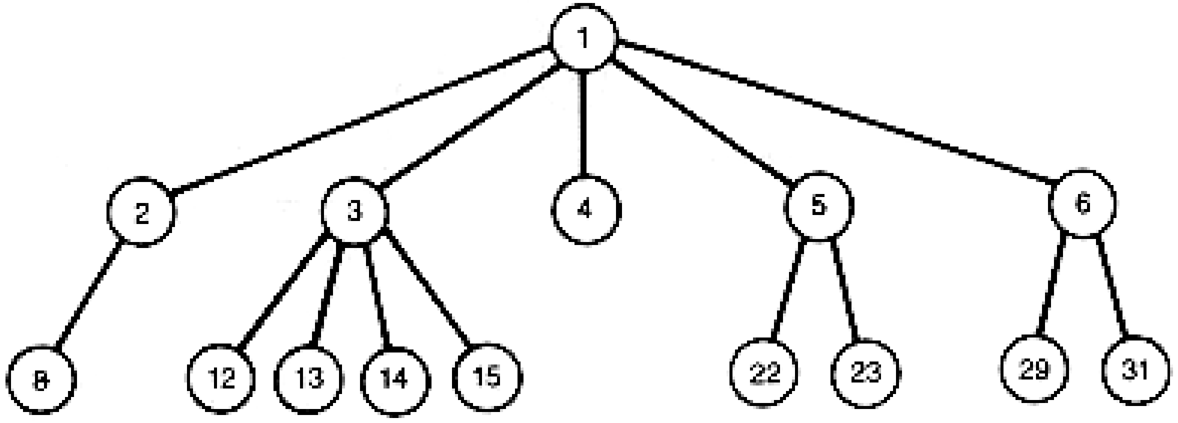
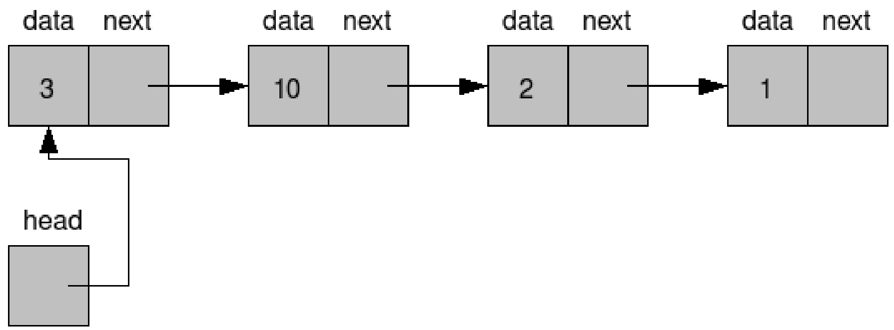
Binary Trees

CS223: Data Structures

Linear or non linear

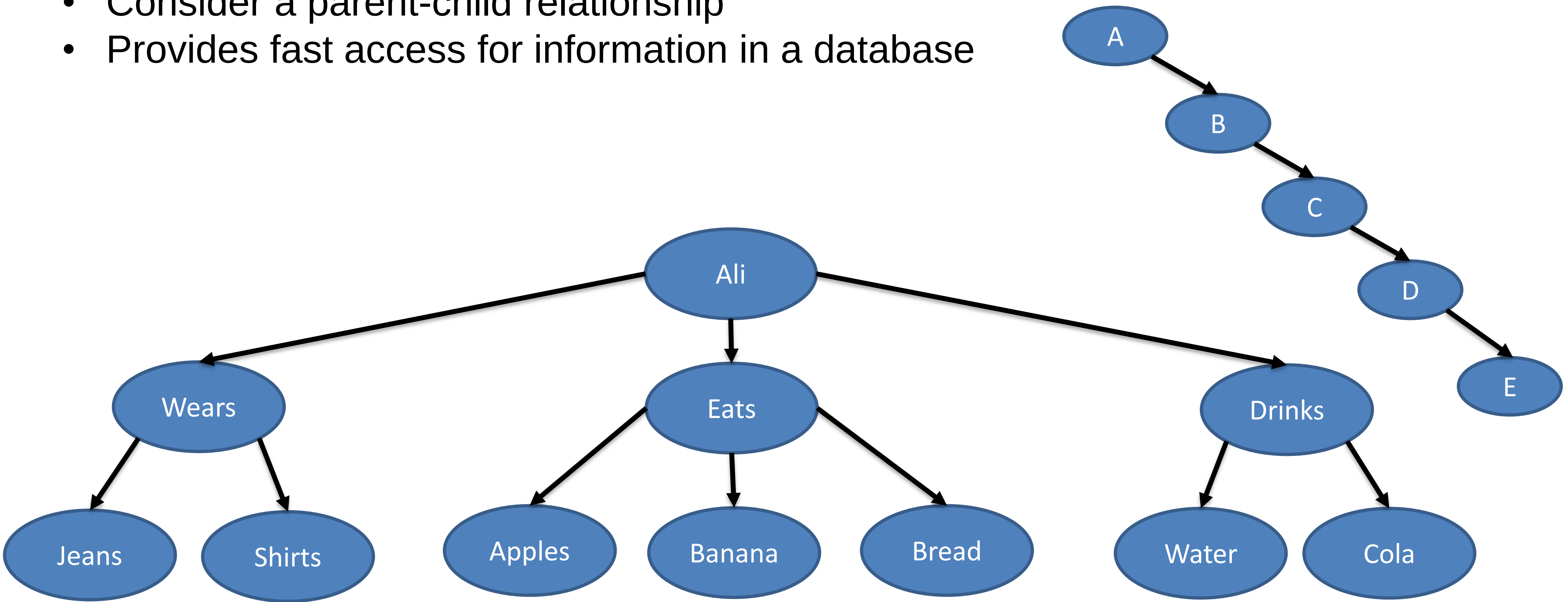
- Lists, stacks, and queues are considered **linear** structures where one item follows another
- Trees are considered non-linear as:
  - Multiple items can follow one item
  - Number of items following another might vary
  - Do not have their elements in a sequence.





A Tree Data Structure

- Useful for representing data containing a hierarchical relationship between elements, for example, records, family trees and table of contents.
- Consider a parent-child relationship
- Provides fast access for information in a database

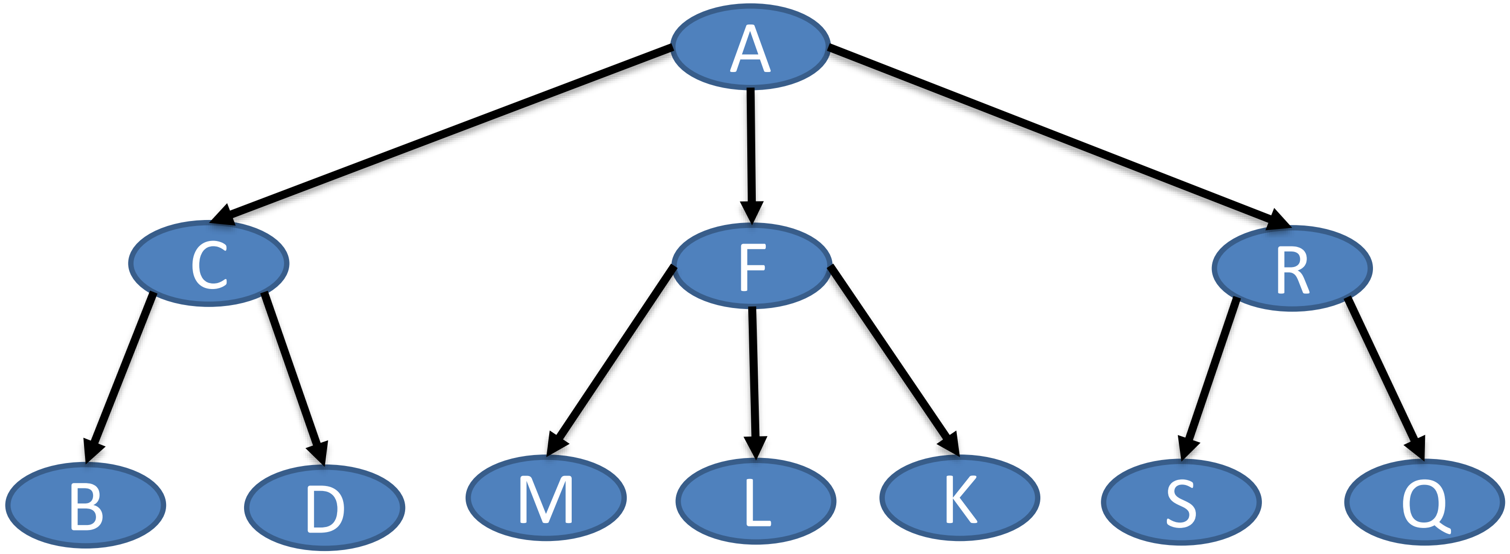


Applications of Trees

- Store hierarchical data, like folder structure, organization structure, XML/HTML data.
- **Binary Search Tree** is a tree that allows fast search, insert, delete on a sorted data. It also allows finding closest item
- A **heap** is a tree data structure which is implemented using arrays and used to implement priority queues.
- **B-Tree** and **B+-Tree**: They are used to implement indexing in databases.
- Syntax Tree: Used in Compilers.
- **K-D Tree:** A space partitioning tree used to organize points in K dimensional space.
- A **trie** is used to implement dictionaries with prefix lookup.
- A **suffix tree** is used for quick pattern searching in a fixed text.
- **Spanning trees** and **shortest path trees** are used in routers and bridges respectively in computer networks.

