DATA STRUCTURES & ALGORITHMS

UNIT NO. 4

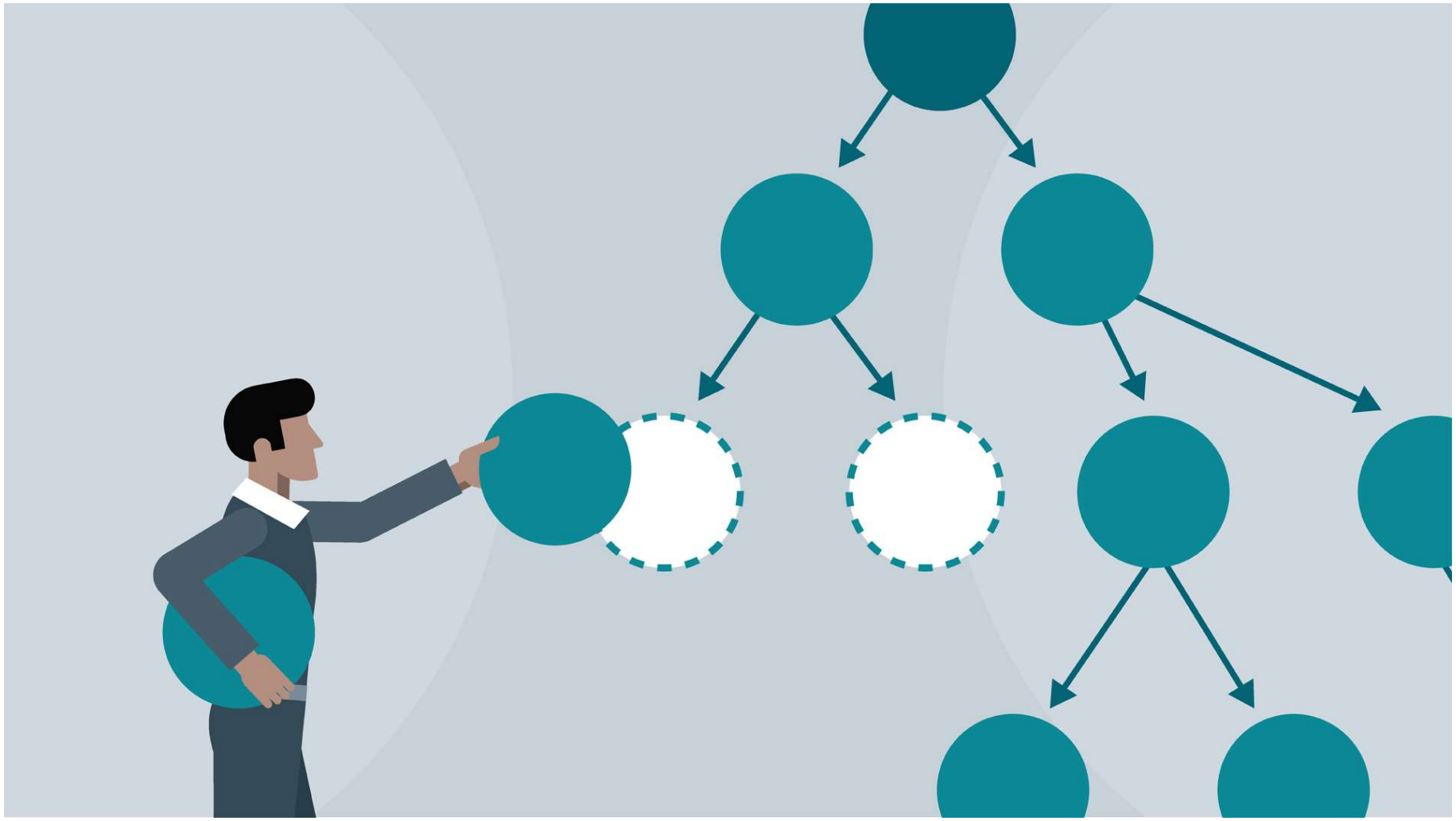
TREE

Topic Name

Introduction

By,
Prof. Dipika Birari

Tree



Tree : Introduction

- In linear data structure like Singly Linked List we can traverse in forward direction node by node.
- The doubly Linked List enhances the traversing by giving facility to traverse in both the directions : forward and backward.
- The Circular Linked List further provides more flexibility by allowing jumping from last node to first node.
- But in all these data structures, we cannot jump from a node to any desired node directly.

Tree : Introduction

- DS enhances the flexibility in traversing by providing the data structure Tree where one node is connected with several other nodes.
- Hence we can jump from one node to many other nodes.

Tree is a data structure in which one node is connected with several nodes and in turn these several nodes are connected with several other nodes.

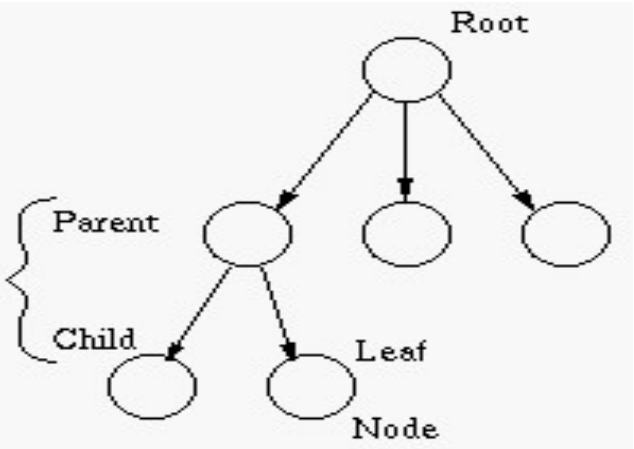
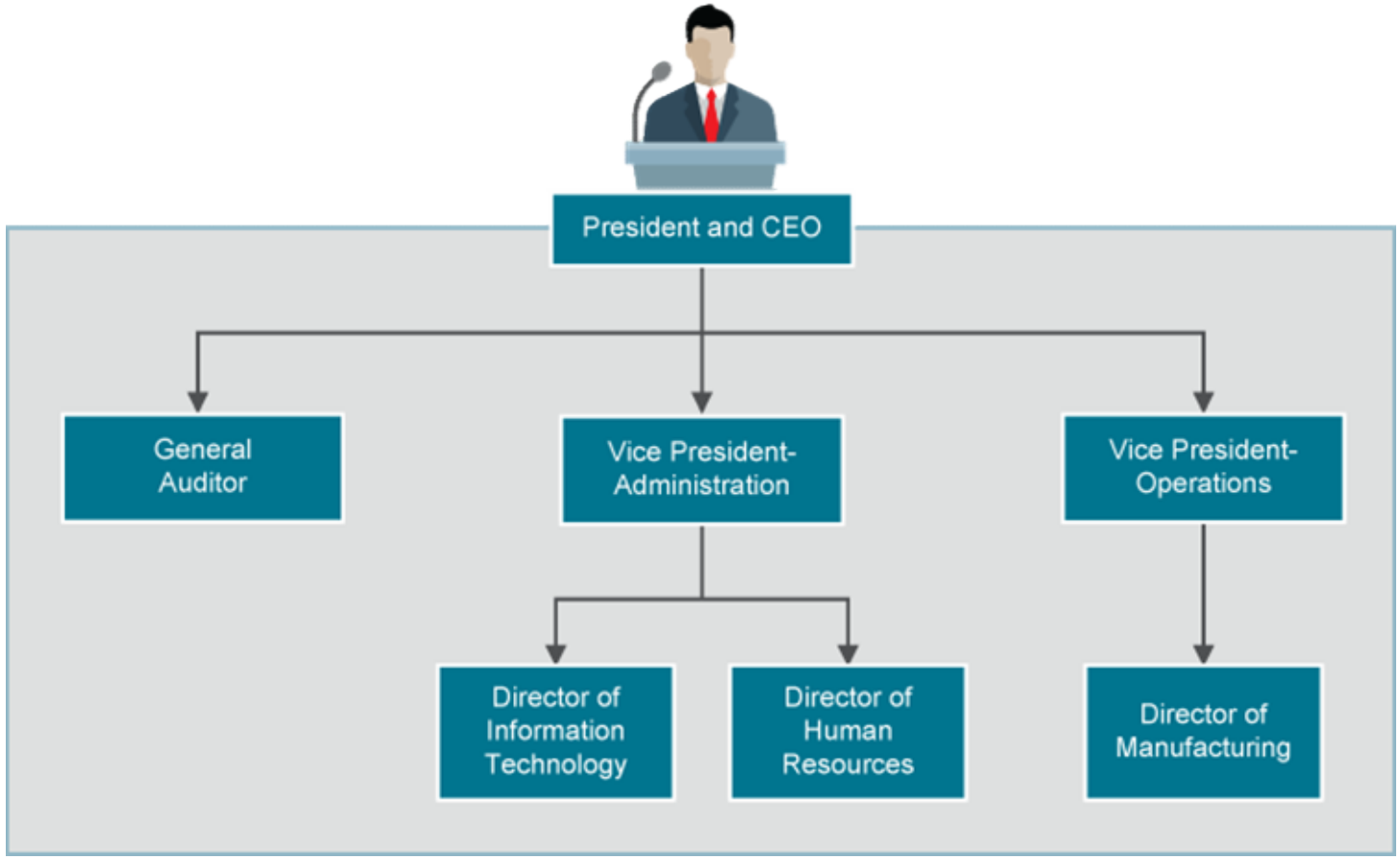


Figure: Tree data structure

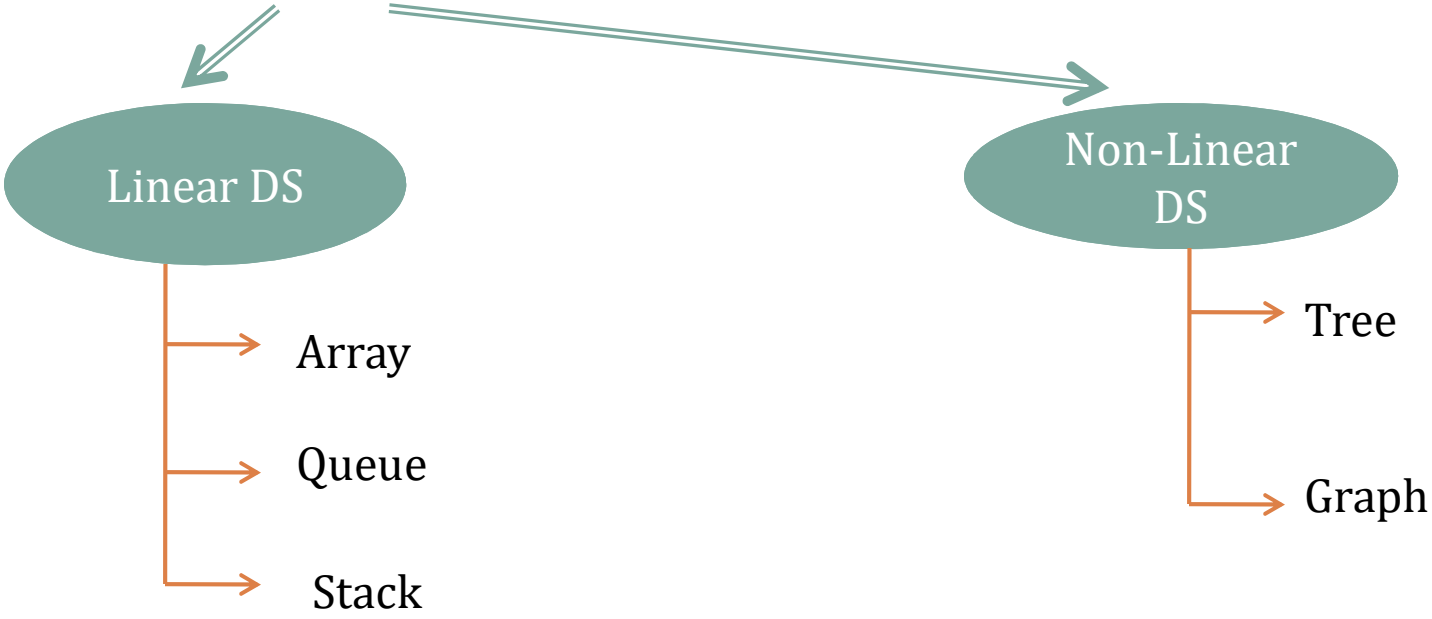
Tree : Introduction

- Tree is a very powerful as well as flexible data structure because of which there are several applications of tree.



Tree : Introduction

- **Data Structures**



A Linear data structure have data elements arranged in sequential manner and each member element is connected to its previous and next element.

A non-linear data structure has no set sequence of connecting all its elements and each element can have multiple paths to connect to other elements.

