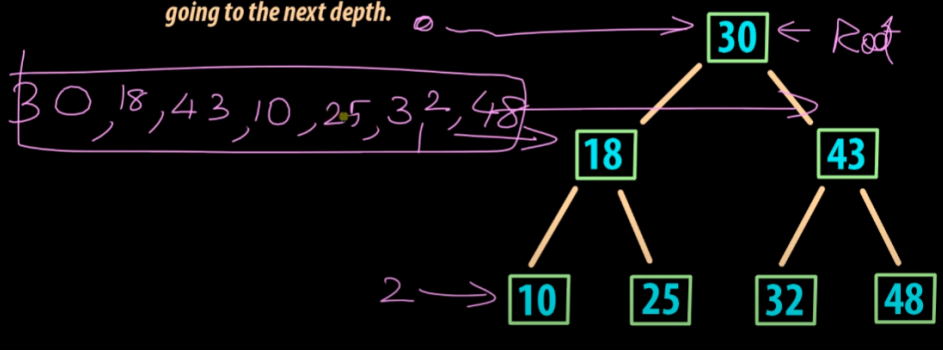
- Traverse the left subtree by recursively calling the post-order function.
- Traverse the right subtree by recursively calling the post-order function.
- Access the data part of the current node.



10, 25, 18, 32, 48, 43, 30

Binary Tree Data Structure (Traversal Techniques)

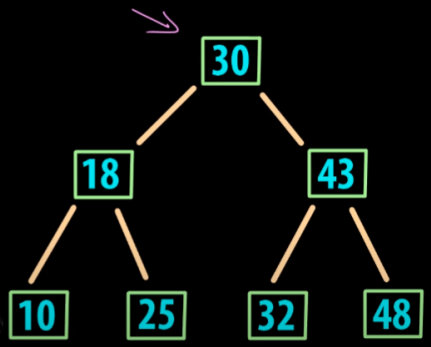
Breadth-first search / level order (BFS) – *Trees can also be traversed in level-order, where we visit every node on a level before going to a lower level. This search is referred to as breadth-first search (BFS), as the search tree is broadened as much as possible on each depth before going to the next depth.*



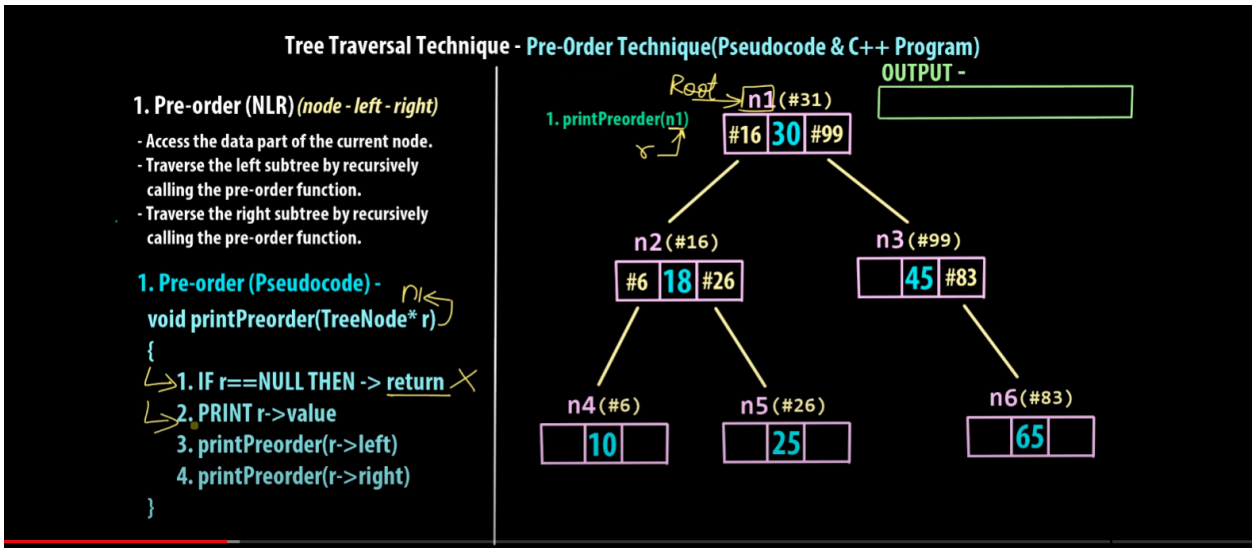
Binary Tree Data Structure (Traversal Techniques)