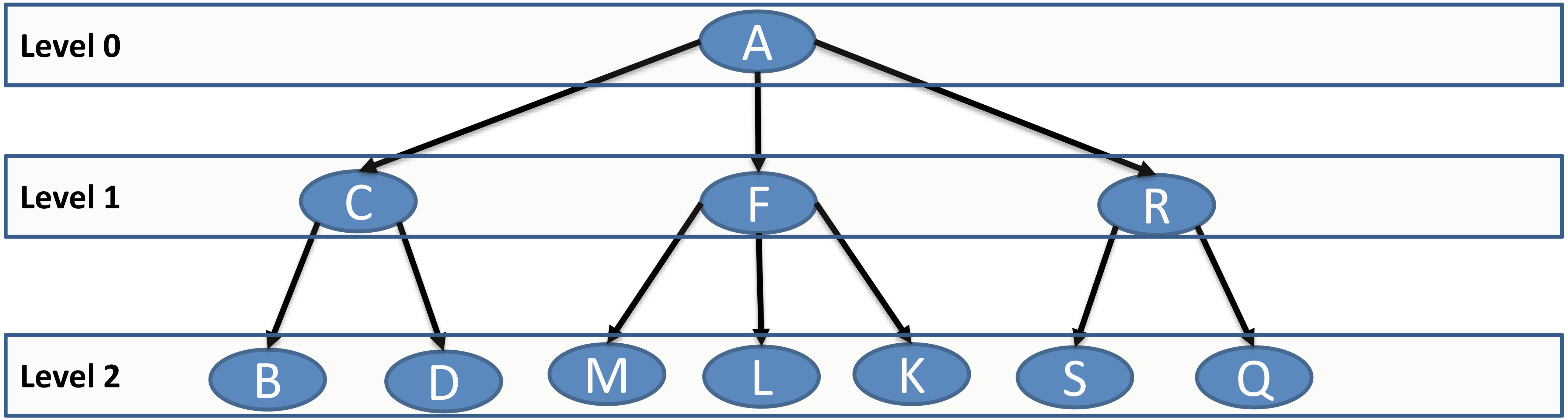
Structure of trees

- Elements (here letters) are called **nodes**
- Arrows are called **edges**
- Topmost node (here A) is called **root**
- Parent-child relations: e.g., F is the **parent** of L and L is the **child** of F
- Nodes which belong to same parent are called as **siblings** (e.g., B;D, or C;F;R).
- Bottom nodes with no edges (here B,D,M, etc) are called **leaf nodes** (0 children)
- The node which has at least one child is called as **internal node** (e.g., C, F, A). Every non-leaf node is internal node.



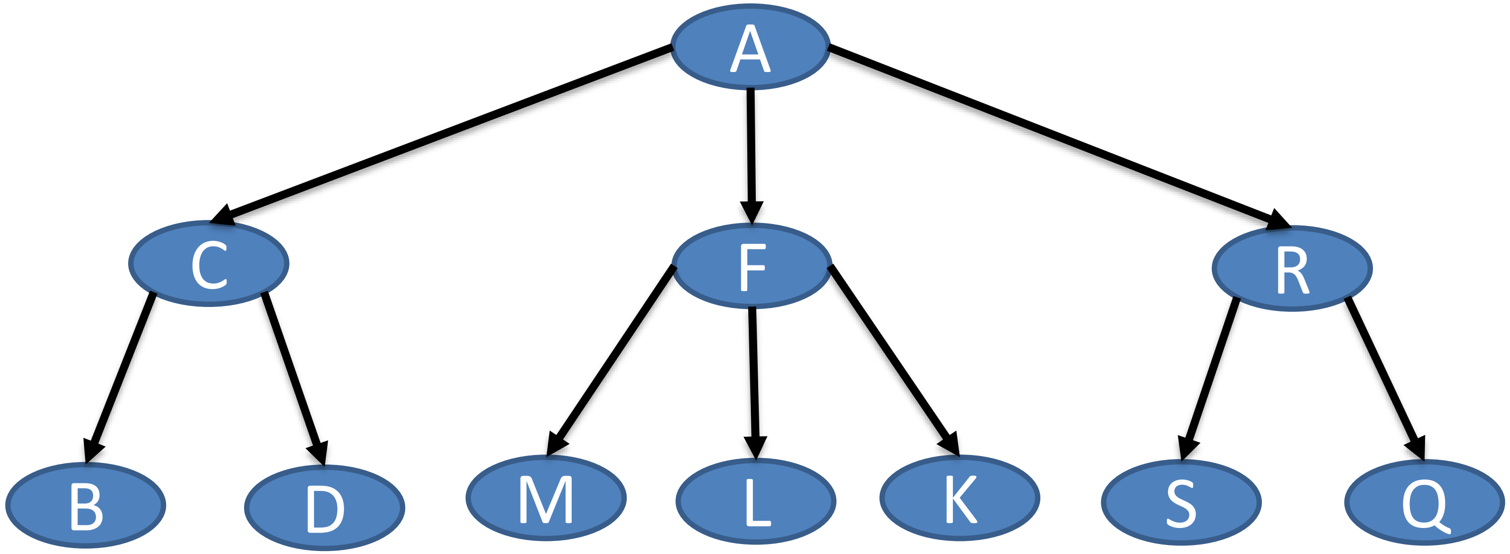
Structure of trees

- The total number of children of a node is referred to as the **degree** of that node
- The highest degree of a node among all the nodes in a tree is referred to as the **degree of tree.** (e.g., degree of C is 2, degree of F is 3, degree of D is 0, degree of the tree is 3)
- In a tree, each step from top to bottom is called as a **level** and the level count starts with '**0**' and **incremented** by one at each level (Step).



Paths, Height, and Depth

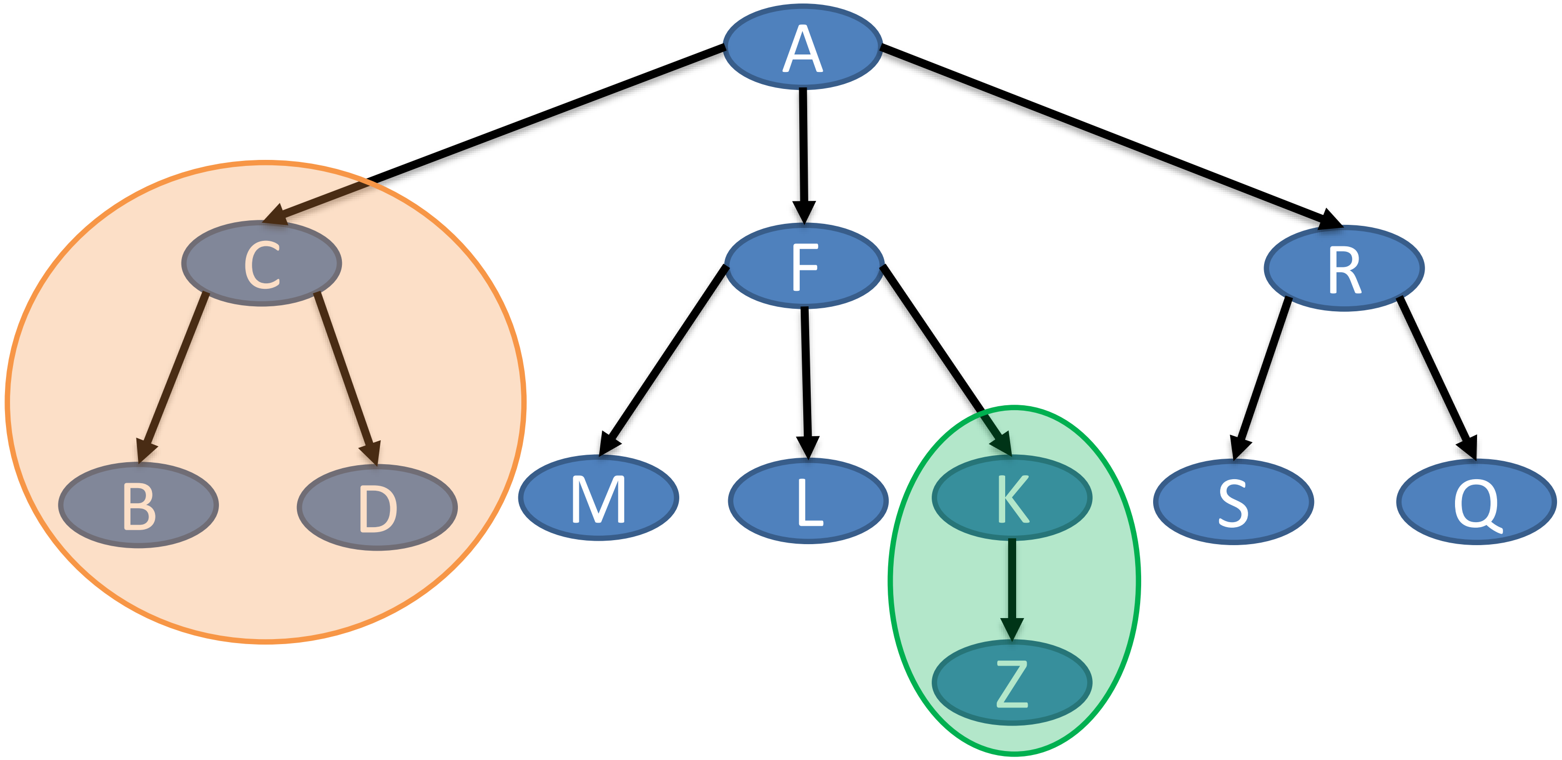
- **Path** is any sequence of consecutive edges from source node to destination node {A,C,B}, {F,M}.
- Length of a path is the number of edges in it
- Longest path from root to leaf is considered the **height** of that tree (here 2), the height of a leaf node is 0.
- **Depth** of a node is the length from the root to that node (Depth of L is 2)



