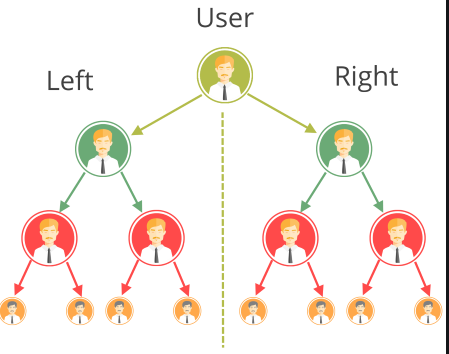
# What is tree?

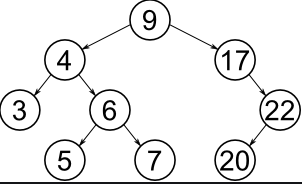
Tree is a hierarchy form of representation of real time examples. It stores the node and reference like Linked list.

It consist of root node and 2 disjoint nodes-> Right and left.

**Company Structure defined:**



**Real time Tree program:**



# Why we should learn tree?

## We have the physical DS, why we need tree?

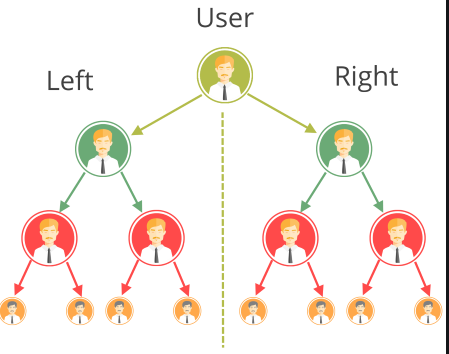
**Performance of Array and Linked List:**

|  |  |  |
| --- | --- | --- |
| Parameters | Array | Linked List |
| Insert/Delete | O (n) | O (n) |
| Search/Traverse | O (n) | O (n) |
| Creation/Deletion | O (1) | O (1) |
| Space efficiency | NO | YES |
| Implementation | Easy | Medium |

What tree comes to the table?

What will be the disadvantages?

# Tree terminology- Part1



**Root**: Node without parent

**Edge**: The link between a child and parent

**Leaf**: A node without any child.

**Sibling**: Children of same parents

**Ancestor**: Parents and Grandparents of a particular node

**Depth of a node:** Length of the tree from root to the particular node.

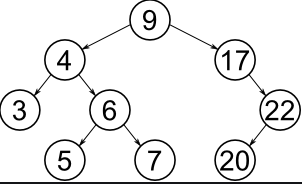
**Height of node:** Length between a particular node and the leaf.

**Depth of a tree**: Length of the tree from root to the deepest node.

**Height of tree:** Length between a root and the deepest node.

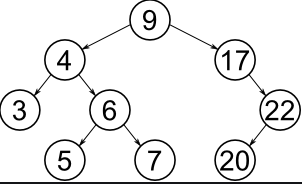
# Tree terminology- Part2

**Predecessor**: The node which is immediate previous node of a tree after performing In-order traversal or ascending order.



**In-order traversal: [3, 4, 5, 6, 7, 9, 17, 20, 22]**

**Successor**: The node which is immediate next node of a tree after performing In-order traversal or ascending order.



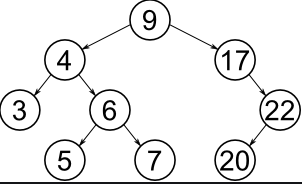
**In-order traversal: [3, 4, 5, 6, 7, 9, 17, 20, 22]**

# What is Binary tree

When a tree contains no more than 2 child then it is called as Binary tree.

Many tree follow the steps of binary tree, like:

* Binary Search tree
* Heap tree
* AVL Tree.



# Why Binary tree needs to be learned?

To get understanding over the problem solving similar to the property of Binary tree

Many trees follows the principle of Binary tree.

* Binary Search tree
* Heap tree
* AVL Tree.