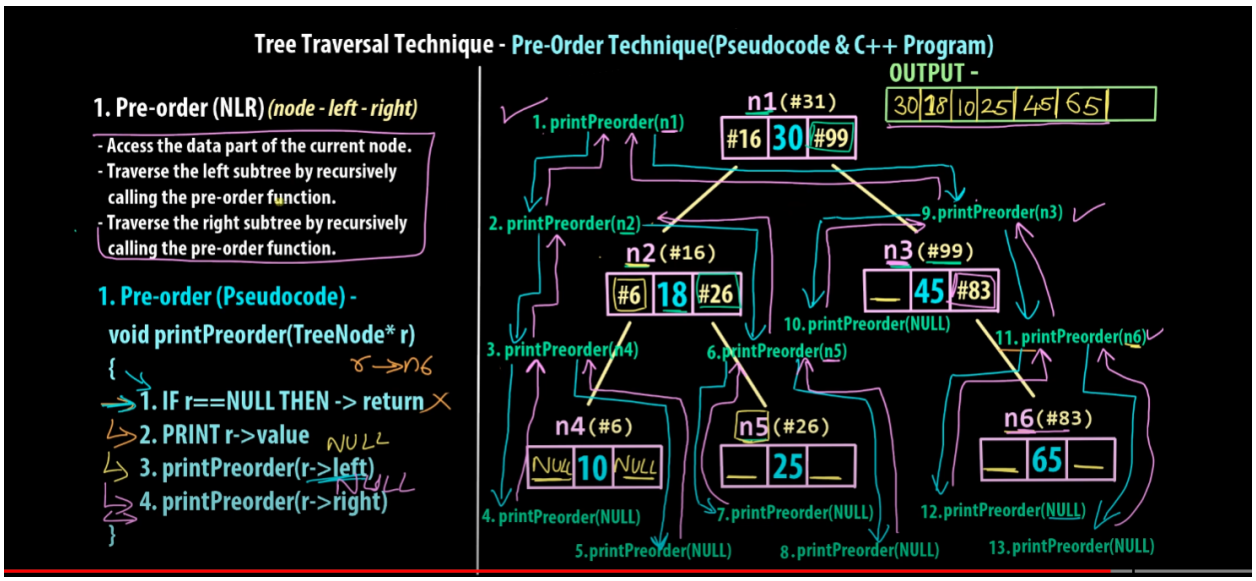
Tree Traversal : *Tree traversal (also known as tree search and walking the tree) refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.*

DFS -	
1. Pre-order (NLR) -	30, 18, 10, 25, 43, 32, 48
2. In-order (LNR) -	10, 18, 25, 30, 32, 43, 48
3. Post-order (LRN) -	10, 25, 18, 32, 48, 43, 30
BFS -	30, 18, 43, 10, 25, 32, 48