Subtrees

Subtree of a node is one of the children and it's descendants (nodes reached from it).

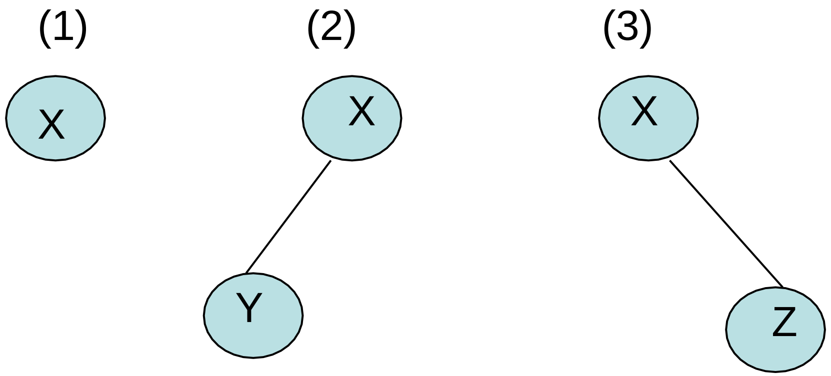
- C, B, D is one subtree of A
- K, Z is the subtree of F

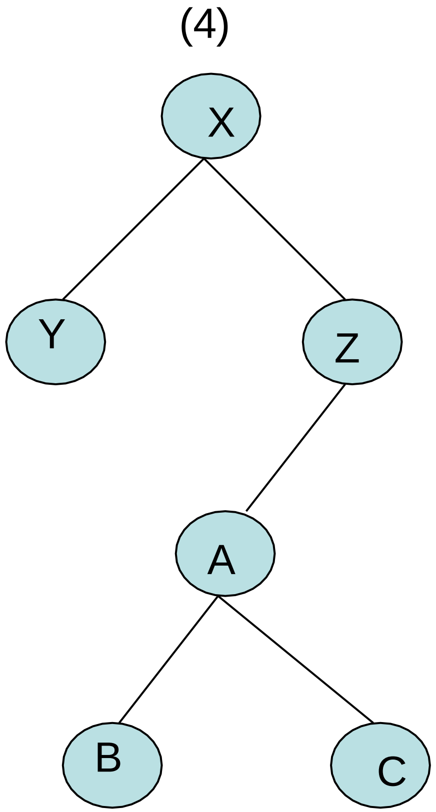


Binary Trees

- A binary tree is a special type of tree in which every node (or vertex) has either no children, one child or two children.
- Characteristics:
  - Every binary tree has a root pointer which points to the start of the tree.
  - A binary tree can be empty.
  - It consists of a node called root, a left subtree and right subtree both of which are binary trees themselves.

Examples : Binary Trees





Root of tree is node having info as X.

(1) Only node is root.

(2) Root has left child Y.

(3) Root X has right child Z.

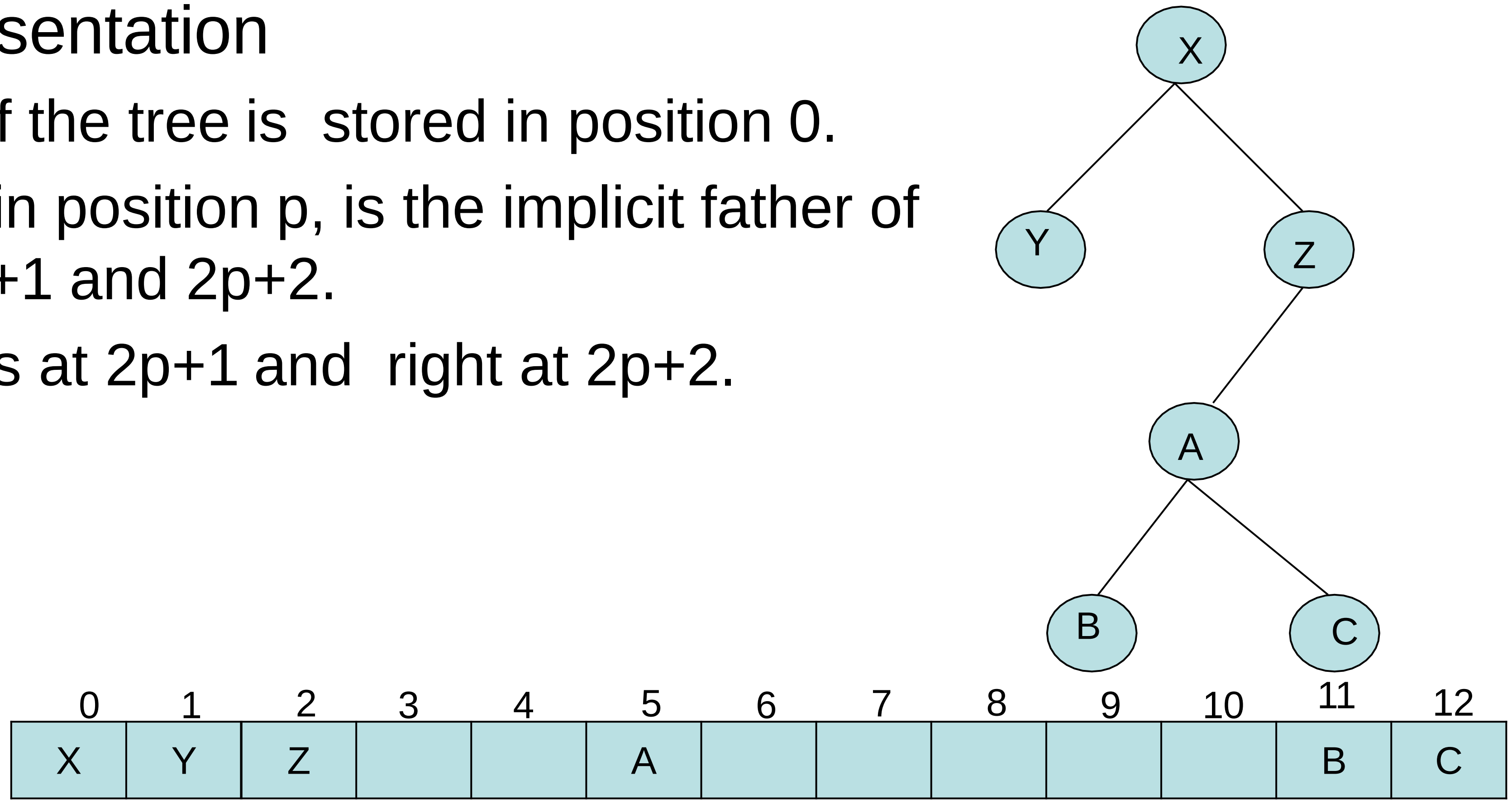
(4) Root X has left child Y and right child Z which is again a binary tree with its parent as Z and left child of Z is A, which in turn is parent for left child B and right child C.

Properties of Binary Tree

- A tree with *n* nodes has exactly (*n*-1) edges or branches.
- In a tree, every node except the root has exactly one parent (and the root node does not have a parent).
- There is exactly one path connecting any two nodes in a tree.
- The maximum number of nodes in a binary tree of height K is $2^{K+1}-1$ where K>=0.