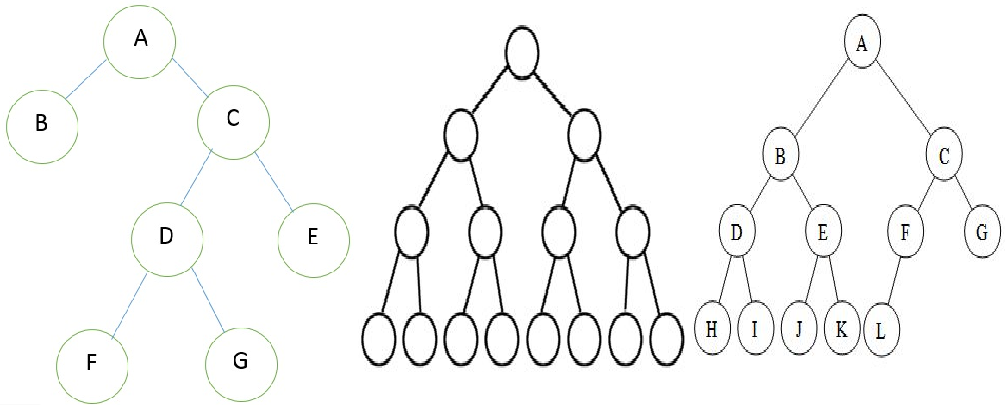
Many algorithm technique used binary tree as a data structure to solve problem. EX: Huffman

# Types of Binary tree

**Strict Binary tree**: If a binary tree is having 2 nodes or 0 nodes.

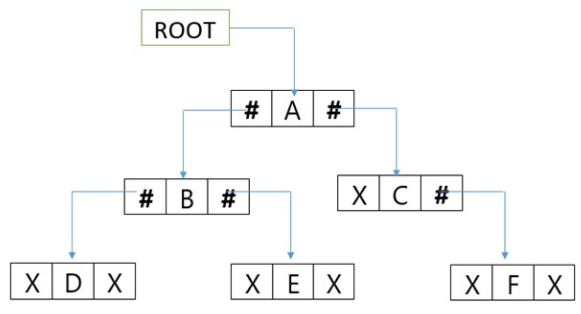
**Full Binary tree**: All the nodes is having 2 nodes equal to the depth of the tree, except the leaf node.

**Complete binary tree:** All the node were filled with 2 nodes except the last level.



# How binary tree represented in code

**Implementation using Linked list:**



**Implementation using array:**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| x | A | B | C | D | E |  | F |  |  |

Left Node: 2X; Right node: 2x+1

# Create a blank Binary tree (LL)

**Common operation of Binary Tree:**

Creation, Insertion, Traverse, search, delete.

**Algorithm:**

CreateBT()

Create object for binary tree

**Time complexity: O(1) Space Complexity: O(1)**

Why to learn traversal before insert.

**Depth First search:**

Pre-Order traversal

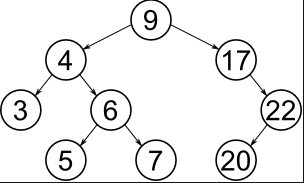
In-Order traversal

Post-Order traversal

**Breath first search:**

Level-Order traversal

# Pre-order traversal in Binary tree (LL)



**Traversal value:**

Preorder(root)

If root=null

Return error

Else

Print root

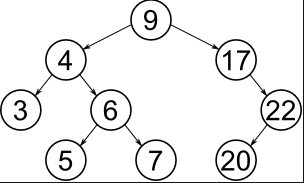
Preorder(root.left)

Preorder(root.right)

**Stack:**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

# In-order traversal in Binary tree (LL)



**Algorithm**:

Inorder(root)

If root = null

Print error

Else

Inorder(root.left)

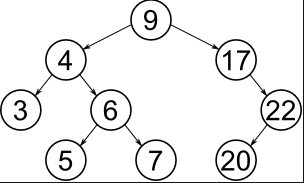
Print (root)