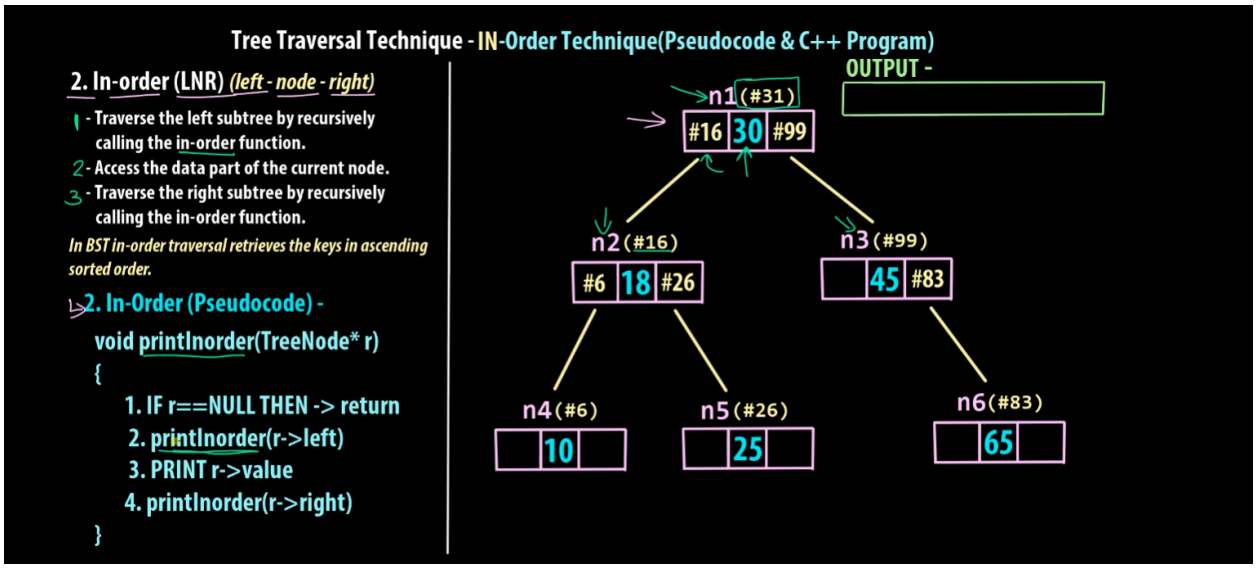


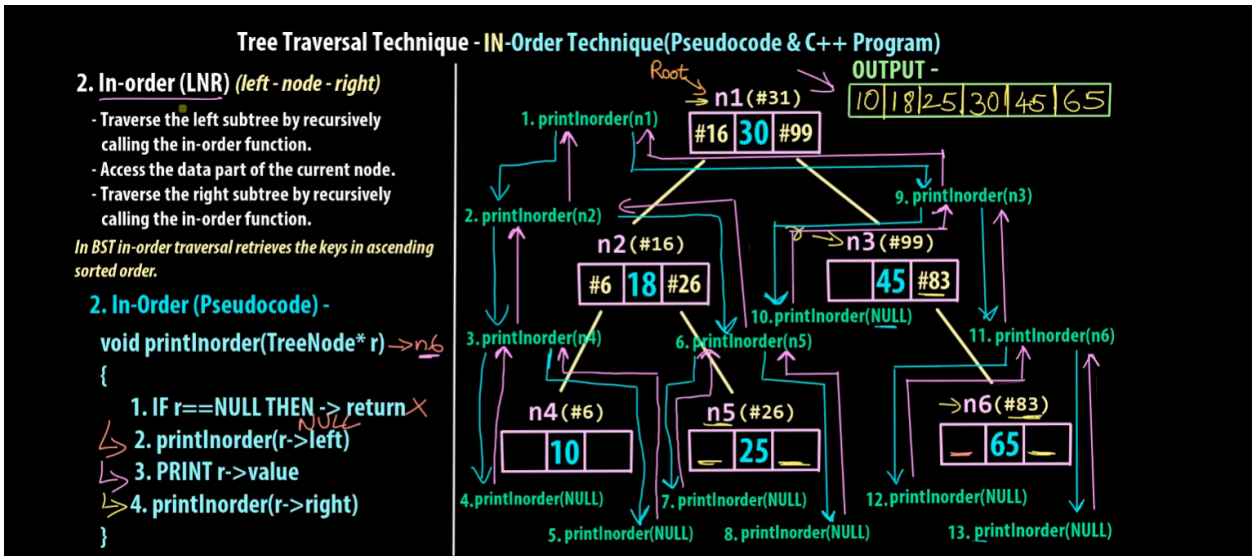
PreOrder


Tree Traversal Technique - Pre-Order Technique(Pseudocode & C++ Program)
1. Pre-order (NLR) (node - left - right)
- Access the data part of the current node.
- Traverse the left subtree by recursively calling the pre-order function.
- Traverse the right subtree by recursively calling the pre-order function.
1. Pre-order (Pseudocode) -
void printPreorder(TreeNode* r)
{
1. IF r==NULL THEN -> return
2. PRINT r->value
3. printPreorder(r->left)
4. printPreorder(r->right)
}
OUTPUT -
Root
n1 (#31)
#16 30 #99
1. printPreorder(n1)
n2 (#16)
#6 18 #26
n3 (#99)
45 #83
n4 (#6)
10
n5 (#26)
25
n6 (#83)
65