Representation of Binary Tree

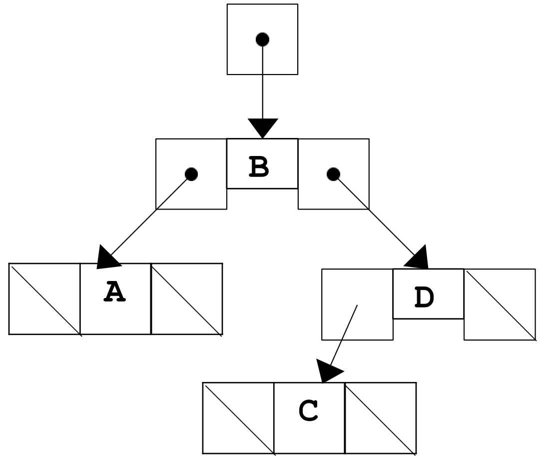
- Array representation
  - The root of the tree is stored in position 0.
  - The node in position p, is the implicit father of nodes 2p+1 and 2p+2.
  - Left child is at 2p+1 and right at 2p+2.



Representation of Binary Tree

- Linked List
  - Every node will consists of information, and two pointers left and right pointing to the left and right child nodes.

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



  - The topmost node or first node is pointed by a root pointer which will inform the start of the tree. Rest of the nodes are attached either to left or right.

Mechanisms of Binary Tree Traversal

- Three common binary tree traversal orderings (each one begins at the root):

  - **preorder traversal**: the current node is processed, then the node's left subtree is traversed, then the node's right subtree is traversed (CURRENT-LEFT-RIGHT)

  - **in-order traversal**: the node's left subtree is traversed, then the current node itself is processed, then the node's right subtree is traversed (LEFT-CURRENT-RIGHT)

  - **postorder traversal**: the node's left subtree is traversed, then the node's right subtree is traversed, and lastly the current node is processed (LEFT-RIGHT-CURRENT)

Pre-order Traversal

- The preorder traversal of a nonempty binary tree is defined as follows:
  - Visit the root node
  - Traverse the left sub-tree in preorder
  - Traverse the right sub-tree in preorder

In-order traversal

- The in-order traversal of a nonempty binary tree is defined as follows:
  - Traverse the left sub-tree in in-order
  - Visit the root node
  - Traverse the right sub-tree in inorder

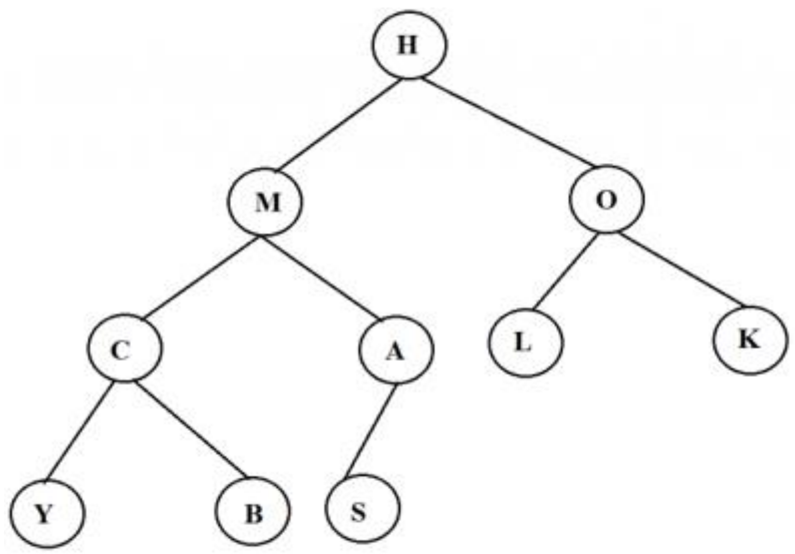
Post-order traversal

- The post-order traversal of a nonempty binary tree is defined as follows:
  - Traverse the left sub-tree in post-order
  - Traverse the right sub-tree in post-order
  - Visit the root node

Level-order traversal

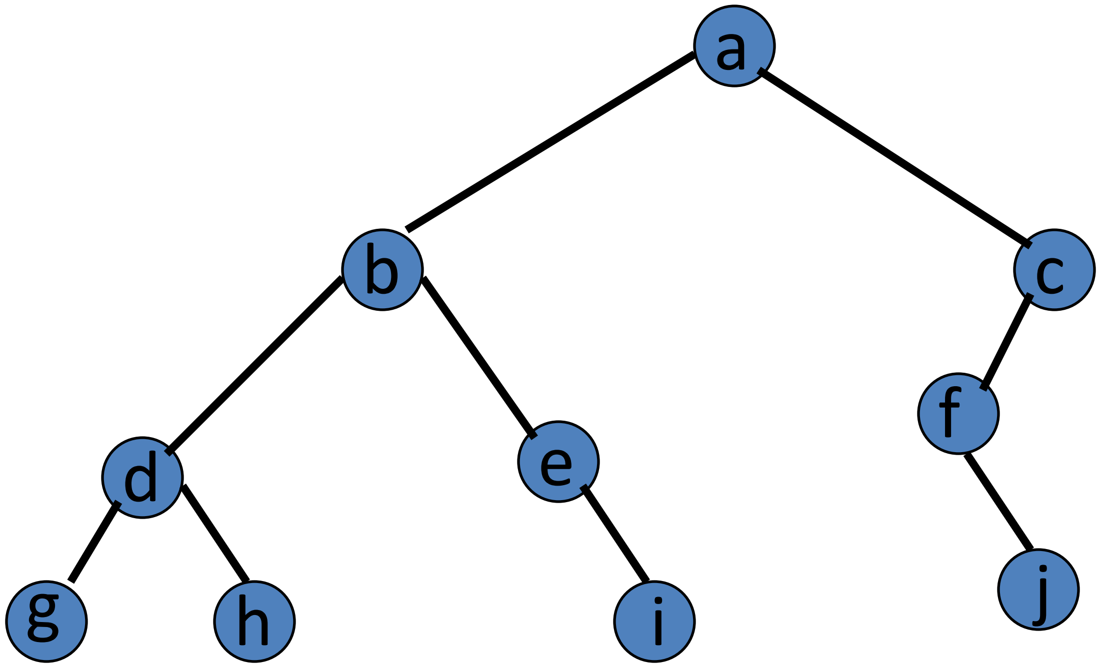
- The level-order traversal of a nonempty binary tree is defined as follows:
  1) Create an empty queue q
  2) Enqueue root /*start from root*/
  3) Loop while queue is not empty
     a) temp_node = dequeue a node from q
     b) print temp_node->data.
     c) if temp_node->left is not null enqueue temp_node->left
     d) if temp_node->right is not null enqueue temp_node-> right

Example of Tree Traversals



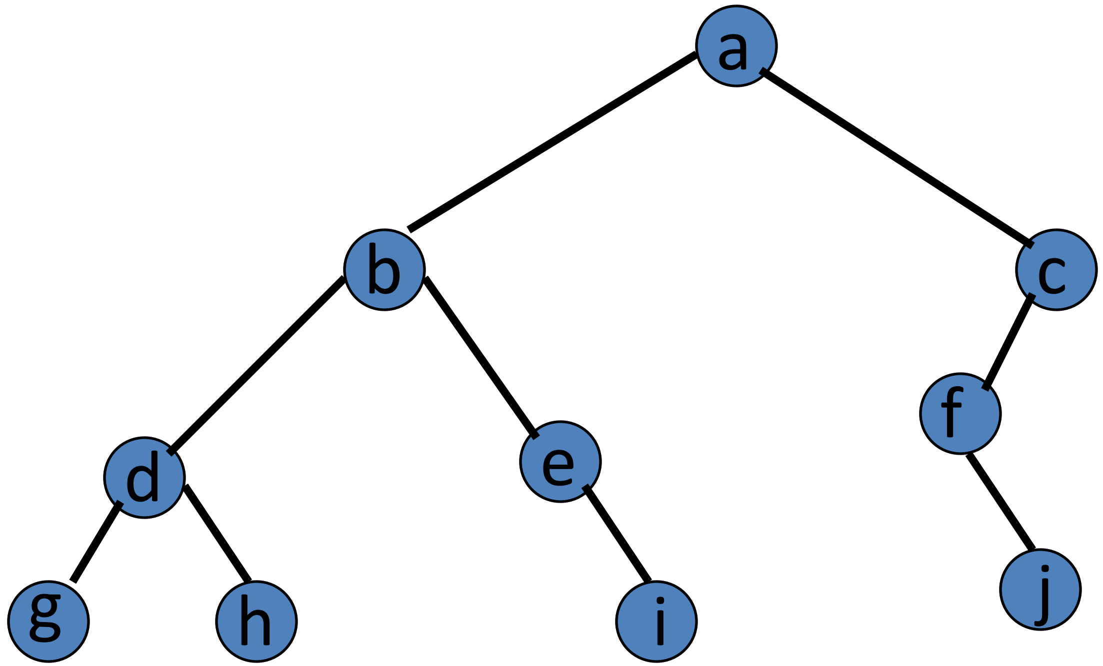
InOrder traversal: Y C B M S A H L O K
PreOrder traversal: H M C Y B A S O L K
PostOrder traversal: Y B C S A M L K O H
LevelOrder traversal: H M O C A L K Y B S