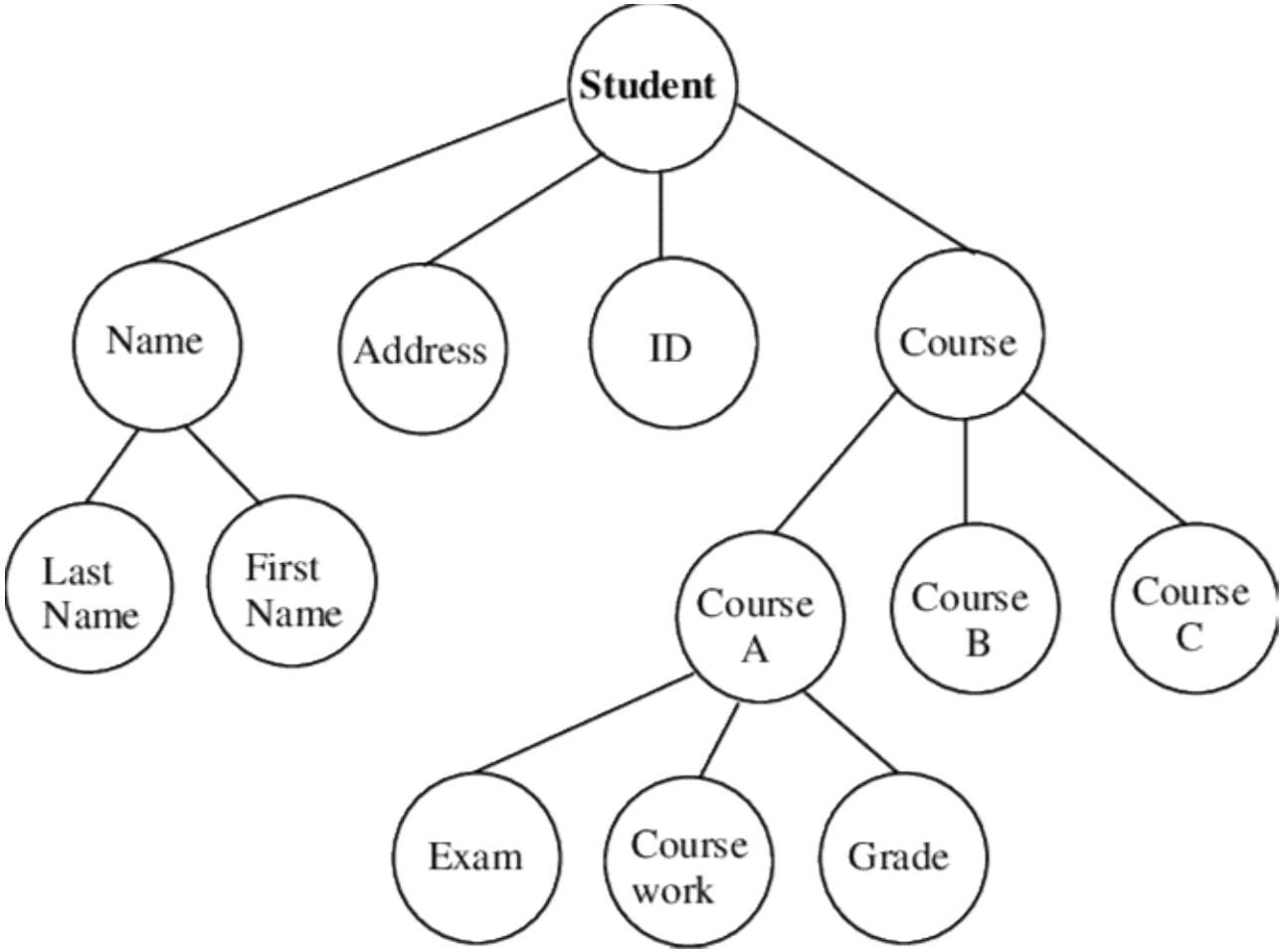
Tree : Definition

- The important use of Tree is representation of data which contains hierarchical relationship between elements,
- for example: records, family trees and table of contents.

Tree may be defined as a finite set 'T' of one or more nodes such that there is a node designated as the root of the tree and the other nodes are divided into n> = 0 disjoint sets
T1, T2, T3, T4 .... Tn.

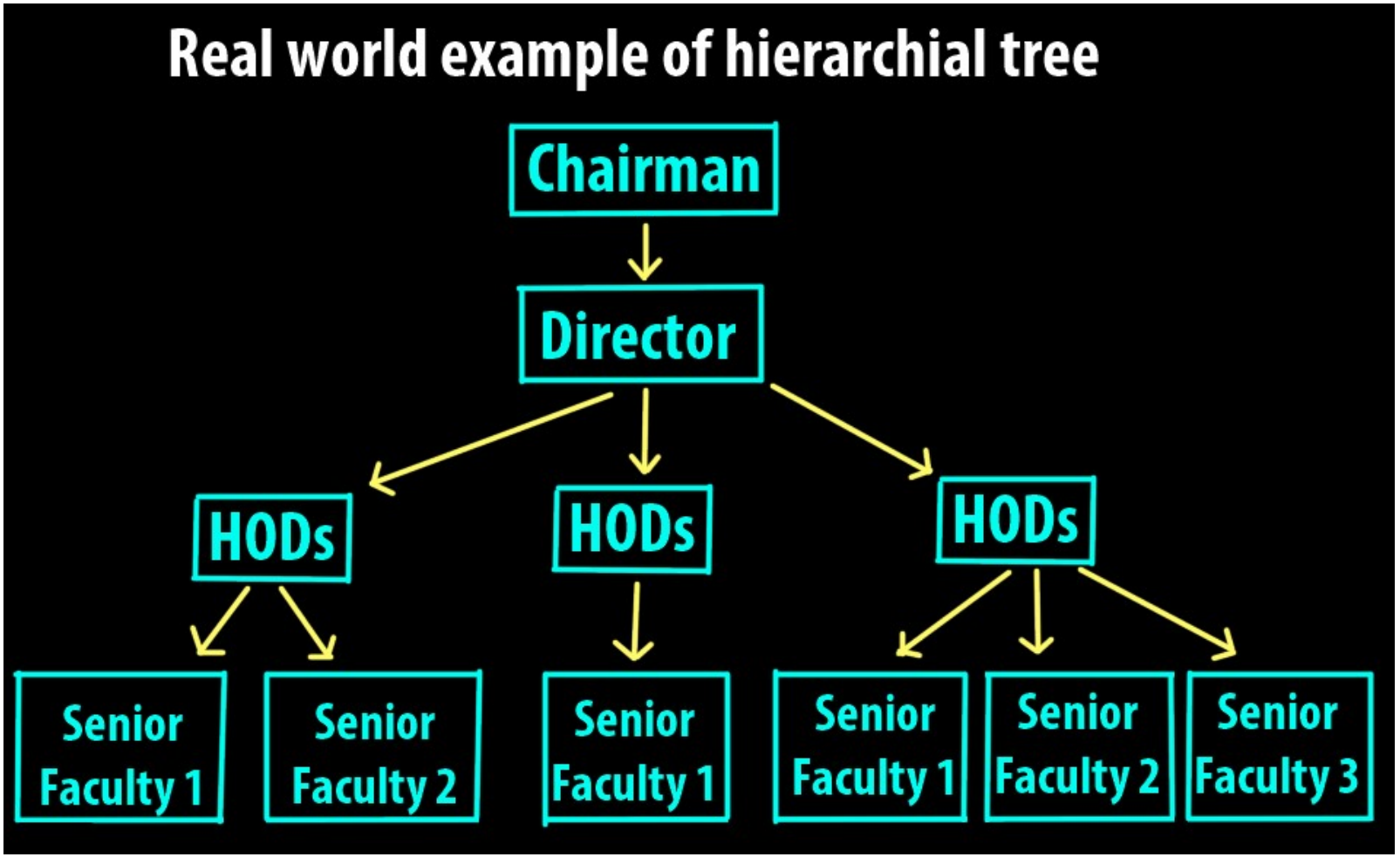
Where each of these set is a tree and
Set T1,....,Tn are called sub trees or children of the root.

Tree : Example



T = { V, E}
R c V
V → Vertex
E → Edge
R → Root

Tree : Example



Advantages of Tree

- Trees reflect structural relationships in between the records of data.
- Hierarchies can be represented by trees.
- Insertion and searching can be done efficiently in trees.
- Trees are very flexible data structures, allowing to move sub-trees around with minimum efforts.

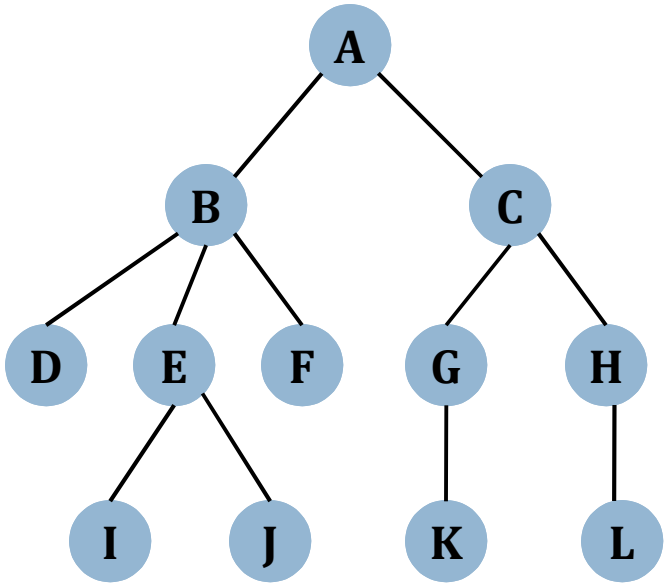
Basic Terminology

- Node
- Root
- Leaf Node
- Degree of Node
- Degree of Tree
- Level of a Node
- Depth/ Height of Tree
- Edge
- Path
- Parent
- Child
- Ancestor
- Descendents
- Siblings
- Left & Right Sub tree

Basic Terminology

Node

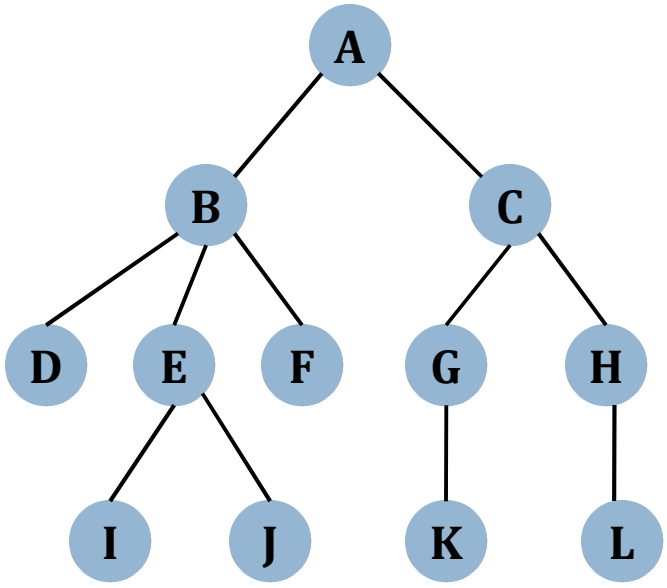
- Every element in the tree is called as **node** which contains data and links to other nodes.
- Here A, B, C, D, E, F, G, H, I, J, and L are nodes.
- In a tree, every individual element is called as **Node**.



Basic Terminology

Root

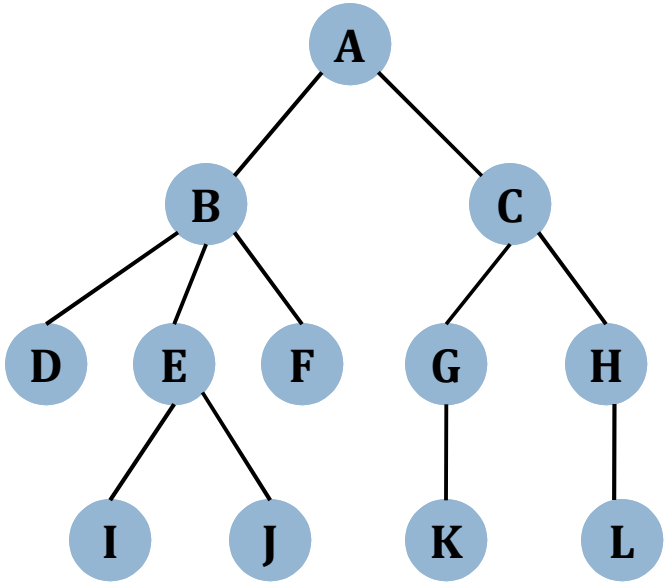
- The node which is placed at the top of the tree is called as root node.
- Every tree has only one root node.
- Here, node A is root node
- In any tree, the first node is called as root node.



Basic Terminology

Leaf Nodes

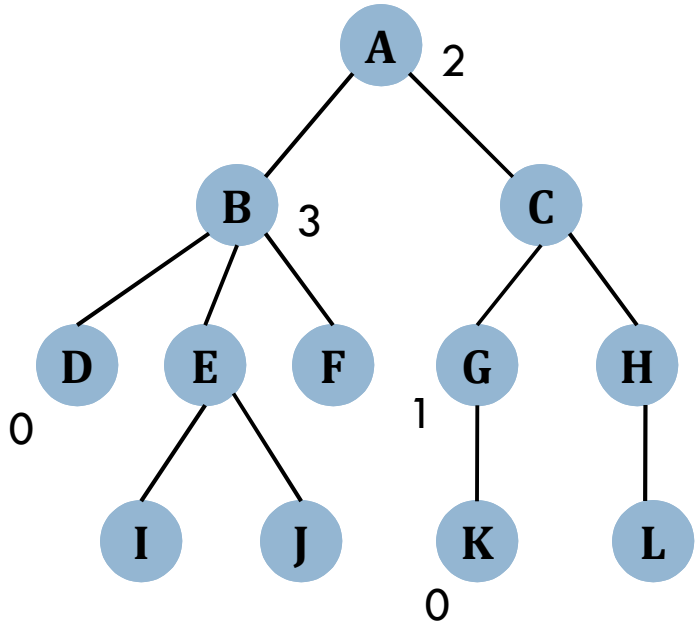
- The lower nodes which does not have any child nodes are called as **leaf nodes.**
- Here D, I, J, F, K and L are Leaf Nodes.
- A node without successor is called "**Leaf Node**".