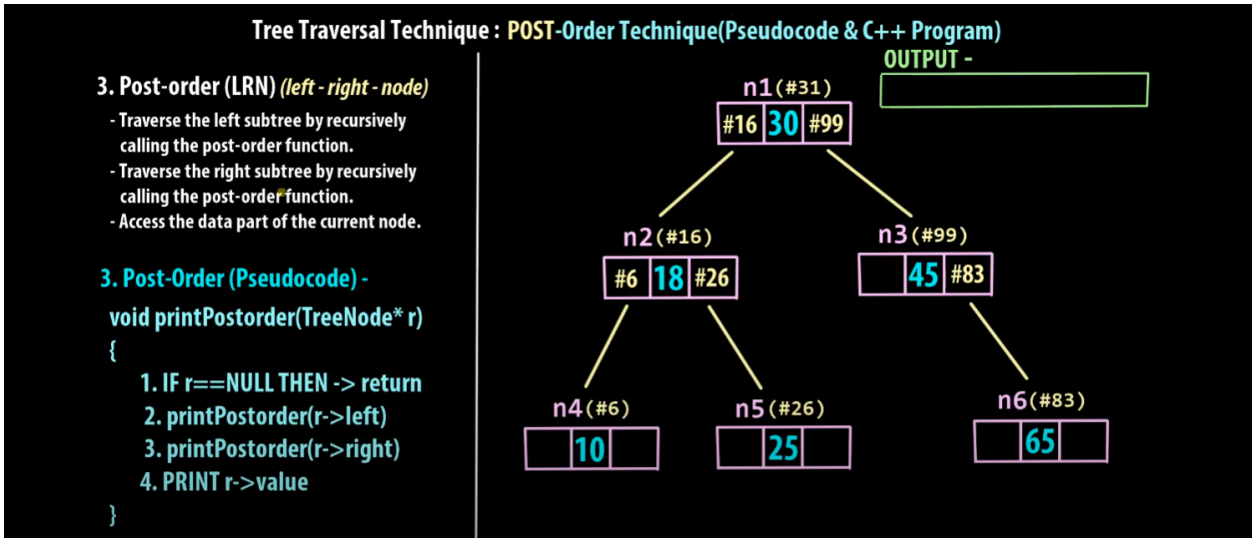

Tree Traversal Technique - Pre-Order Technique(Pseudocode & C++ Program)
1. Pre-order (NLR) (node - left - right)
- Access the data part of the current node.
- Traverse the left subtree by recursively calling the pre-order function.
- Traverse the right subtree by recursively calling the pre-order function.
1. Pre-order (Pseudocode) -
void printPreorder(TreeNode* r)
{
1. IF r==NULL THEN -> return
2. PRINT r->value
3. printPreorder(r->left)
4. printPreorder(r->right)
}
OUTPUT -
30 18 10 25 45 65
n1 (#31)
#16 30 #99
1. printPreorder(n1)
2. printPreorder(n2)
9. printPreorder(n3)
n2 (#16)
#6 18 #26
n3 (#99)
45 #83
3. printPreorder(n4)
6. printPreorder(n5)
10. printPreorder(NULL)
11. printPreorder(n6)
n4 (#6)
NULL 10 NULL
n5 (#26)
25
n6 (#83)
65
4. printPreorder(NULL)
5. printPreorder(NULL)
7. printPreorder(NULL)
8. printPreorder(NULL)
12. printPreorder(NULL)
13. printPreorder(NULL)