Preorder Example



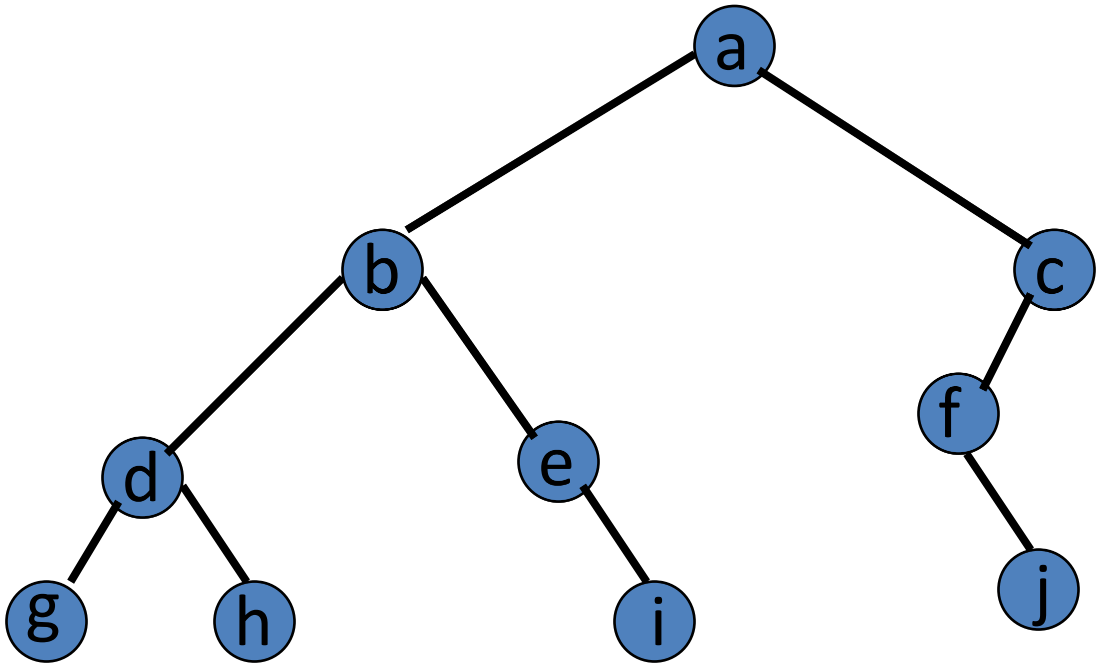
a b d g h e i c f j

Inorder Example



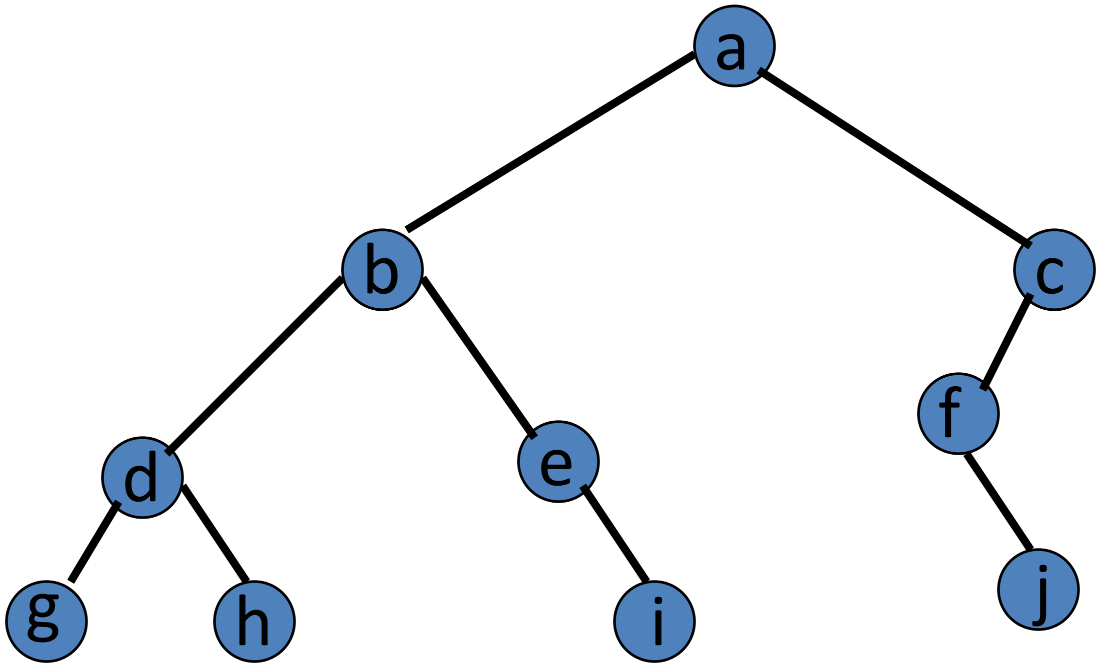
g d h b e i a f j c

Postorder Example



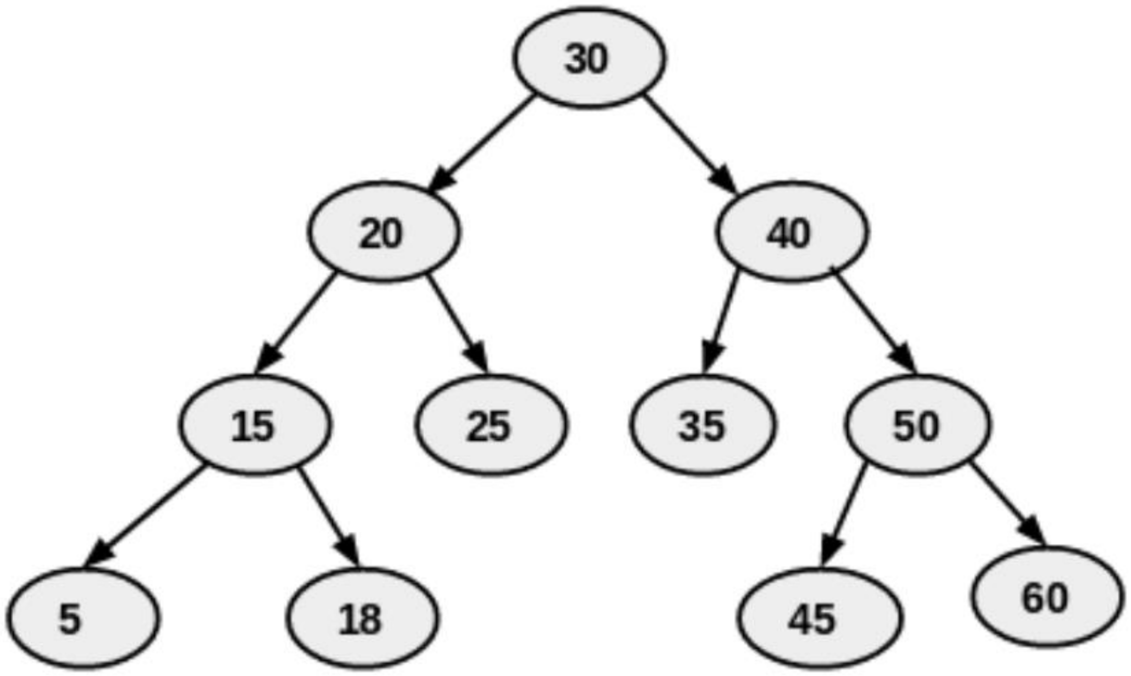
g h d i e b j f c a

Levelorder Example



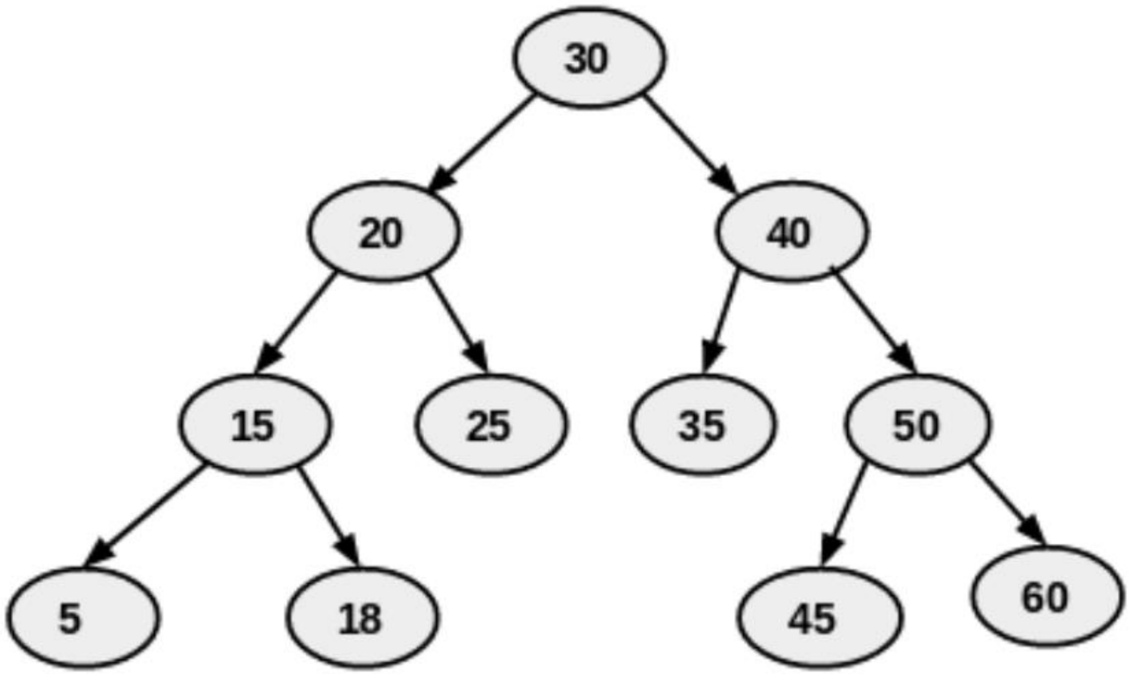
a b c d e f g h i j