Basic Terminology

Degree of Node

- The total number child nodes of any particular node is called as **degree of that node.**
- In any tree, '**Degree**' of a node is total numbers of children it as.
- Here, Degree of A=2
  Degree of B=3
  Degree of D=0
  Degree of G=1
  Degree of K=0

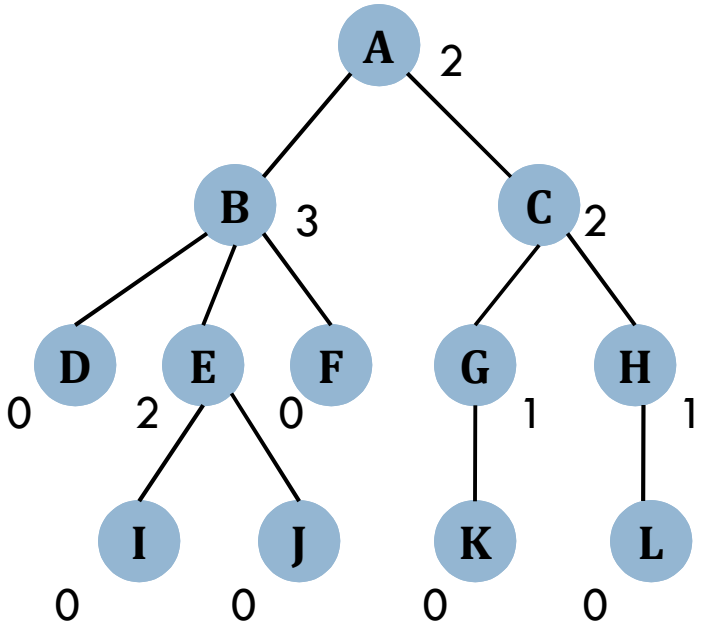


Basic Terminology

Degree of Tree

- **Degree of tree** is the degree of node/nodes in the tree having maximum degree.
- Here degree of B is more(i.e. 3)
- Hence it is a degree of tree.

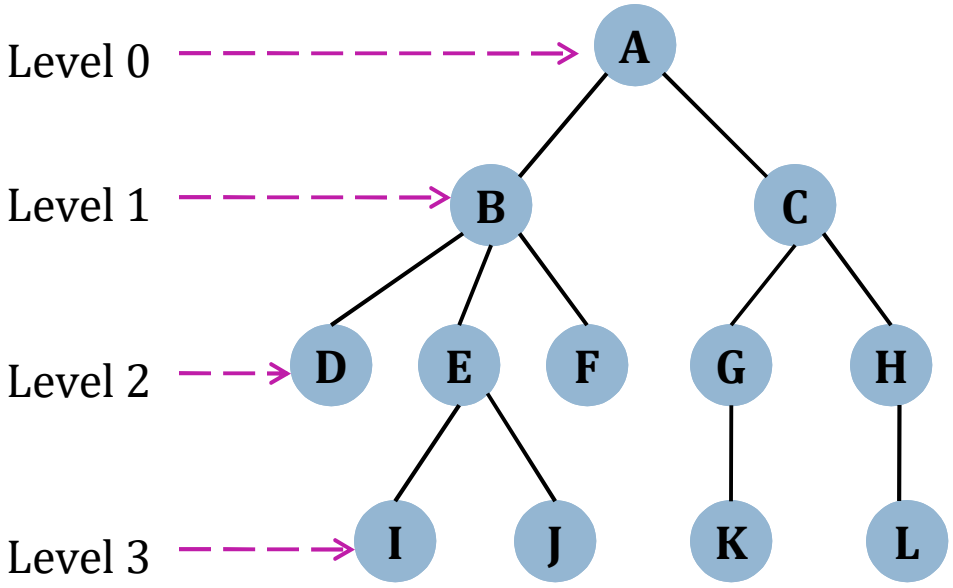
Degree of Tree=3



Basic Terminology

Level of a node

- The generation of node is represented by the term level.
- The level of root node = 0.
- The level of child of root = 1 and so on.
- The level of any node is one more than its parent.

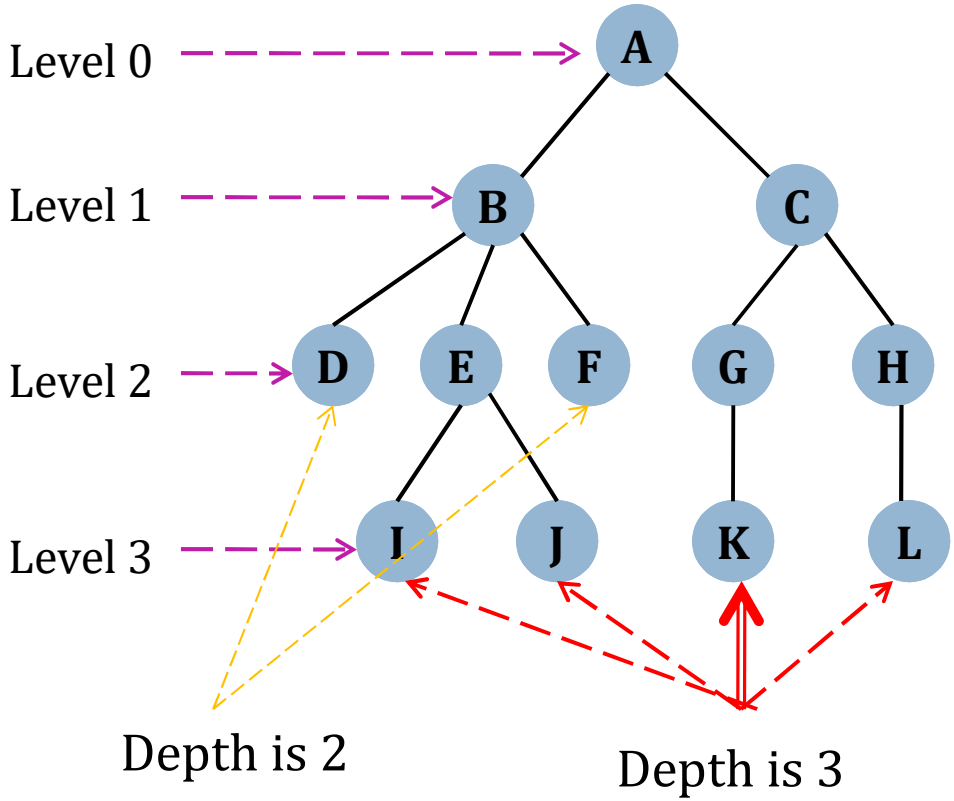


Basic Terminology

Depth/Height of a Tree

- The maximum level of any leaf node in the tree is called **depth or height of the tree.**
- Depth of Tree= 3

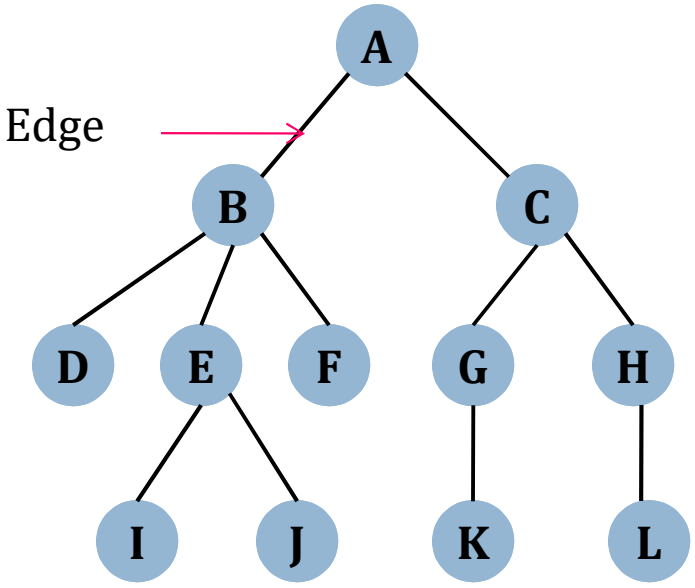
  OR
- Height of Tree=3



Basic Terminology

Edge

- The connecting link between any two nodes is called as **edge.**
- If in a tree there are n number of nodes,
- Then there will be n – 1 number of edges.

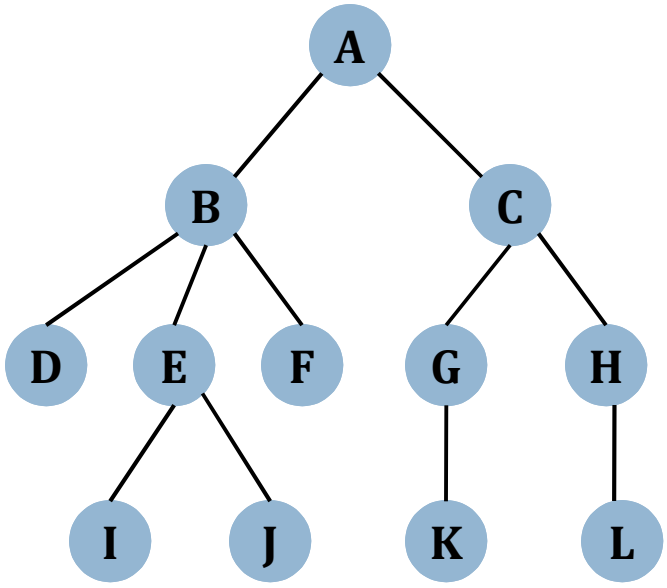


Basic Terminology

Path

- Sequence of edges from one node to another is called as **path** between those two nodes.
- Here path between A and J is

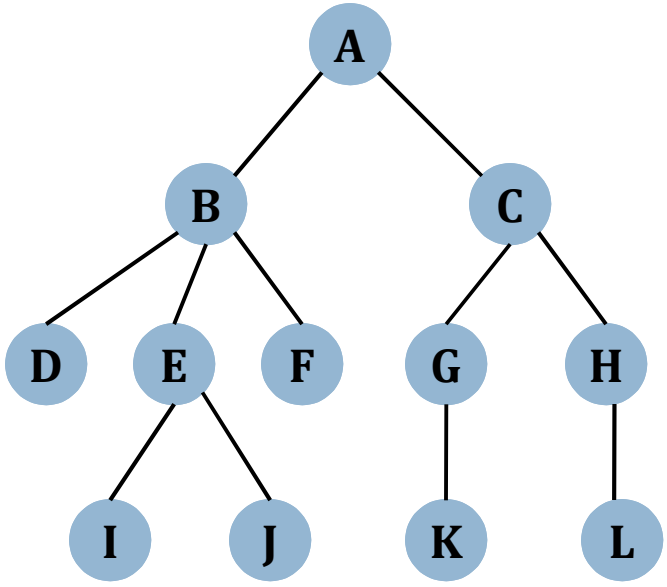
  A-B-E-J



Basic Terminology

Parent

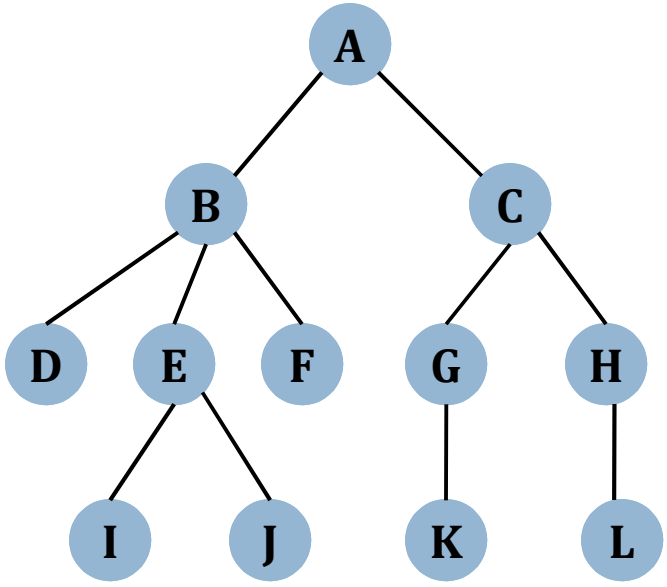
- The node which has child node is called as **parent.**
- This node is predecessor of other node/nodes.
- E.g. A, B, C, E, G and H



Basic Terminology

Child

- The node which has a link from its parent node is known as **child** node.
- This node is successor of other nodes
- E.g. B and C → Child of A

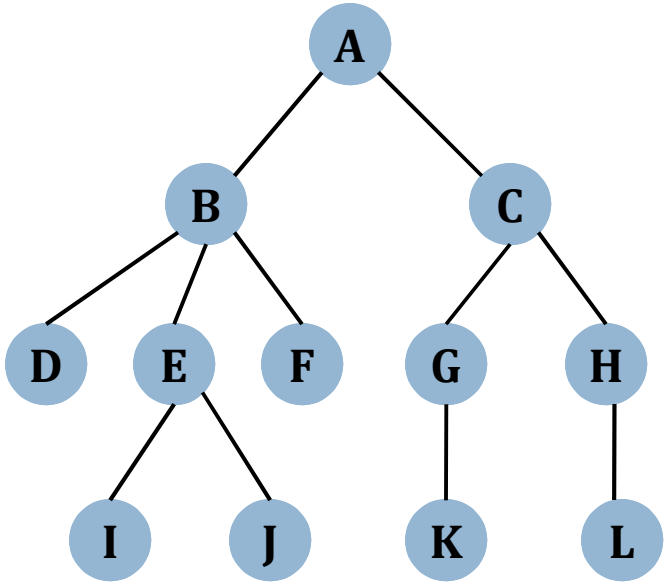


Basic Terminology

Child

- The node which has a link from its parent node is known as **child** node.
- This node is successor of other nodes
- E.g. B and C → Child of A