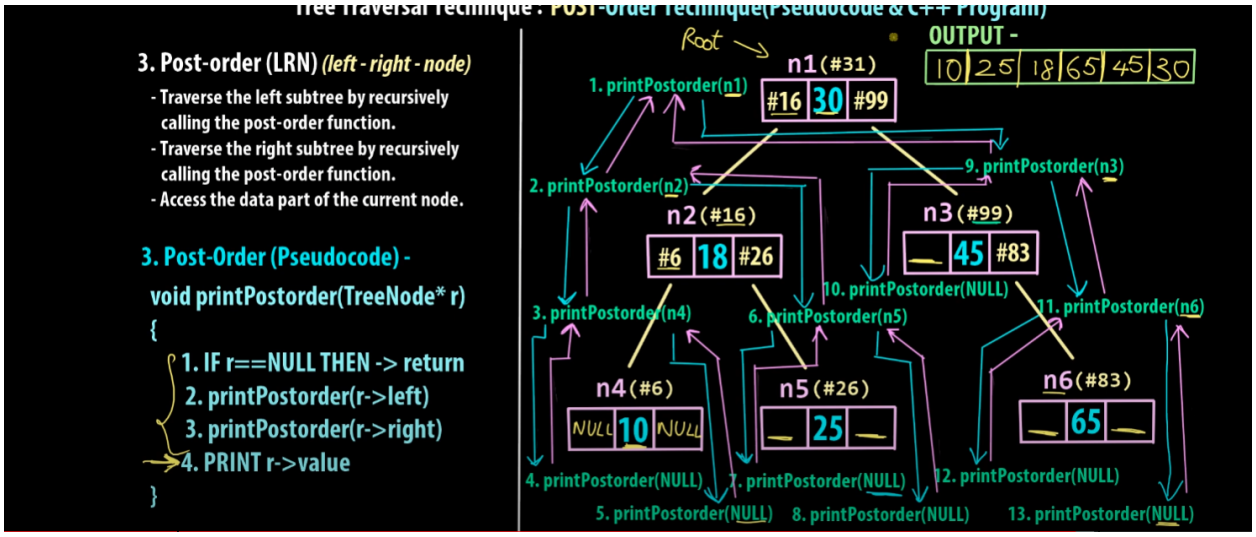
InOrder



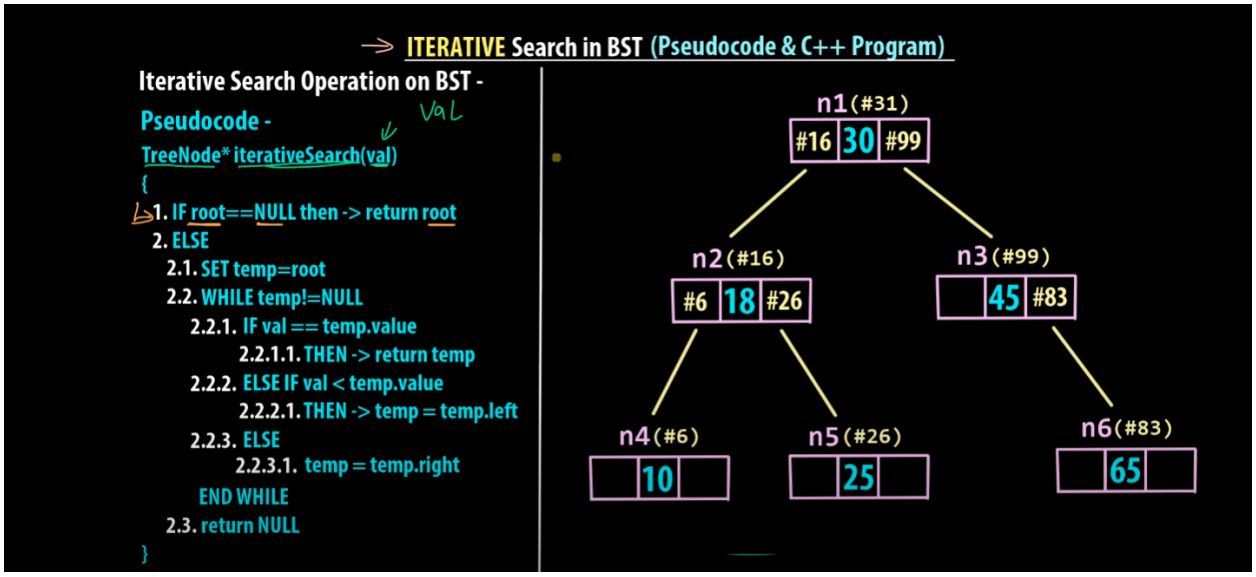


PostOrder

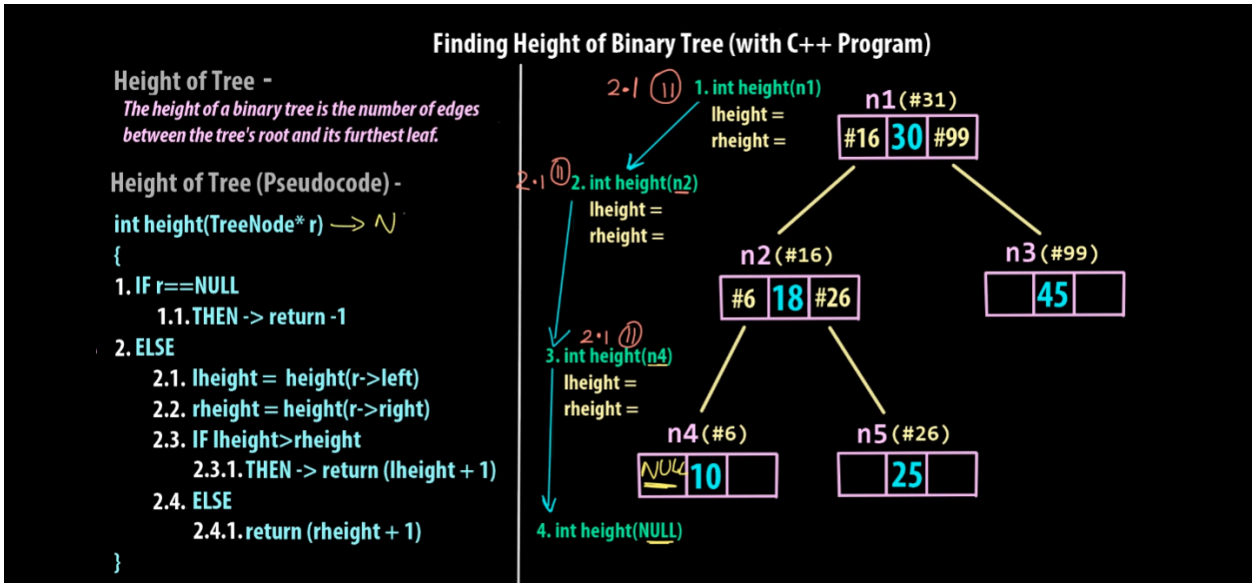

Tree Traversal Technique : POST-Order Technique(Pseudocode & C++ Program)
3. Post-order (LRN) (left - right - node)
- Traverse the left subtree by recursively calling the post-order function.
- Traverse the right subtree by recursively calling the post-order function.
- Access the data part of the current node.
3. Post-Order (Pseudocode) -
void printPostorder(TreeNode* r)
{
1. IF r==NULL THEN -> return
2. printPostorder(r->left)
3. printPostorder(r->right)
4. PRINT r->value
}
OUTPUT -
n1(#31)
#16 30 #99
n2(#16)
#6 18 #26
n3(#99)
45 #83
n4(#6)
10
n5(#26)
25
n6(#83)
65


3. Post-order (LRN) (left - right - node)
- Traverse the left subtree by recursively calling the post-order function.
- Traverse the right subtree by recursively calling the post-order function.
- Access the data part of the current node.
3. Post-Order (Pseudocode) -
void printPostorder(TreeNode* r)
{
1. IF r==NULL THEN -> return
2. printPostorder(r->left)
3. printPostorder(r->right)
4. PRINT r->value
}
Root
OUTPUT -
10 25 18 65 45 30
n1(#31)
#16 30 #99
n2(#16)
#6 18 #26
n3(#99)
45 #83
n4(#6)
NULL 10 NULL
n5(#26)
25
n6(#83)
65
1. printPostorder(n1)
2. printPostorder(n2)
3. printPostorder(n4)
4. printPostorder(NULL)
5. printPostorder(NULL)
6. printPostorder(n5)
7. printPostorder(NULL)
8. printPostorder(NULL)
9. printPostorder(n3)
10. printPostorder(NULL)
11. printPostorder(n6)
12. printPostorder(NULL)
13. printPostorder(NULL)