Inorder Example



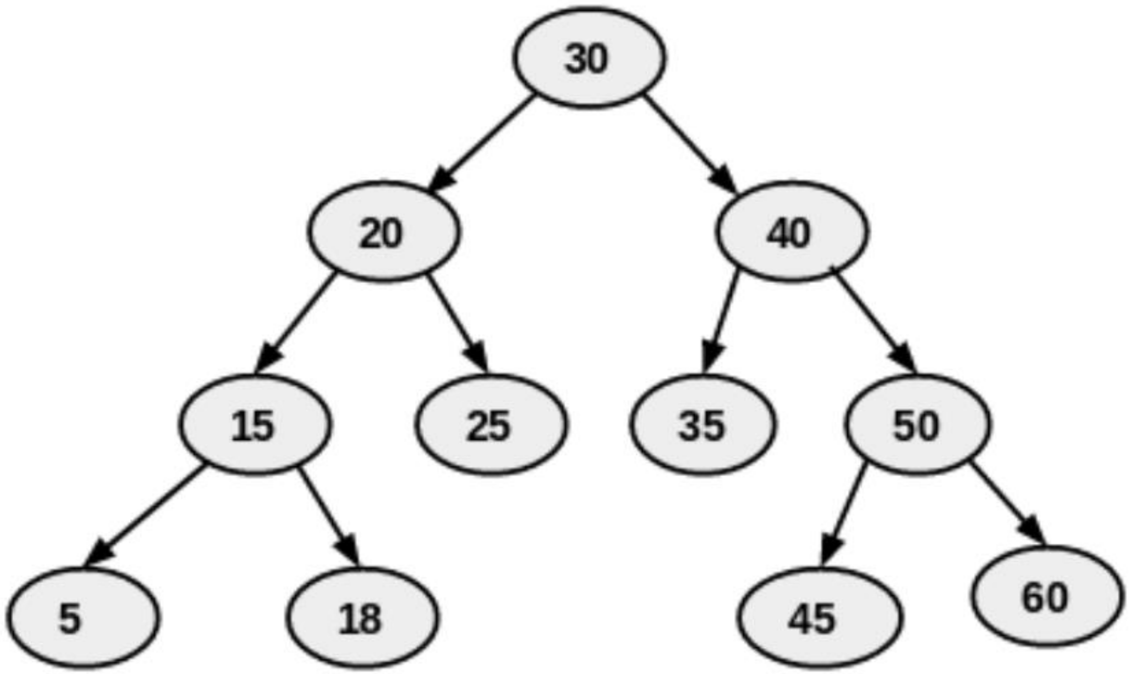
5 , 15 , 18 , 20 , 25 , 30 , 35 , 40 , 45 , 50 , 60

Preorder Example



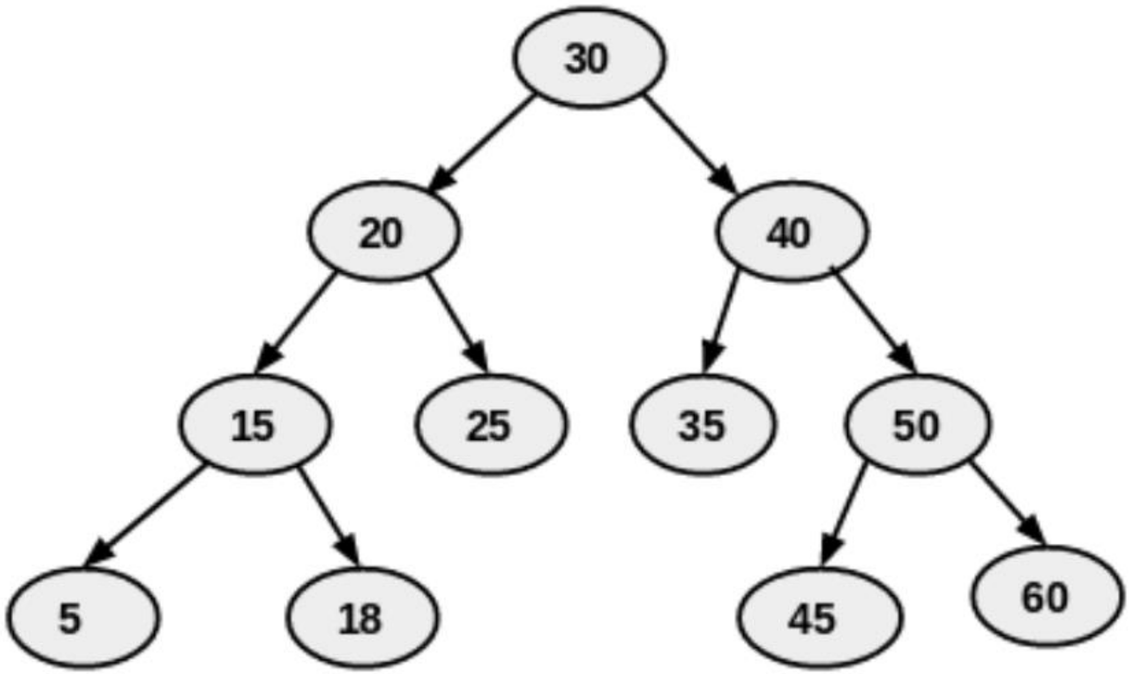
30 , 20 , 15 , 5 , 18 , 25 , 40 , 35 , 50 , 45 , 60

Postorder Example



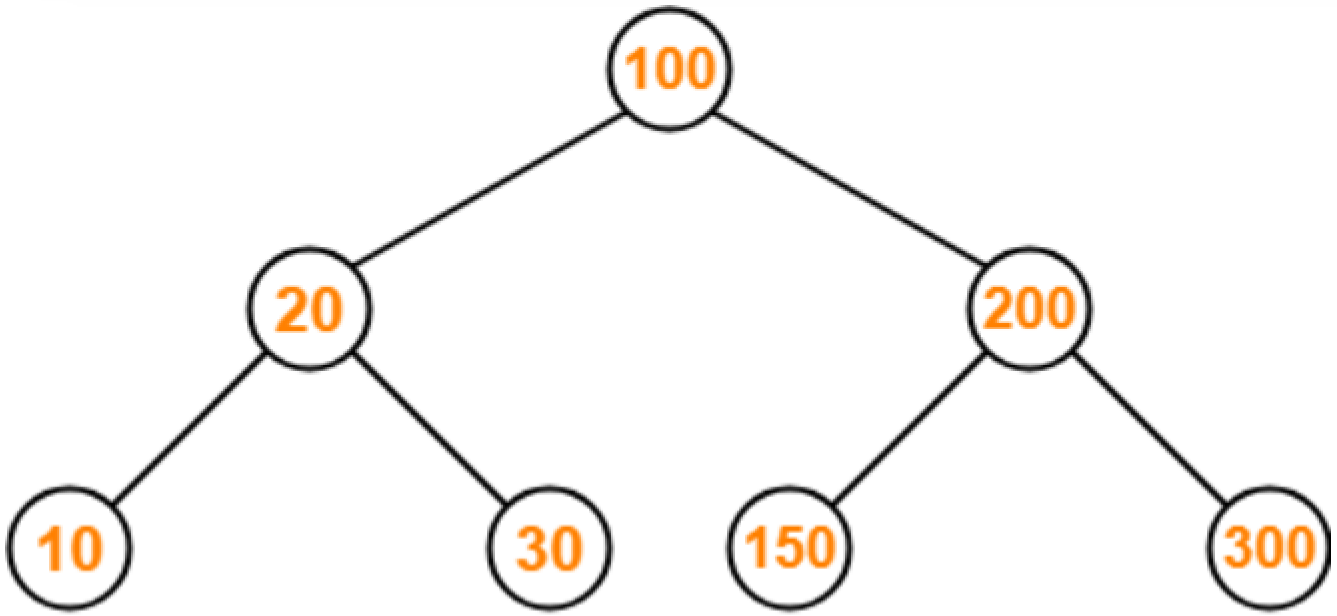
5 , 18 , 15 , 25 , 20 , 35 , 45 , 60 , 50 , 40 , 30