  G and H → Child of C



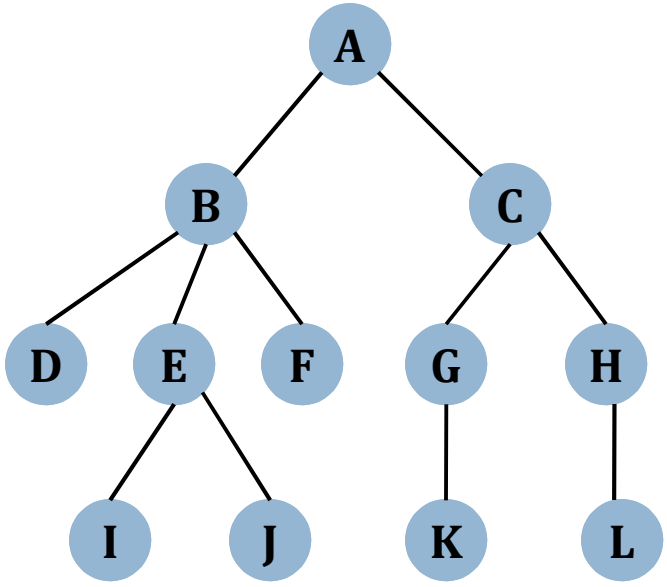
Basic Terminology

Child

- The node which has a link from its parent node is known as **child** node.
- This node is successor of other nodes
- E.g. B and C → Child of A

  G and H → Child of C

  D, E & F → Child of B

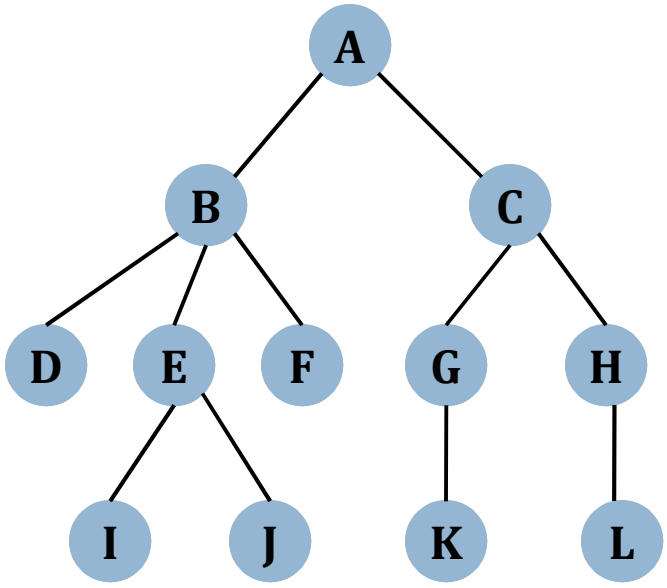


Basic Terminology

Ancestor

- The ancestors of a node are all the nodes along the path from the root to that node.
- Here,

  B and A → Ancestor of E



Basic Terminology

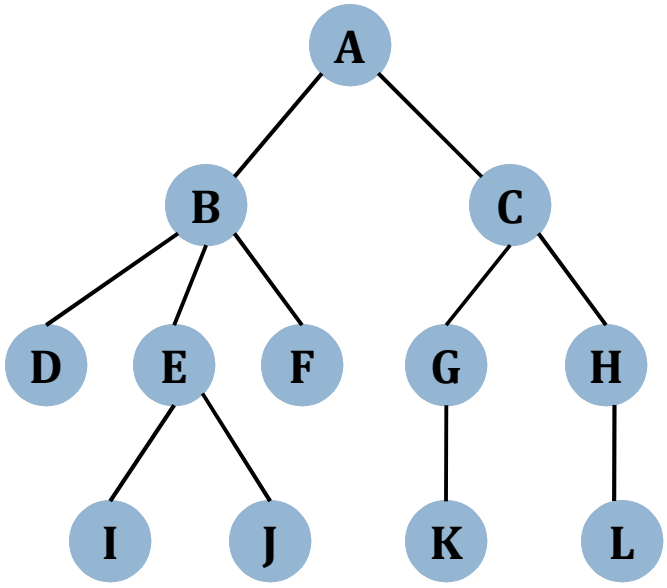
Ancestor

- The ancestors of a node are all the nodes along the path from the root to that node.

- Here,

  B and A → Ancestor of E

  C and A → Ancestor of G



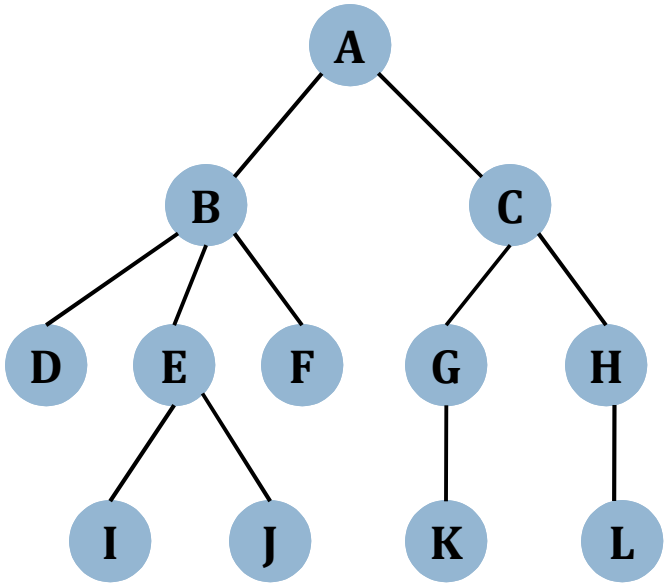
Basic Terminology

Descendents

- Descendents of a node are all such nodes in downward direction which are reachable from that node..

- Here,

  G, H, K and L → descendents of C

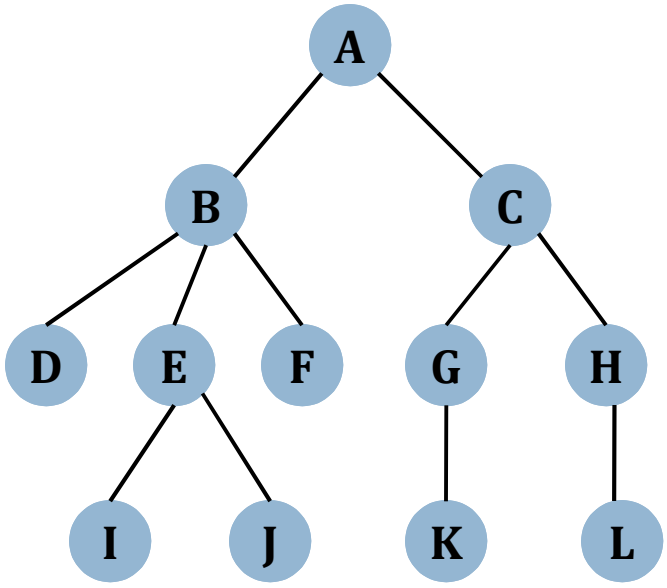


Basic Terminology

Siblings

- The nodes which belong to same Parent are known as Siblings.

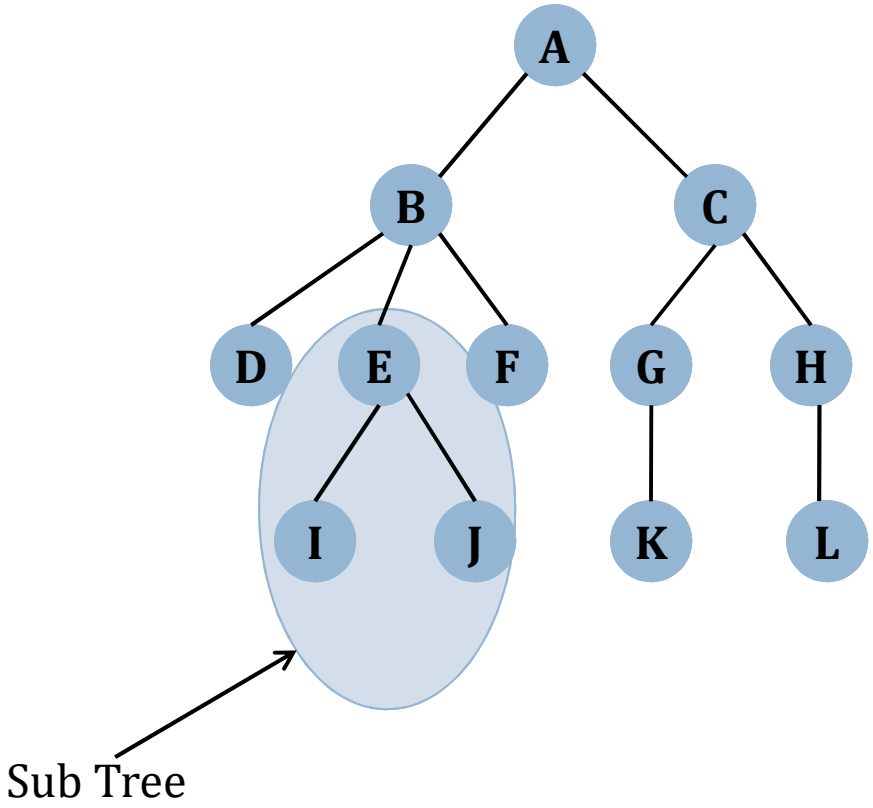
- Here,
  - B and C are siblings.
  - D, E, & F are siblings.
  - G and H are siblings.
  - I and J are siblings.
  - K and L are siblings.



Basic Terminology

Left and Right Sub tree

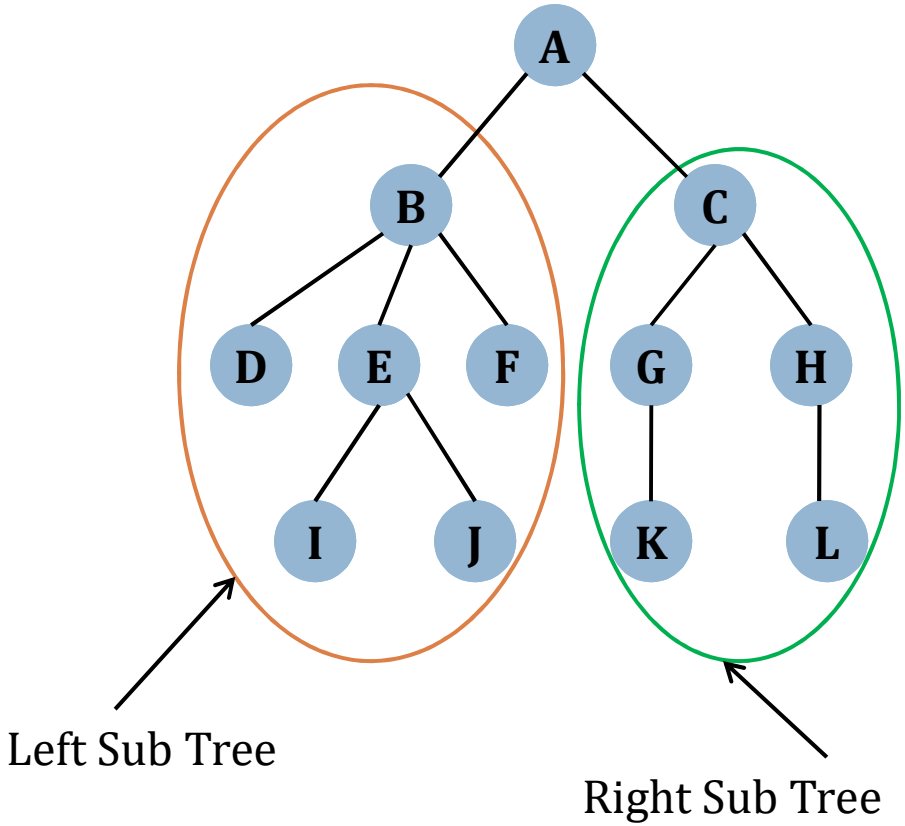
- In a tree, every child from a node forms a subtree recursively.
- Every child node will form a subtree on its parent node.