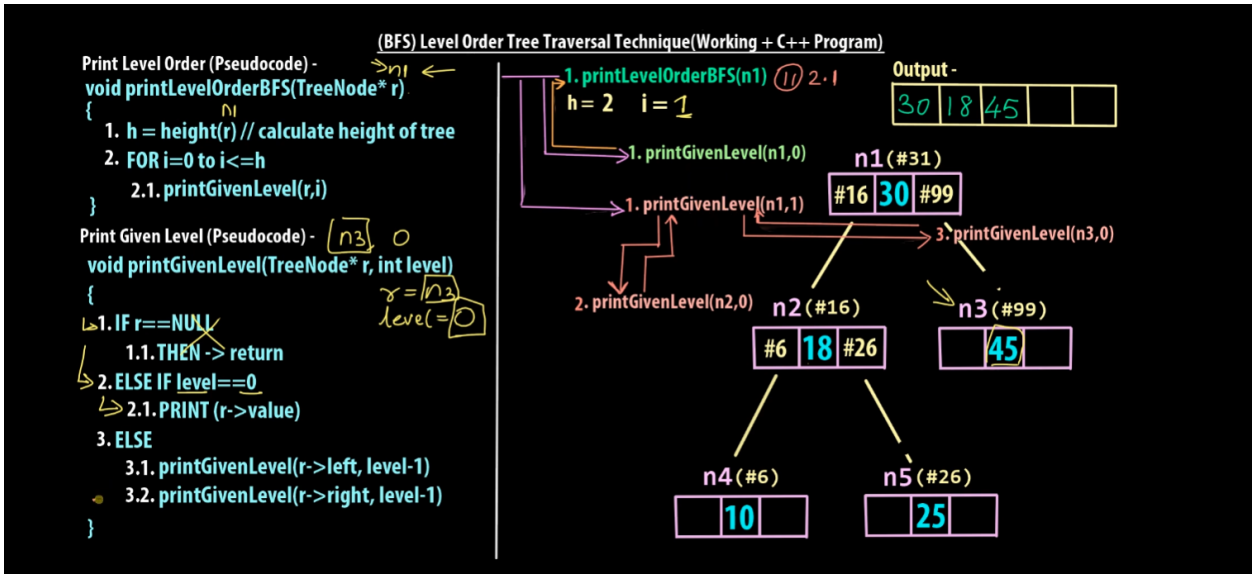
Search



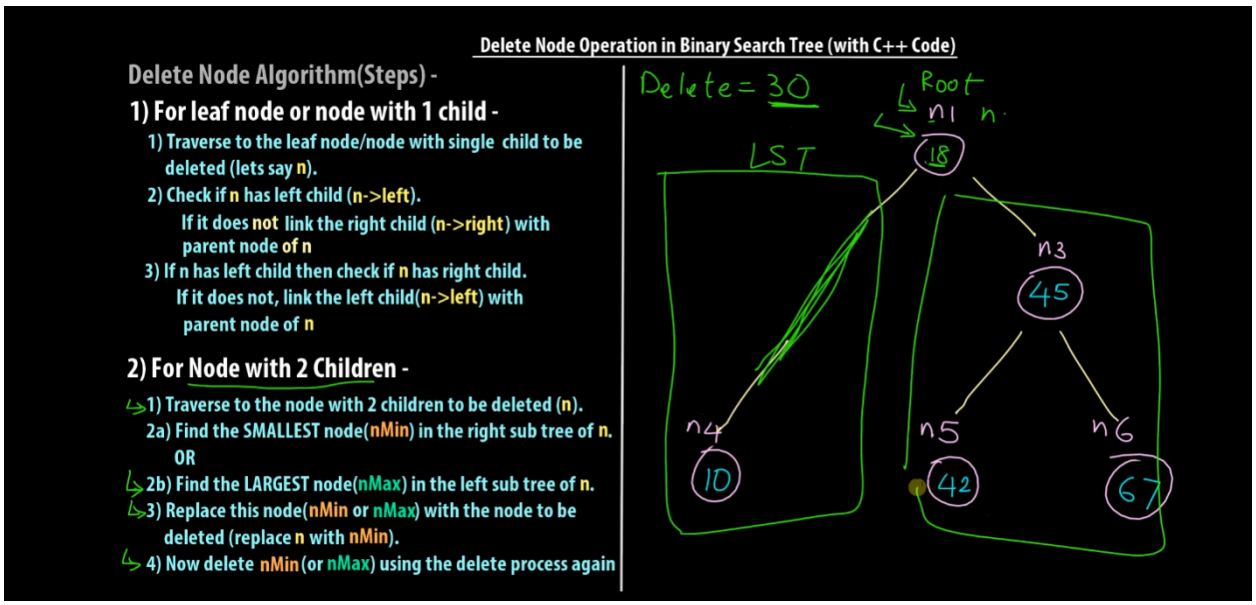
Height Of BST

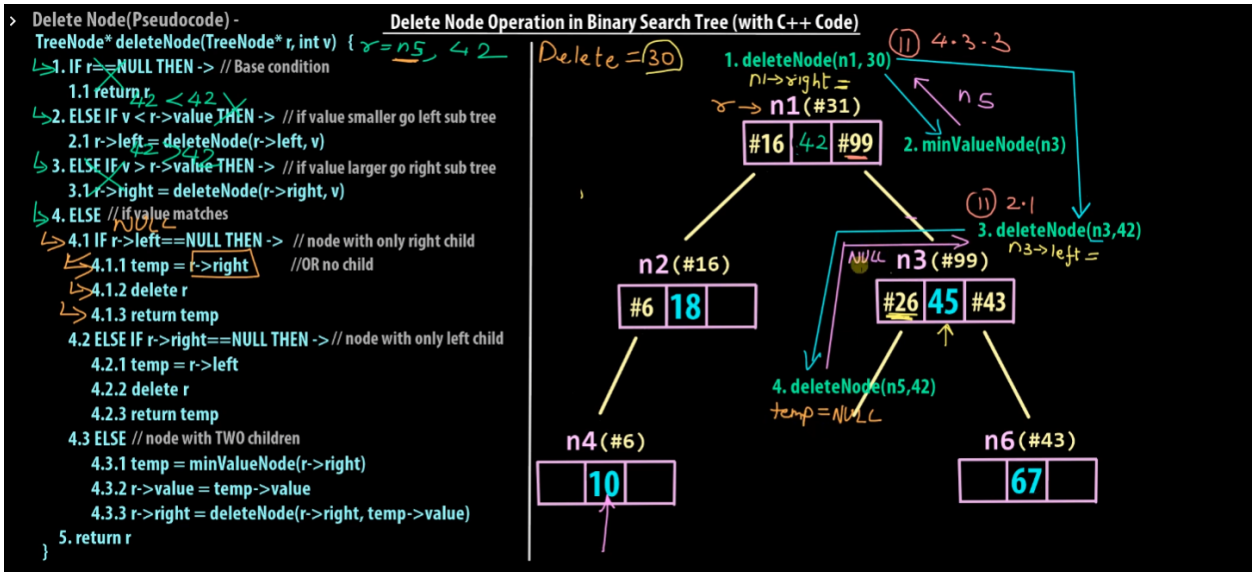


BFS OR LEVEL ORDER TRAVERSAL

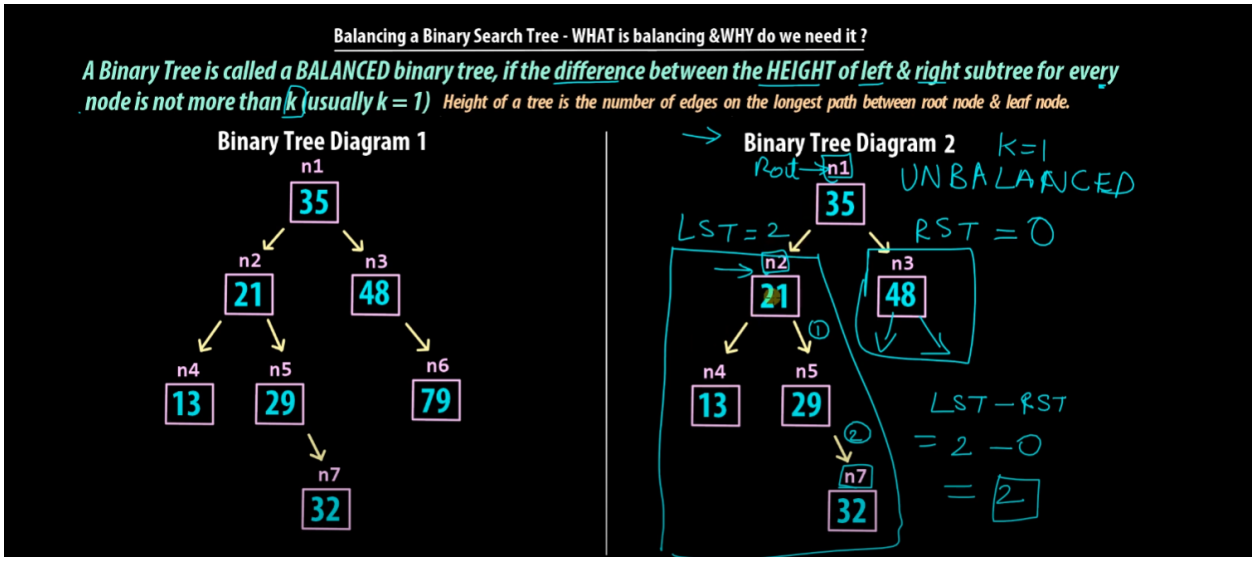

(BFS) Level Order Tree Traversal Technique(Working + C++ Program)
Print Level Order (Pseudocode) -
void printLevelOrderBFS(TreeNode* r)
{
1. h = height(r) // calculate height of tree
2. FOR i=0 to i<=h
2.1. printGivenLevel(r,i)
}
Print Given Level (Pseudocode) -
void printGivenLevel(TreeNode* r, int level)
{
1. IF r==NULL
1.1. THEN -> return
2. ELSE IF level==0
2.1. PRINT (r->value)
3. ELSE
3.1. printGivenLevel(r->left, level-1)