Levelorder Example



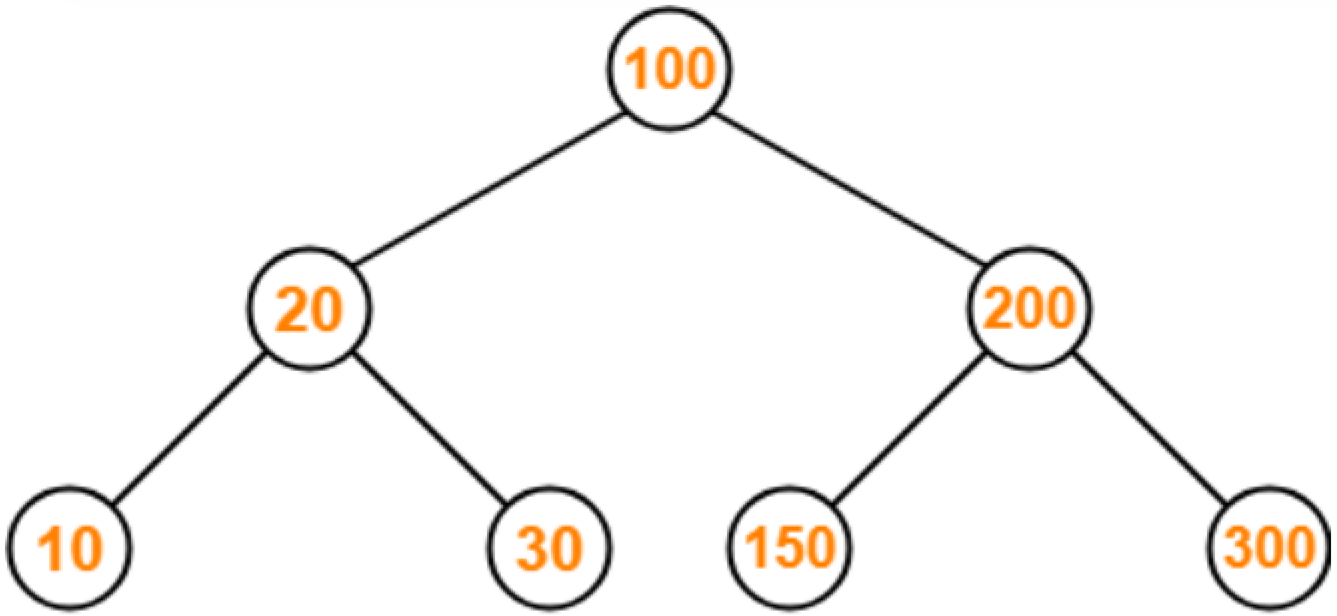
30, 20, 40, 15, 25, 35, 50, 5, 18, 45, 60