Basic Terminology

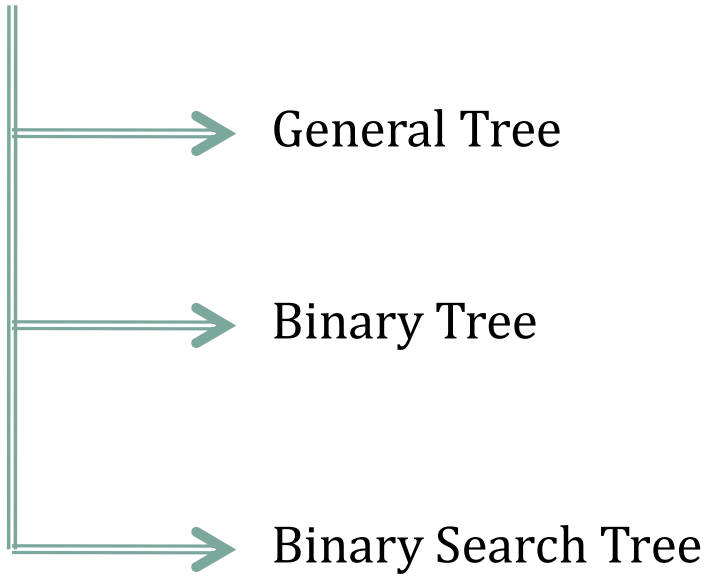
Left and Right Sub tree

- In a tree, every child from a node forms a subtree recursively.
- Every child node will form a subtree on its parent node.
- Sub-tree at left side is called as **Left sub-tree** and subtree at right side is called as **right sub-tree.**



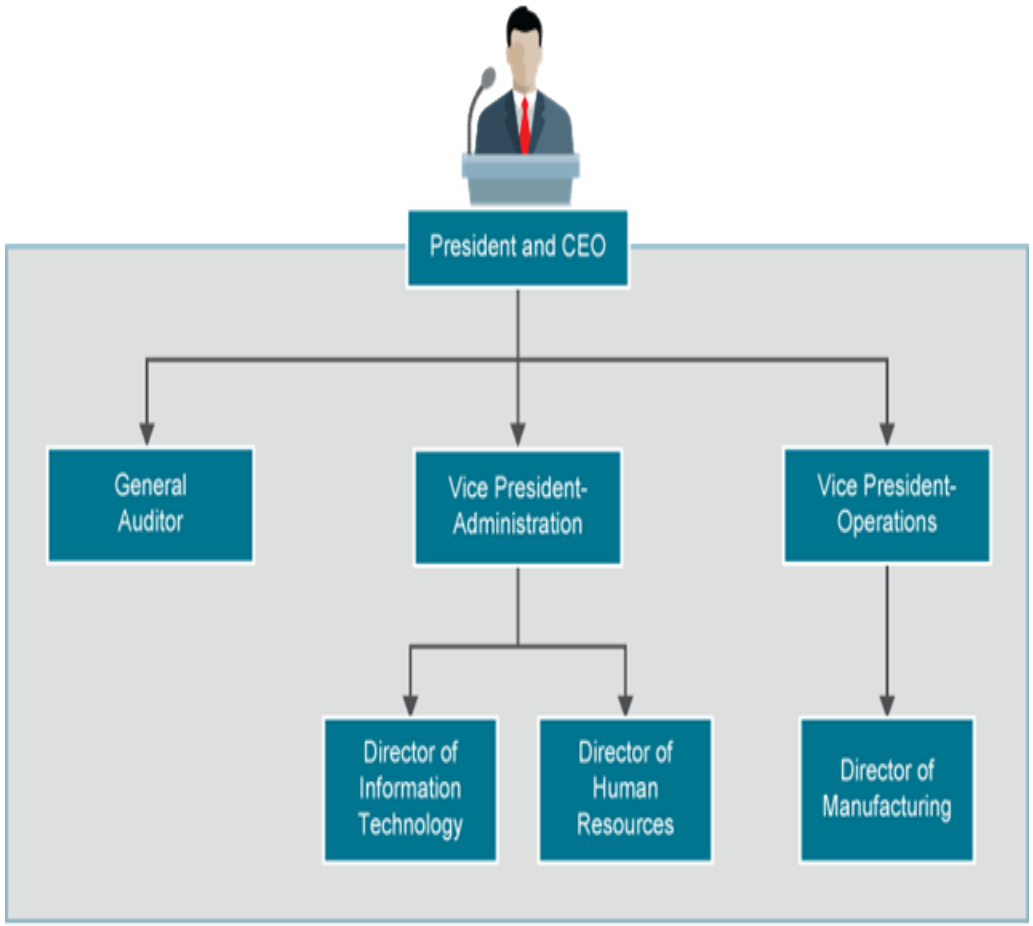
Tree Types

- Following types of trees:



General Tree

- Usually maximum of organizations are hierarchical in nature.

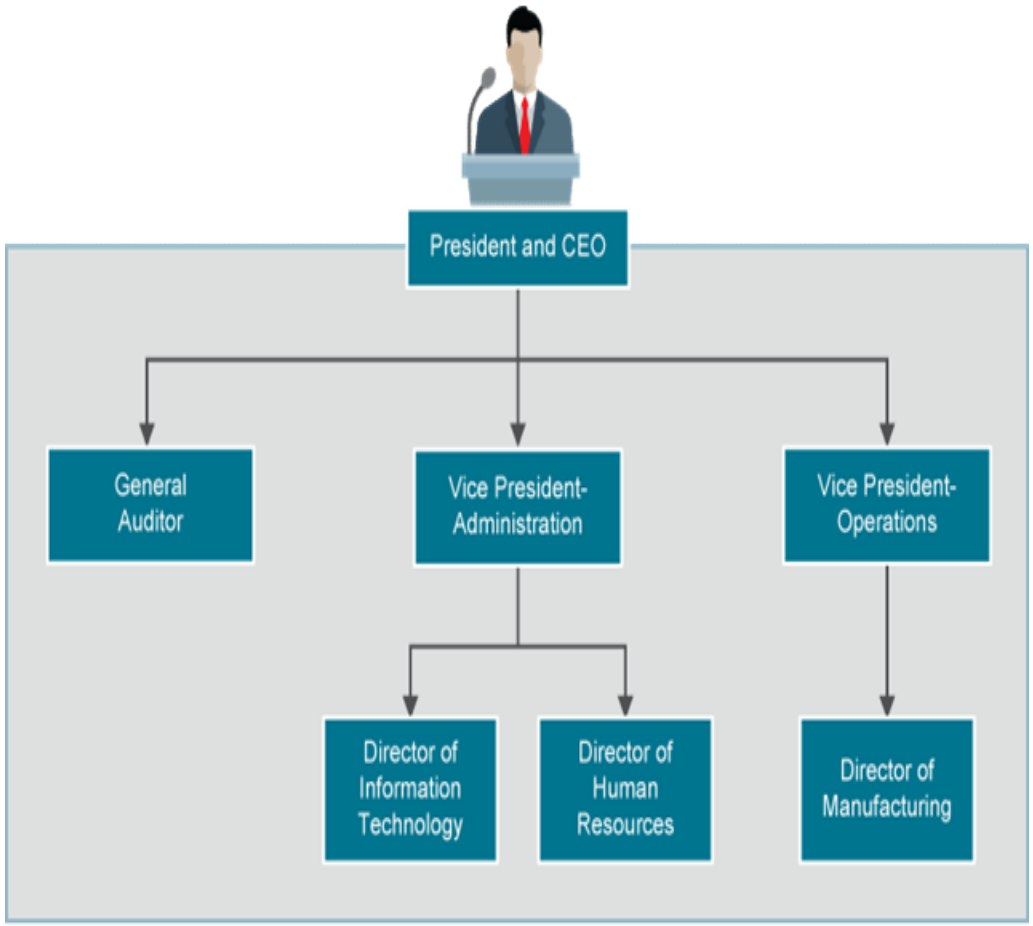


When there is need to model this organization with the help of a data structure,

- President → Root Node
- Vice President → level 1
- Directors → level 2

General Tree

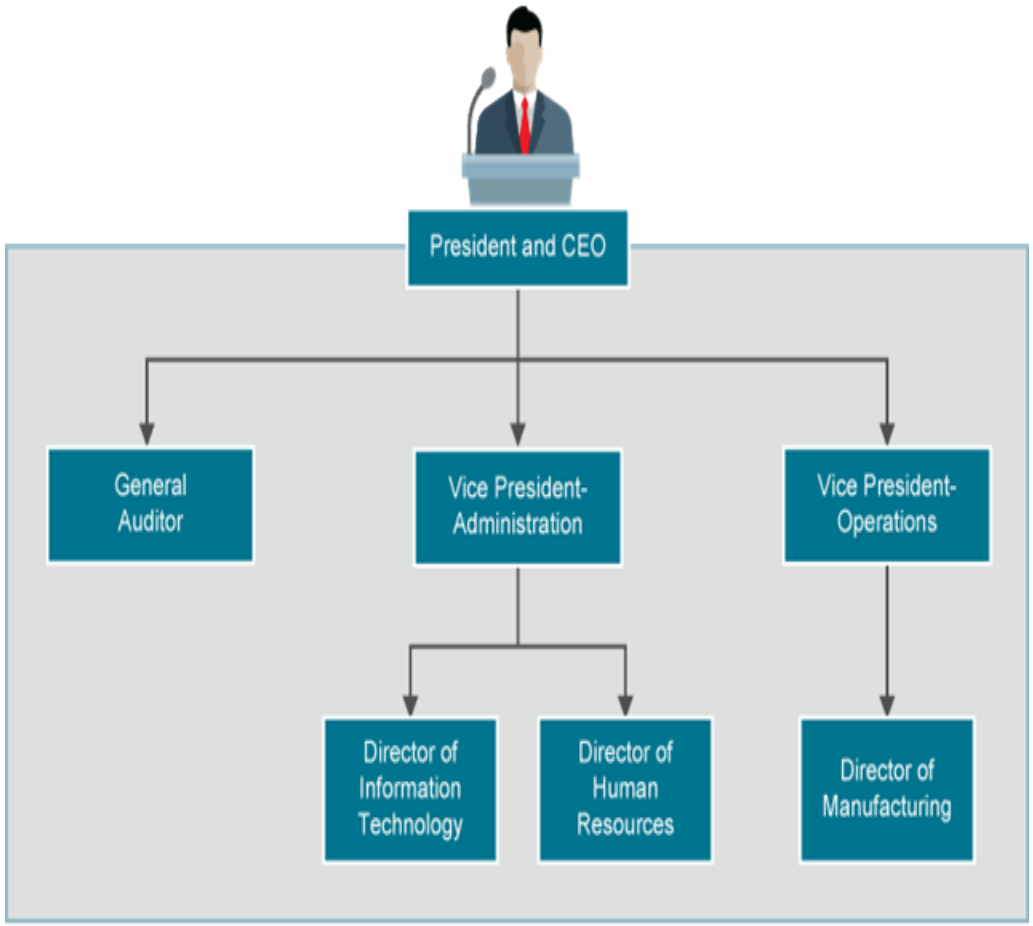
- Usually maximum of organizations are hierarchical in nature.



As the number of vice presidents or their subordinates may be more than two, there is need of such a tree structure whose nodes can have an arbitrary number of children. Such a tree is called as **general tree.**

General Tree

- Usually maximum of organizations are hierarchical in nature.

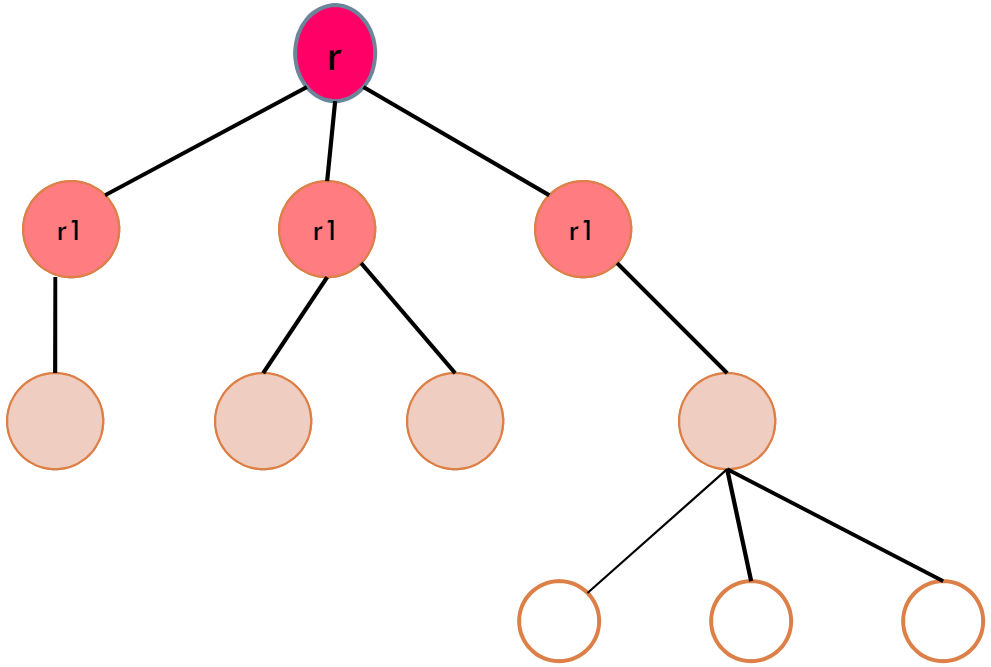


• But the important factor is that when nodes in a tree can have multiple children, it becomes much harder to implement such tree.

• The general tree is the tree in which all the nodes can have any number of child nodes without any restriction.

General Tree

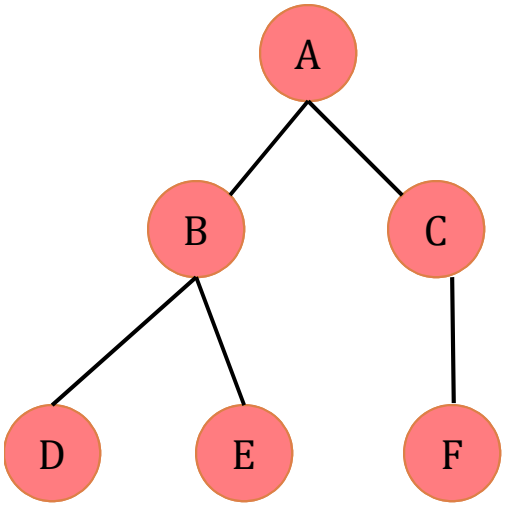
A general tree T is a finite set of one or more nodes such that there is one designated node r, called the root of T, and the remaining nodes are partitioned into n ≥ 0 disjoint subsets T1, T2, …, Tn, each of which is a tree, and whose roots r1, r2, …, rn, respectively, are children of r.



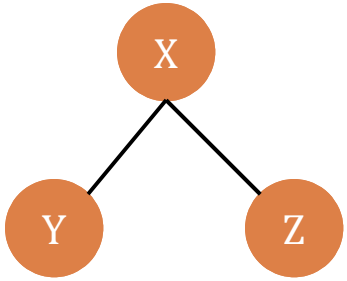
Forest

Forest is an arbitrary set of trees.

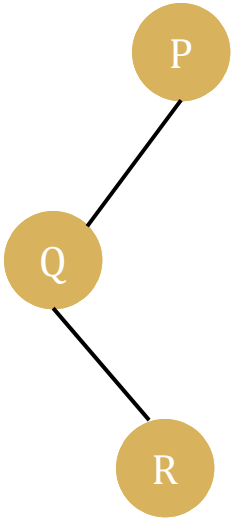
In other words, we can consider that a general tree as the root of a forest, and a forest is an ordered combination of zero or more general trees.