Inorder(root.right)

**Time Complexity: O (n) Space complexity: O(n)**

|  |
| --- |
|  |
|  |
|  |
|  |

# Post-order traversal in Binary tree (LL)



**Algorithm**:

PostOrder(root)

If root=null

Return error

Else

PostOrder(root.left)

PostOrder(root.right)

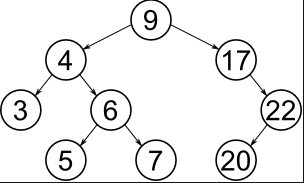
Print(root)

**Time Complexity: O (n) Space complexity: O(n)**

**Stack**

|  |
| --- |
|  |
|  |
|  |
|  |

# Level-order traversal in Binary tree (Array)



Uses queue for processing the Binary tree.

**Algorithm**:

LevelOrder(root)

Create enqueue(Q)

Enqueue (root)

While queue is not empty

Enqueue() # Child of first element

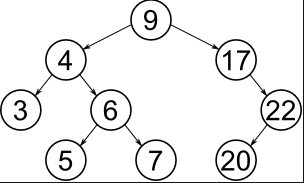
Dequeue() and print

**Time Complexity: O (n) Space complexity: O(n)**

**Queue**:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 14 |
|  | 9 | 4 | 17 | 3 | 6 |  | 22 |  |  | 5 | 7 |  | 20 |

# Insertion in Binary tree (LL)



Two condition:

1. Insert when tree is empty
2. Insert as vacant child or searching for a possible position

**Method used:** Level order traversal.

**Algorithm**:

Insert(node)

If root is null

Root = node

Else

Perform Level search and find first blank

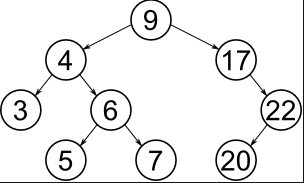
Insert in blank node

**Queue:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

Time complexity: O(n) Space Complexity: O(n)

# Searching in Binary tree (LL)



Method used: **Level Order**

**Algorithm:**

Search(Value)

If root = null

Return Error

Else

LevelOrder(root)

Create enqueuer(Q)

Enqueue (root)

While queue is not empty

Enqueuer() # Child of first element

If value is found

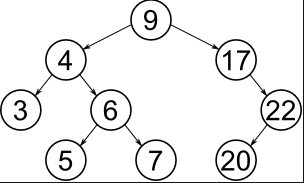
Return value/Success

Dequeuer()

Return NA

**Time Complexity: O (n) Space complexity: O(n)**

# Deletion in Binary tree (LL)



Two condition:

1. The node is not available
2. Node available

**Method used:** Level order traversal.

**Deletion:** Root, leaf and parents

**Algorithm**:

Delete (node)

If root is null

Root = node

Else

Search for the node to be deleted

Find the deepest node of the tree

Replace the deepest node into the node to be deleted

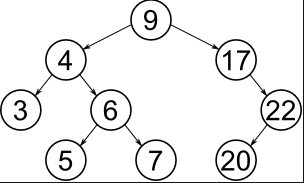
Delete deepest node

**Queue:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

**Time complexity: O(n) Space Complexity: O(n)**

# Delete Binary tree (LL)



Algorithm:

DeleteTree()

If root= null

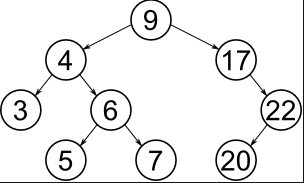
Return error

Else

Root = null

**Time complexity: O(1) Space Complexity: O(1)**

# Create a blank Binary tree (Array)



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| x | 9 | 4 | 17 | 3 | 6 |  | 22 |  |  | 5 | 7 |  |  | 20 |  |  |  |  |  |

Left Node: 2X

Right Node: 2X+1

**Algorithm to create a Binary tree using array:**

Create ()