Inorder Example



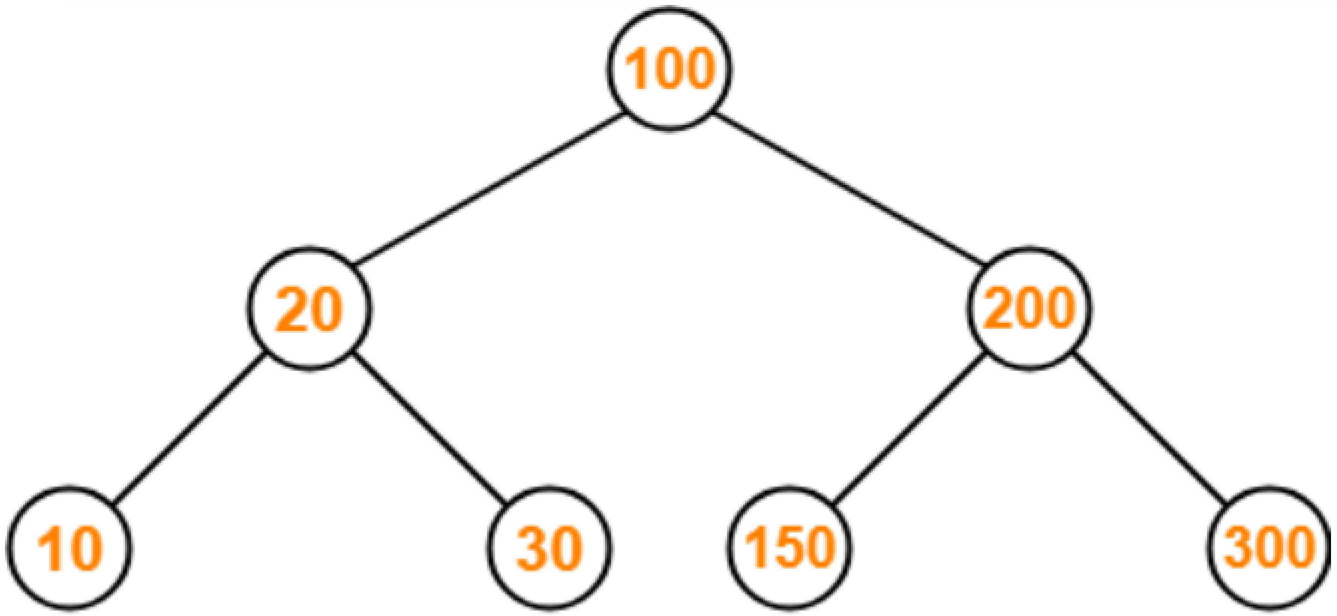
10 , 20 , 30 , 100 , 150 , 200 , 300

Preorder Example



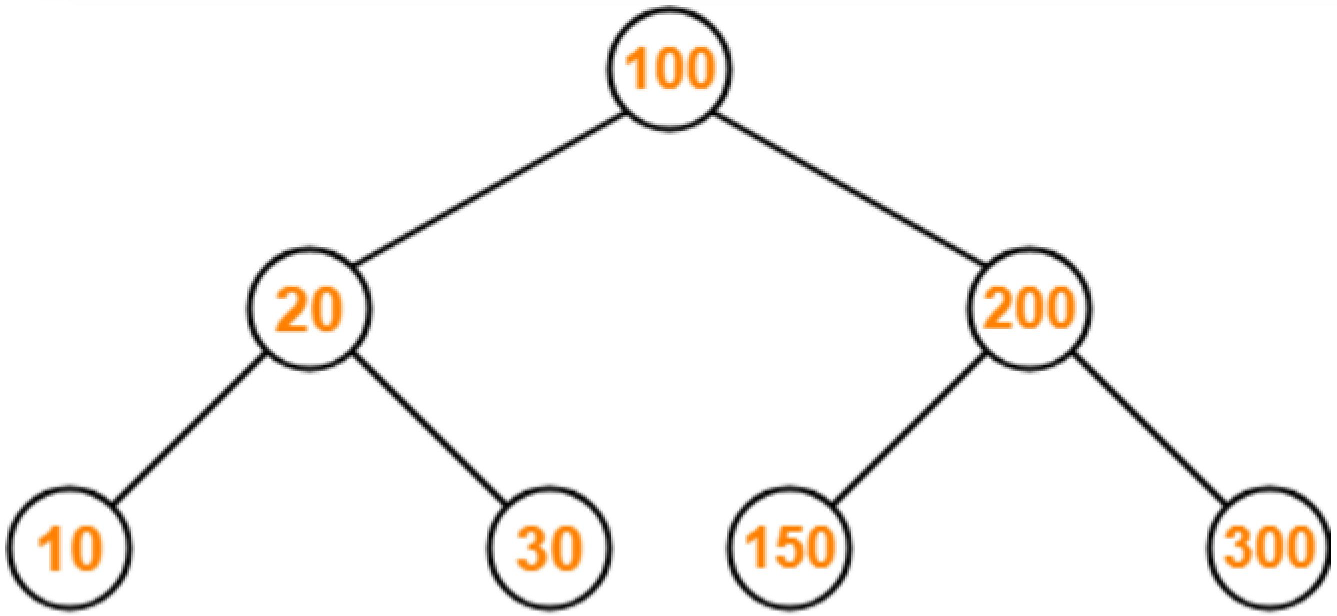
100 , 20 , 10 , 30 , 200 , 150 , 300

Postorder Example



10 , 30 , 20 , 150 , 300 , 200 , 100

Levelorder Example

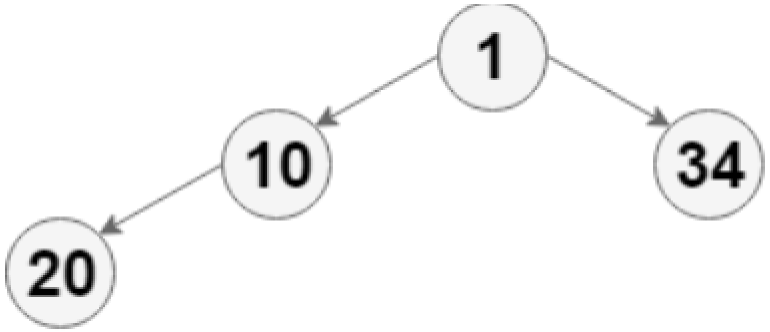


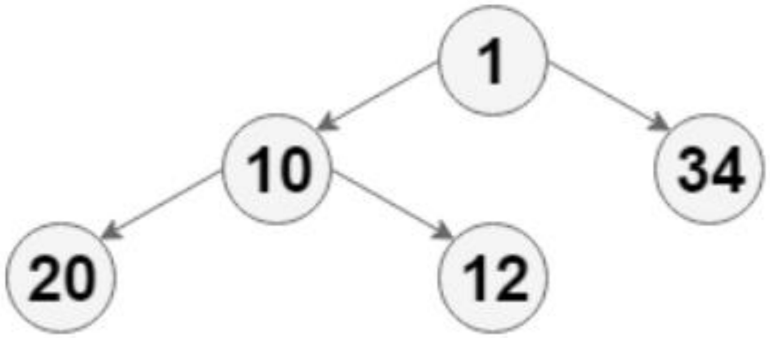
100 , 20 , 200 , 10 , 30 , 150 , 300

Insert - Binary trees

1) Create an empty queue q
2) Enqueue root
3) Loop while queue is not empty
   a) temp = queue.front
   b) dequeue
   c) if temp->left is null, insert the node as left child for temp and break, else enqueue temp->left
   d) if temp->right is null, insert the node as right child for temp and break, else enqueue temp->right

Insert - Binary trees



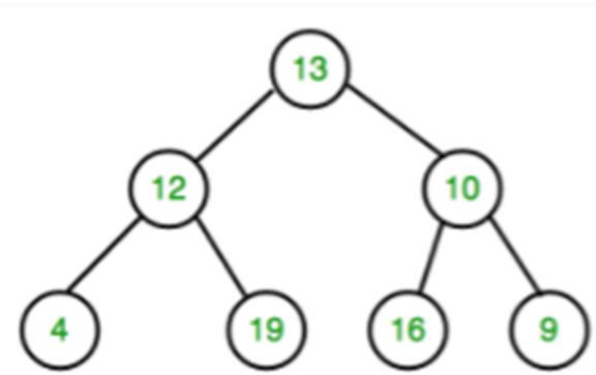


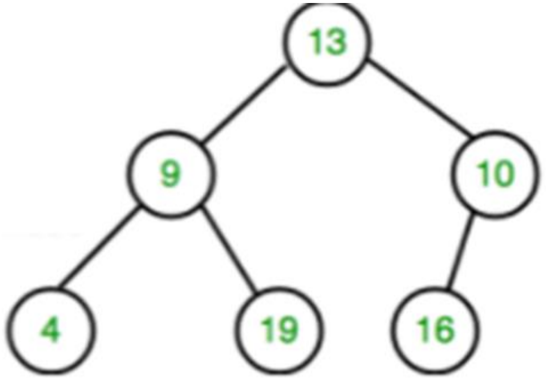
Insert 12

Delete – Binary trees

1. Start searching from the root, the address of node which is to be deleted by traversing in level order-wise.
2. Continue Traversing Tree to find the deepest and rightmost node in level order wise to find the deepest and the rightmost node.
3. If the node to delete is different from rightmost deepest node, then replace the node to be deleted with rightmost deepest node and delete the later node
4. If the node to delete is same as rightmost deepest node, then simply delete the node.

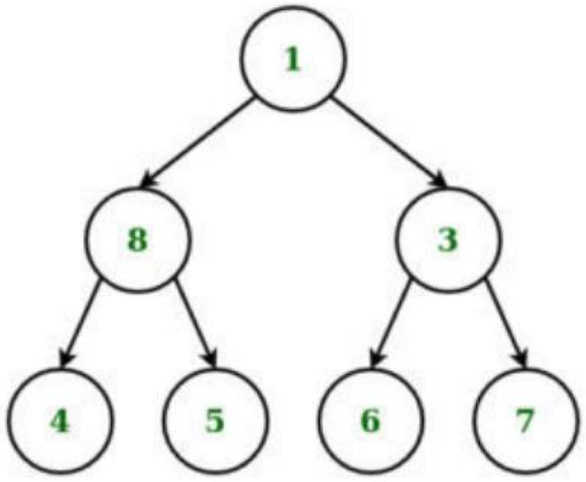
Delete – Binary trees

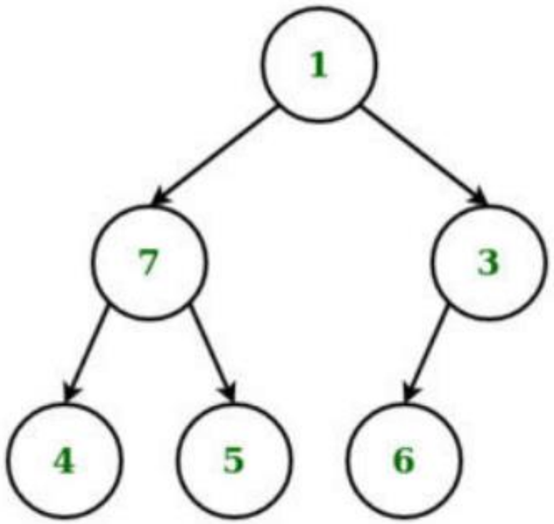




Delete 12

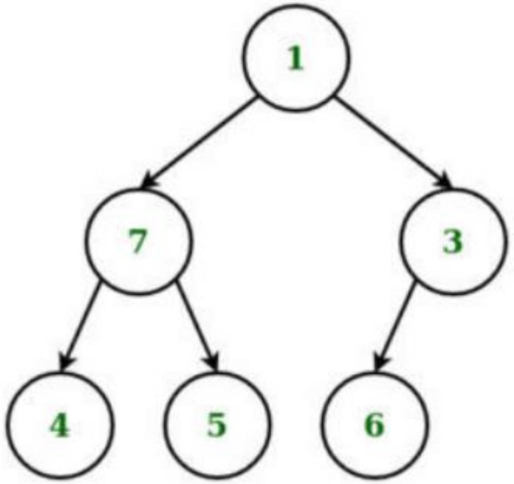
Delete – Binary trees

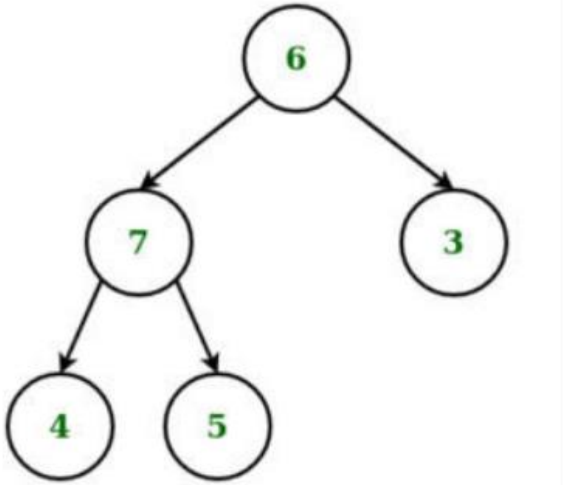




Delete 8

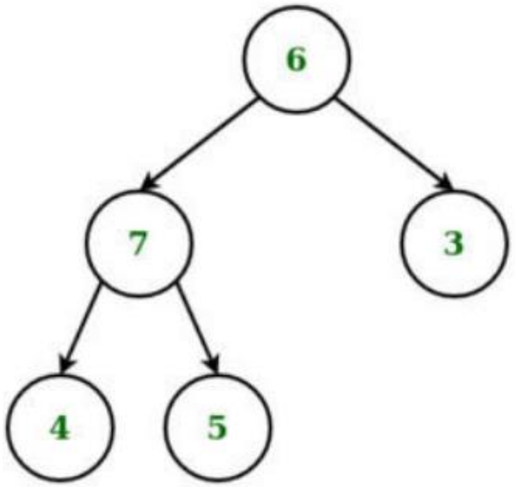
Delete – Binary trees

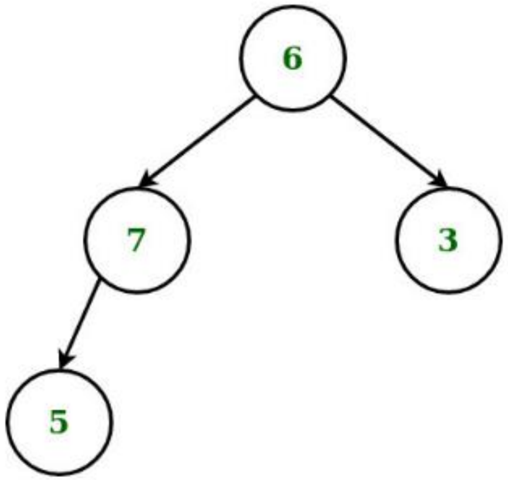




Delete 1

Delete – Binary trees





Delete 4