T1



T2

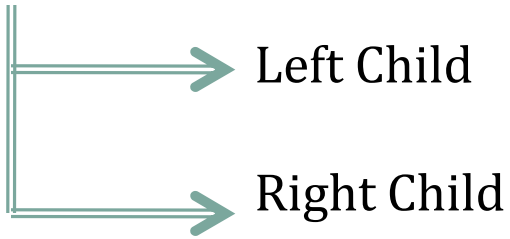


T3

Binary Tree

- In a normal or general tree, there may be any number of children to each node which make it complicated to implement such tree.

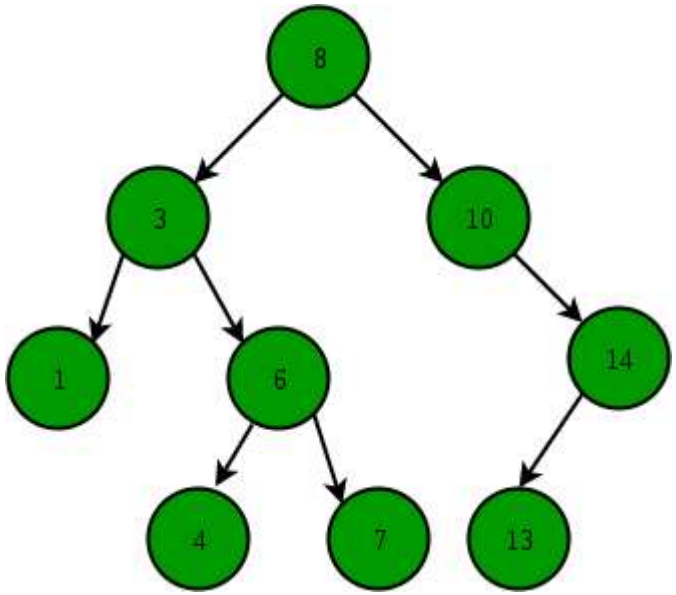
- Binary tree is a specialized form of tree data structure in which every node is allowed to have maximum of two child nodes.



Binary Tree

A tree in which every node can have maximum of two child nodes is known as Binary Tree.

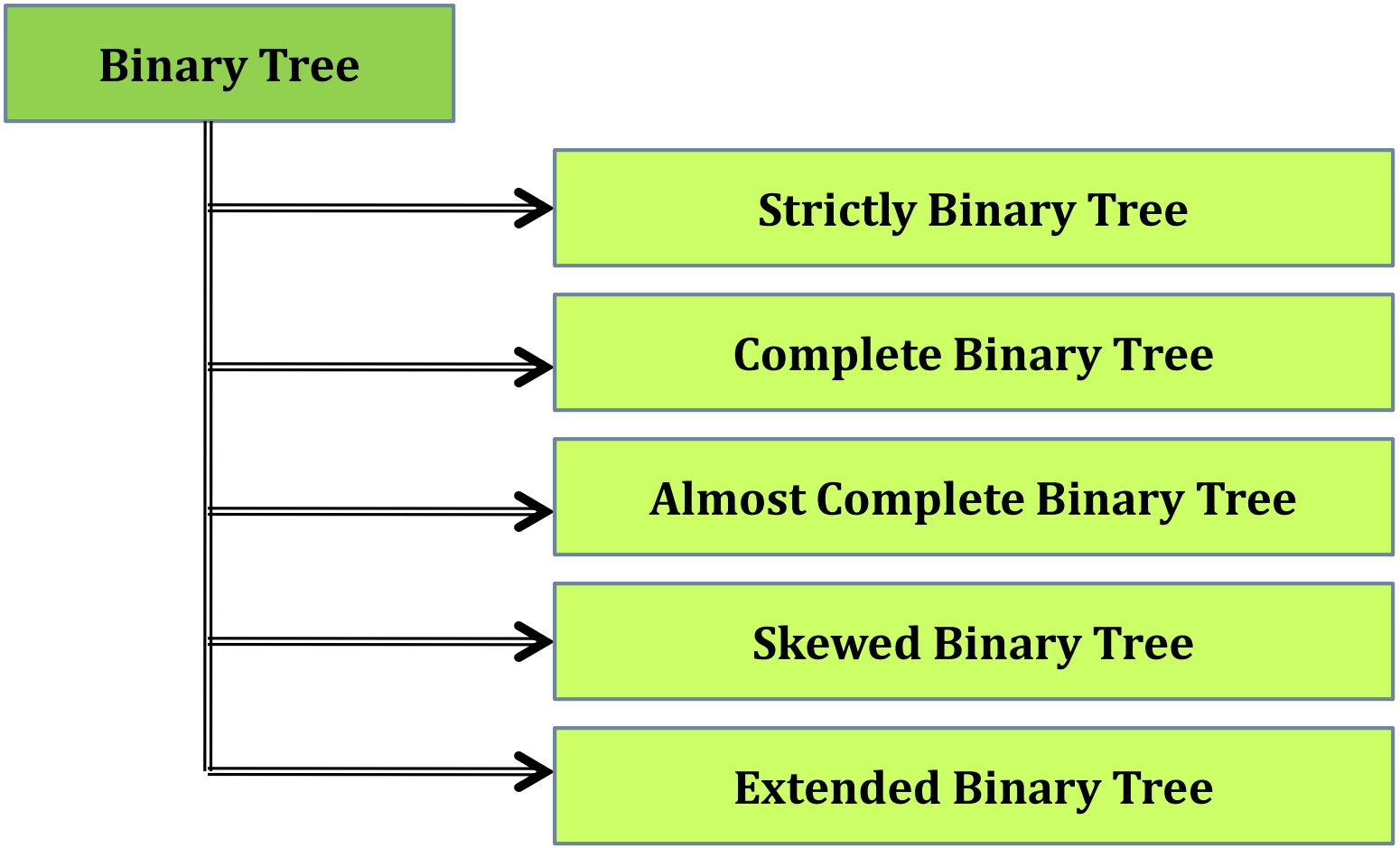
- In a binary tree, each and every node can have either 0, 1 or 2 children but not more than 2.



Applications of Binary Tree

- Manipulate hierarchical data.
- Make information easy to search.
- Manipulate sorted lists of data.
- As a workflow for compositing digital images for visual effects.
- Router algorithms

Types of Binary Tree



Types of Binary Tree

Strictly Binary Tree

- In Binary Tree → Every Node have → Maximum two child nodes.

- In Strictly BT → Every node have → Exactly two child nodes.

Only leaf nodes are skipped from this rule.

A binary tree in which every node has either two or zero number of child nodes is called Strictly Binary Tree.

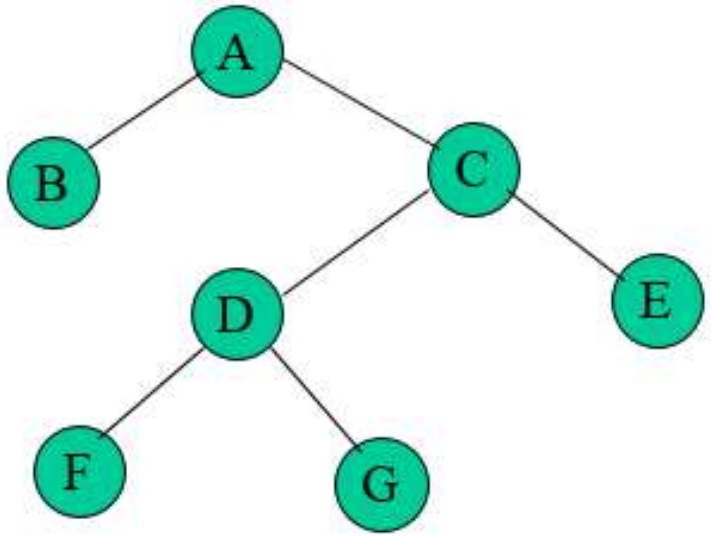
- Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree.

Types of Binary Tree

Strictly Binary Tree

A binary tree in which every node has either two or zero number of child nodes is called Strictly Binary Tree.

- Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree.



Types of Binary Tree

Complete Binary Tree

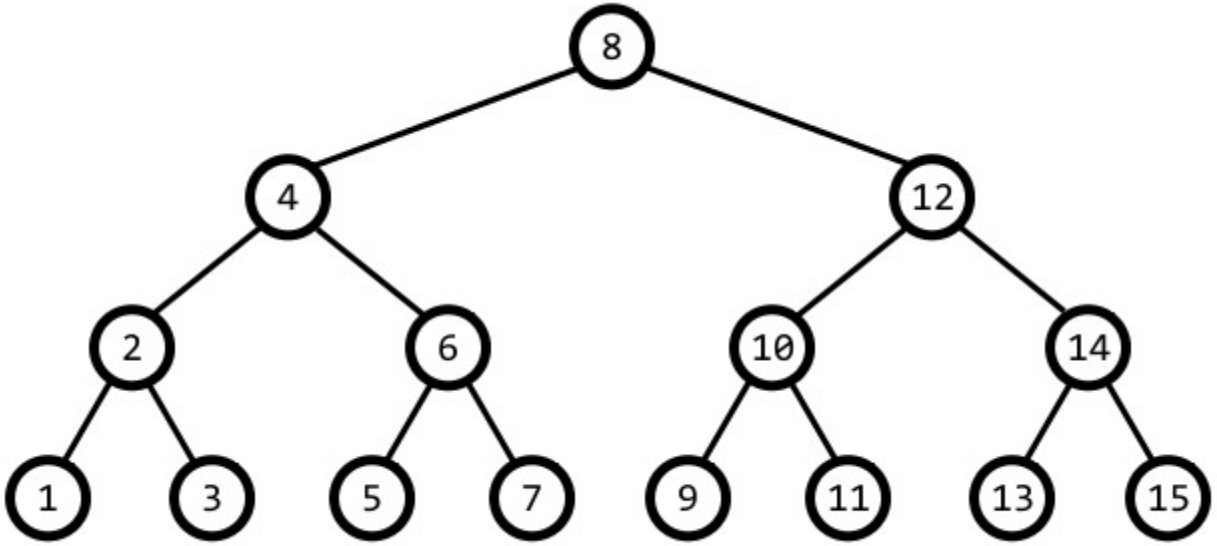
- In Binary Tree → Every Node have → Maximum two child nodes.
- In Strictly BT → Every node have → Exactly two child nodes.
- In complete BT → All node have → Exactly two child nodes.

  Every level must have →2 ^ level no. of nodes.
- Example:
  - At level 2 → 2^2=4 nodes
  - At level 3 → 2^3=8 nodes

Types of Binary Tree

Complete Binary Tree

A binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at same level is called Complete Binary Tree.

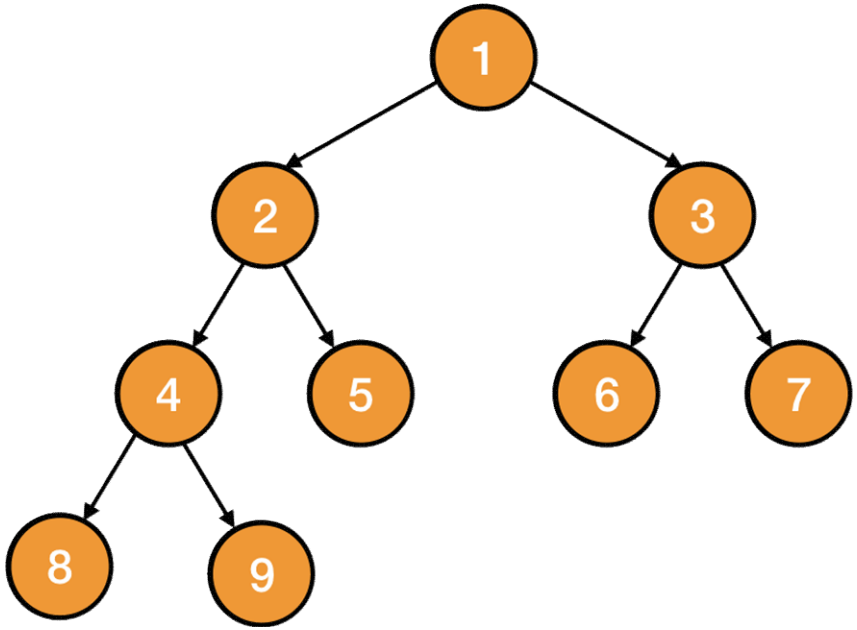


Types of Binary Tree

Almost Complete Binary Tree

- Here the last level is partially filled.