3.2. printGivenLevel(r->right, level-1)
}
1. printLevelOrderBFS(n1)
h = 2 i = 1
1. printGivenLevel(n1,0)
1. printGivenLevel(n1,1)
2. printGivenLevel(n2,0)
3. printGivenLevel(n3,0)
Output -
30 18 45
n1 (#31)
#16 30 #99
n2 (#16)
#6 18 #26
n3 (#99)
45
n4 (#6)
10
n5 (#26)
25

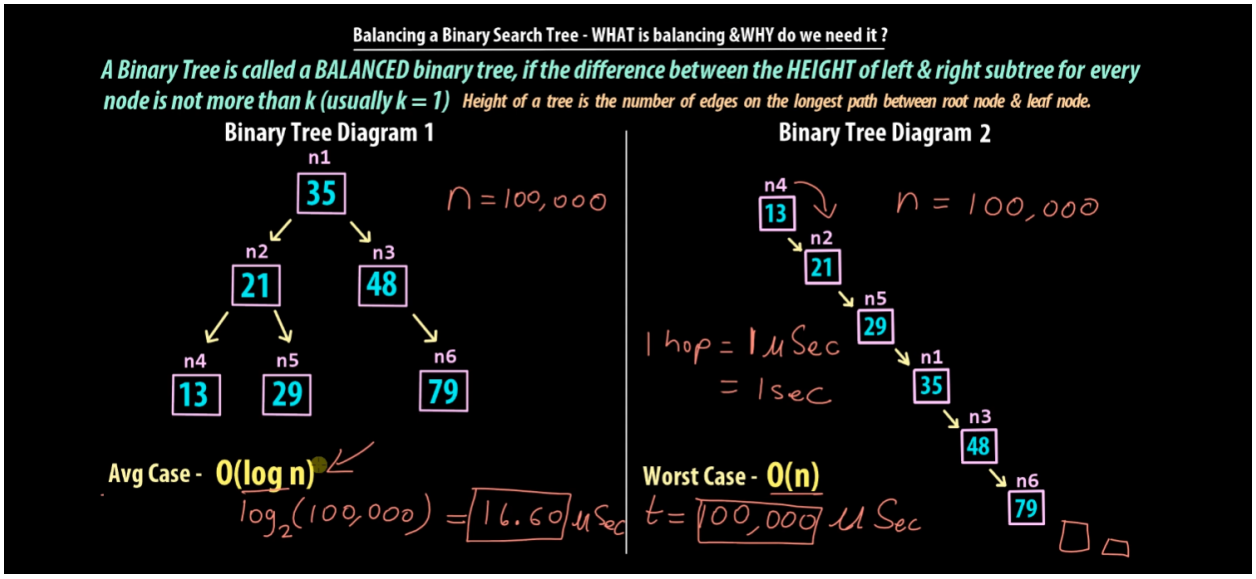
DELETE


Delete Node Operation in Binary Search Tree (with C++ Code)
Delete Node Algorithm(Steps) -
1) For leaf node or node with 1 child -
1) Traverse to the leaf node/node with single child to be deleted (lets say n).
2) Check if n has left child (n->left).
If it does not link the right child (n->right) with parent node of n
3) If n has left child then check if n has right child.
If it does not, link the left child(n->left) with parent node of n
2) For Node with 2 Children -
1) Traverse to the node with 2 children to be deleted (n).
2a) Find the SMALLEST node(nMin) in the right sub tree of n.
OR
2b) Find the LARGEST node(nMax) in the left sub tree of n.
3) Replace this node(nMin or nMax) with the node to be deleted (replace n with nMin).
4) Now delete nMin (or nMax) using the delete process again
Delete = 30
Root
LST
18
n3
45
n4
10
n5
42
n6
67



Balancing a binary tree and why do we need balancing





AVL Tree is a self balancing tree.