Create blank array of size

Last updated value is 0

**Time complexity: O(1) Space Complexity: O(n)**

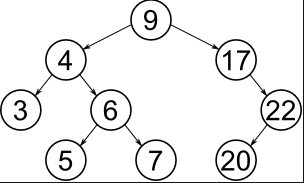
# Insertion in Binary tree (Array)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| x |  | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |  |  |  |  |  |  |  |  |  |  |  |

Insertion:

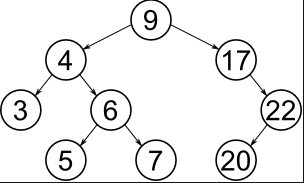
1. Insert when array is full, return error
2. Insert at first vacant array

**Advanced case:**



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| x | 9 | 4 | 17 | 3 | 6 |  | 22 |  |  | 5 | 7 |  |  | 20 |  |  |  |  |  |

**Algorithm**:



|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| x | 9 | 4 | 17 | 3 | 6 |  | 22 |  |  | 5 | 7 |  |  | 20 |  |  |  |  |  |

InsertNode(Node)

If tree is full

Return error

Else

Insert the value in first unused cell

Update the last used index

**Time complexity: O(1) Space Complexity: O(n)**

# Searching in Binary tree (Array)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| x |  | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |  |  |  |  |  |  |  |  |  |  |  |

**Condition of search:**

1. Node missing
2. Node available

How to search: Traversal

**Algorithm**:

Search(node)

Traverse from 0 to LastUsedIndex-1

If value=node

Print node/Success

Return Error

**Time complexity: O (n); Space Complexity: O (1)**

# In-order traversal in Binary tree (Array)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| x |  | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |  |  |  |  |  |  |  |  |  |  |  |

Logic: Left, root, right

**Algorithm**:

Inorder(index)

If index>lastUsedIndex

Return null

Else:

Inorder(index\*2)

Print(currentindex)

Inorder(index\*2+1)

**Time complexity: O (n); Space Complexity: O (1)**

# Pre-order traversal in Binary tree (Array)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| x |  | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |  |  |  |  |  |  |  |  |  |  |  |

Logic: root, left, right

**Algorithm**:

Preorder(index)

If index>lastUsedIndex

Return null

Else:

Print(currentindex)

Inorder(index\*2)

Inorder(index\*2+1)

**Time complexity: O (n); Space Complexity: O (1)**

# Post-order traversal in Binary tree (Array)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| x |  | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |  |  |  |  |  |  |  |  |  |  |  |

Logic: left, right, root

**Algorithm**:

Postorder(index)

If index>lastUsedIndex

Return null

Else:

Inorder(index\*2)

Inorder(index\*2+1)

Print(currentindex)

**Time complexity: O (n); Space Complexity: O (1)**

# Level-order traversal in Binary tree (Array)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| x |  | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |  |  |  |  |  |  |  |  |  |  |  |

Logic: Level

**Algorithm**:

Levelorder(index)

Loop from index to lastUsedIndex-1

Print currentIndex

**Time complexity: O (n); Space Complexity: O (1)**

# Deletion in Binary tree (Array)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| x |  | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |  |  |  |  |  |  |  |  |  |  |  |

Logic: Level

**Conditions**:

1. Node missing in array
2. Node available in array

Replace logic: LastUsedIndex<->Node ; Delete and modify LastUsedIndex to LastUsedIndex-1

**Algorithm**:

DeleteNode(Node)

Search from index to LastUsedIndex

If value=node

Print value/success message

Print not available

**Time complexity: O (n); Space Complexity: O (1)**

# Delete Binary tree (Array)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| x |  | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 |  |  |  |  |  |  |  |  |  |  |  |

Logic: Array deletion

**Algorithm**:

DeleteTree()

Array = null

**Time complexity: O (1); Space Complexity: O (1)**