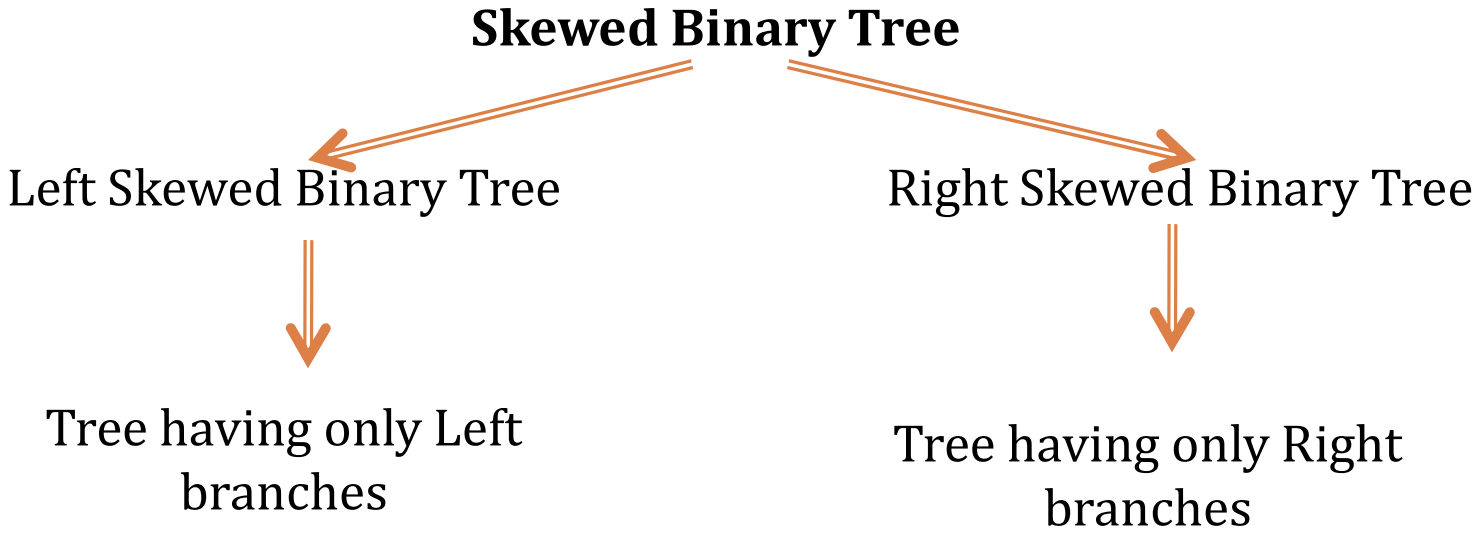
A binary tree with L levels, having all levels 1 to L-1 are completely filled without any gaps and last level L is partially filled from left to right, is as **Almost Complete Binary Tree.**



Types of Binary Tree

Skewed Binary Tree

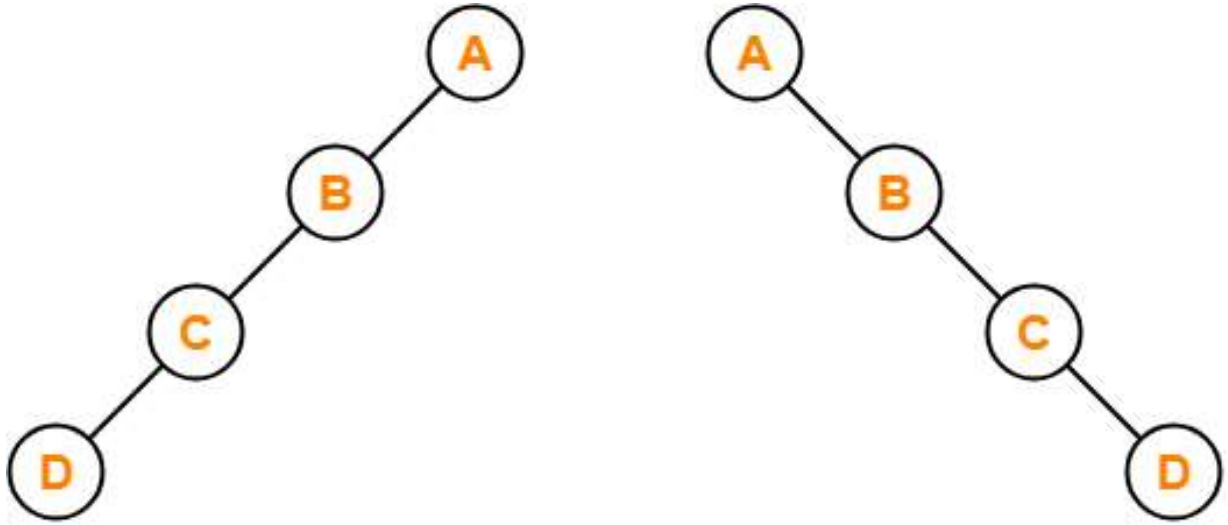
The binary tree in which either only left branches are present or only right branches are present is called as **Skewed Binary Tree.**



Types of Binary Tree

Skewed Binary Tree

The binary tree in which either only left branches are present or only right branches are present is called as **Skewed Binary Tree.**



Left Skewed Binary Tree

Right Skewed Binary Tree

Types of Binary Tree

Extended Binary Tree

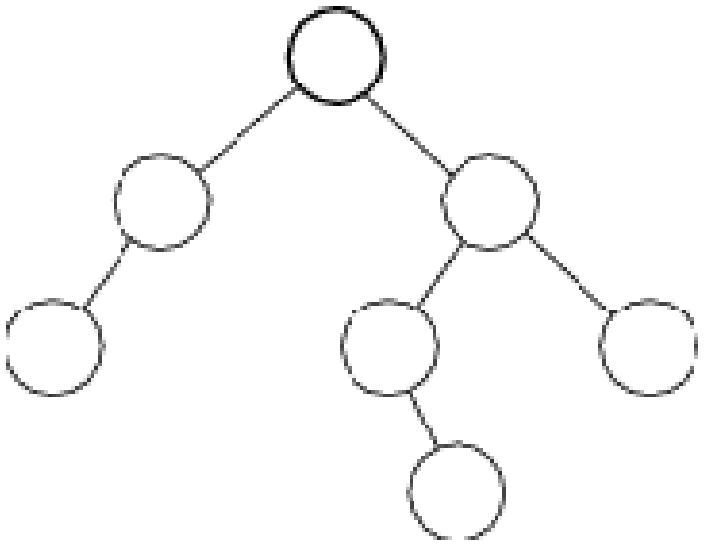
- Binary Tree —(Converted to / By inserting some dummy nodes)→ Full Binary Tree.

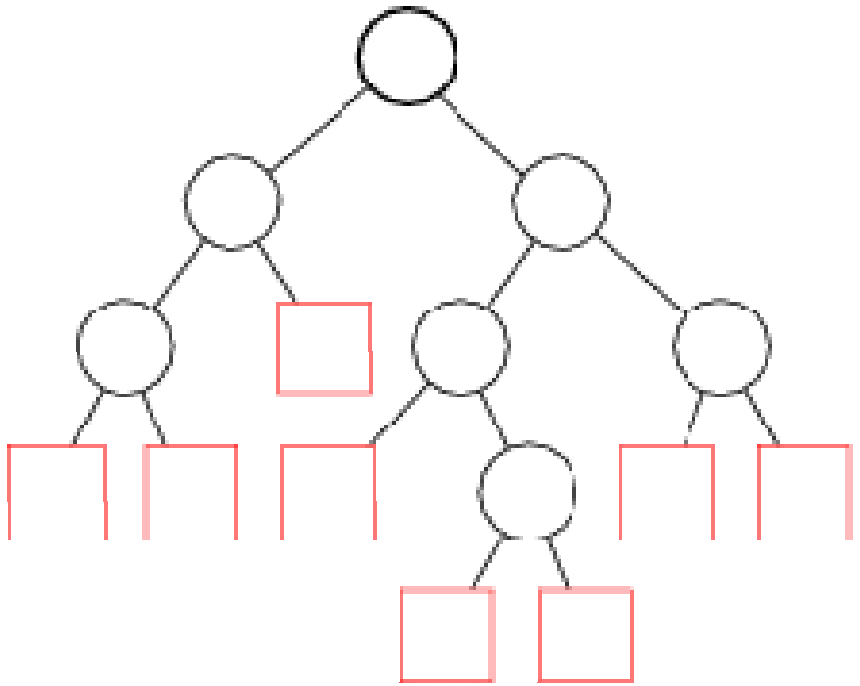
The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.

Types of Binary Tree

Extended Binary Tree

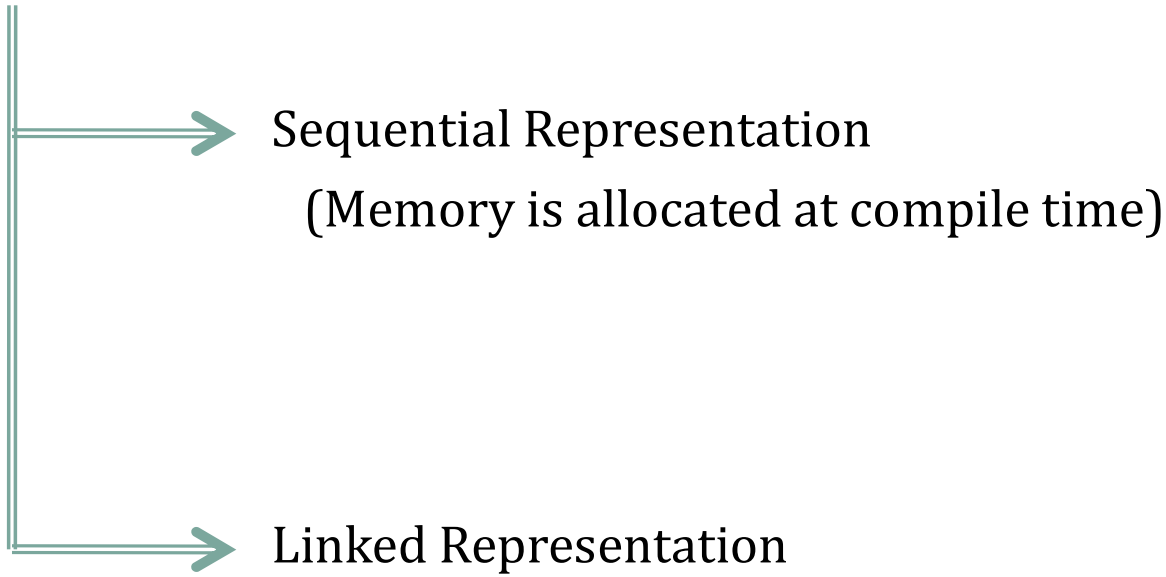
- Binary Tree —(Converted to / By inserting some dummy nodes)→ Full Binary Tree.





Representation of Binary Tree

- Following types of representation:



Array Representation

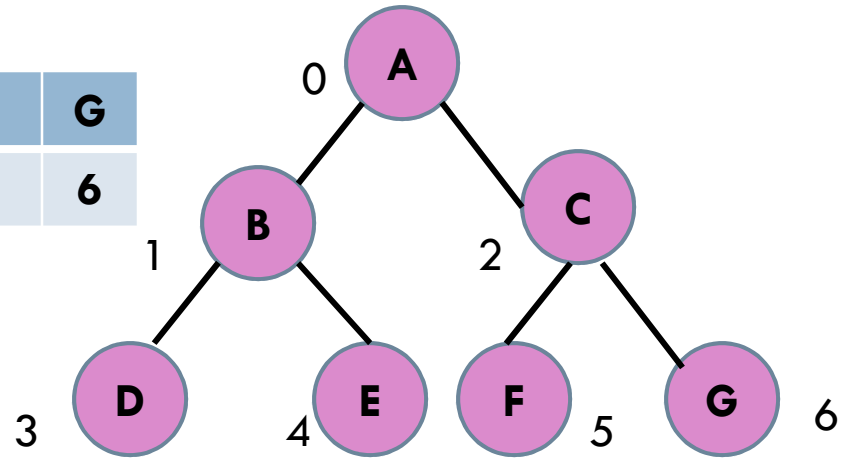
In this,

- Binary Tree ⟹ (Represented in) Sequence in memory with the help of single dimensional array.
- BT of level L ⟹ (may contain) at most (2^L)-1 nodes
- There for array of maximum (2^L)-1 is used.

Array Representation

- All the nodes of the tree are assigned a sequence number from 0 to (2^L) – 1 level by level.

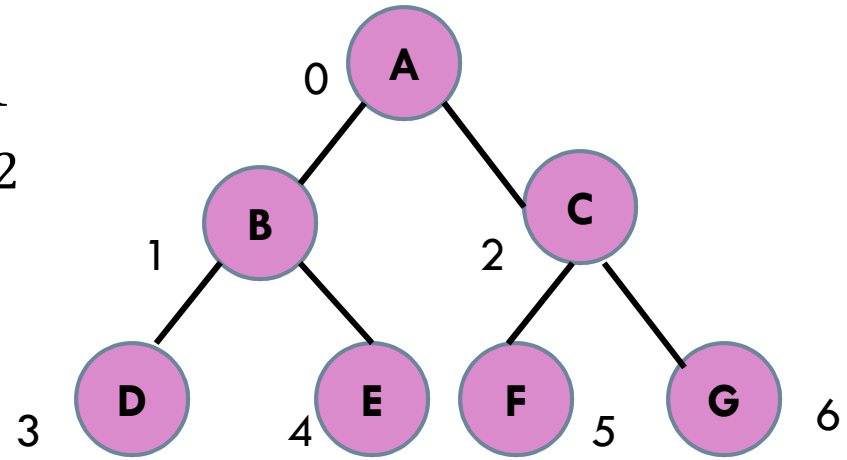
Element	A	B	C	D	E	F	G
Position	0	1	2	3	4	5	6



- Example: To represent a binary tree, with 3 levels.
- We required array size= (2^3)-1=7

Array Representation

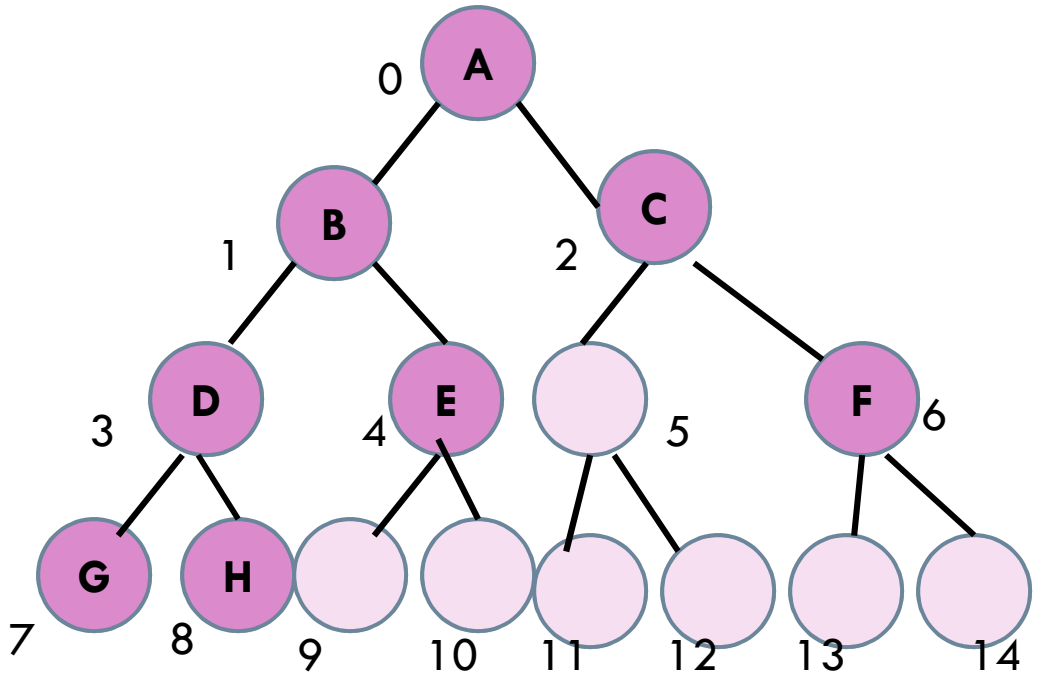
- Index value → at which position the node is stored.
- Any node → Position P
- Left child node → Position = 2*P+1
- Right child node → Position = 2*P+2



- For node B →Position 1
- Left child node D → Position = 2*P+1=(2*1+1)=3
- Right child node E → Position = 2*P+2=(2*1+2)=4

Array Representation

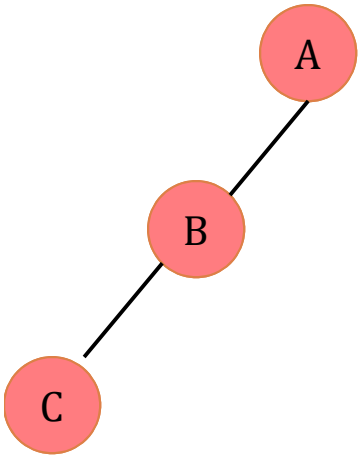
- If any of the nodes in the tree has empty sub trees, the nodes forming the part of these empty sub trees is also numbered and their values in the corresponding position in the array is NULL.



Element	A	B	C	D	E		F	G	H						
Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Array Representation

An array with maximum size is usually declared for data storage which may lead to wastage of lot of memory space when there is implementation of unbalanced trees such as skewed tree.



No. of nodes < max possible no. of nodes for a given height.
Level of tree = L =3
Size of array= (2^3)-1=8-1=7

Element	A	B	-	C	-	-	-
Position	0	1	2	3	4	5	6

Here, the wastage of lot of memory space de to more NULL values.
This advantage is removed in Linked representation.

Array Representation

Advantages

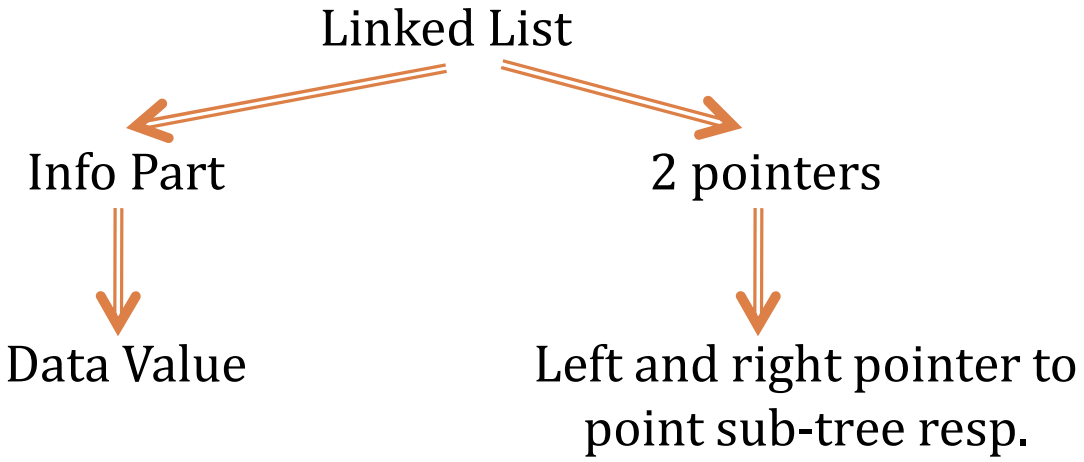
- Very easy to understand.
- Best representation for compete and full Binary Tree.
- Programming is very easy.
- Easy to move from child to it's parent or vice-versa.

Disadvantages

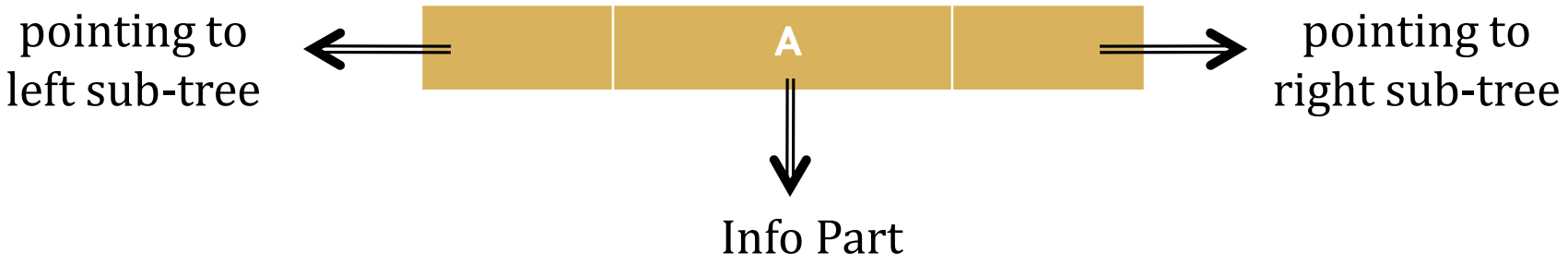
- Lot of memory area is wasted.
- Insertion and deletion needs lot of data movement.
- Execution time is high.
- No suited for trees other than full and complete Binary Tree.

Linked Representation

- Linked representation is considered as one of the most common and important method to represent a binary tree in memory.
- The linked representation of a binary tree is implemented with the help of linked list.

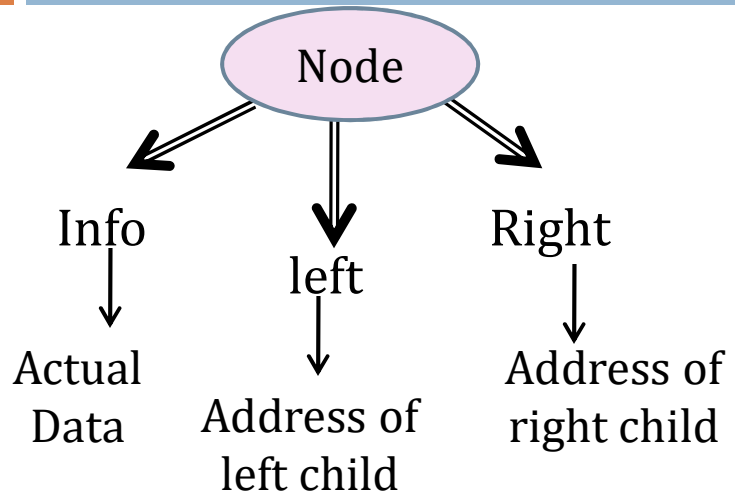


Linked Representation

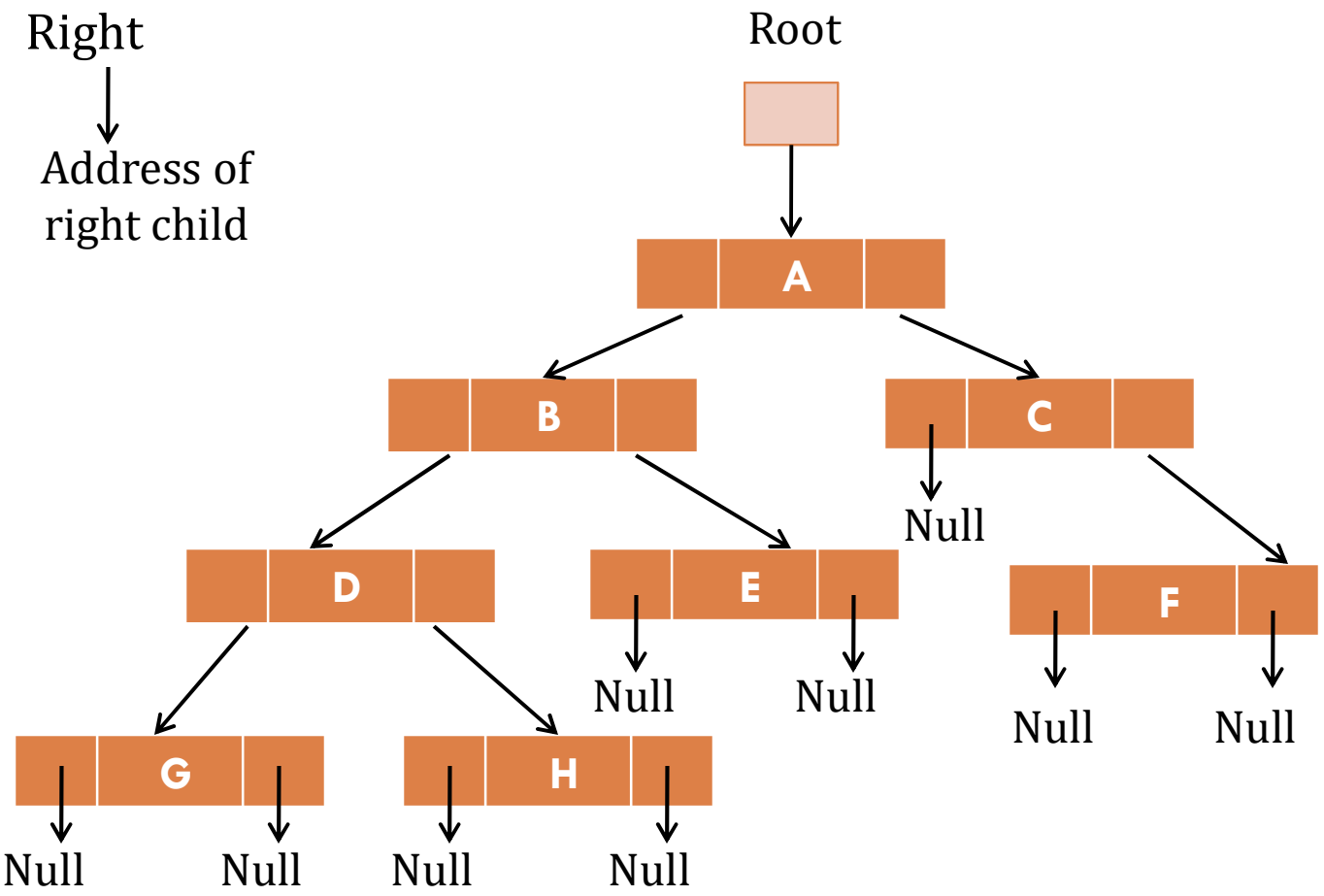



```
typedef struct node
{
        Int info;
        struct node *left;
        struct node *left;
} Node;
```

Linked Representation



Both left and Right are pointers and used to store address.



Linked Representation

Advantages

- It is easy to add new node at any location in tree.
- It is possible to insert or delete node directly without data movements.
- It is best for any types of trees.
- It is flexible because the system takes care of allocating and de-allocating of nodes.

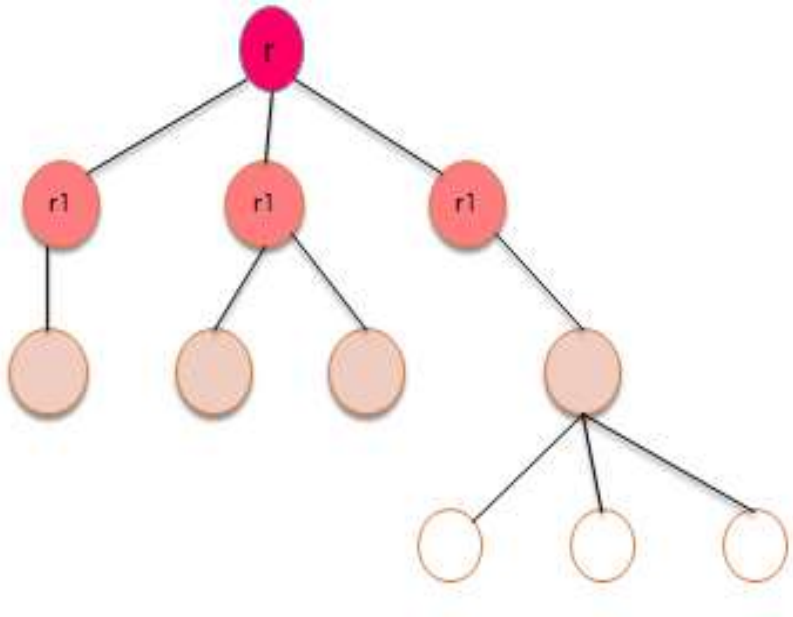
Disadvantages

- The mechanism is difficult to understand.
- There is a need of additional memory to store pointers.
- Accessing a particular node is not complicated.

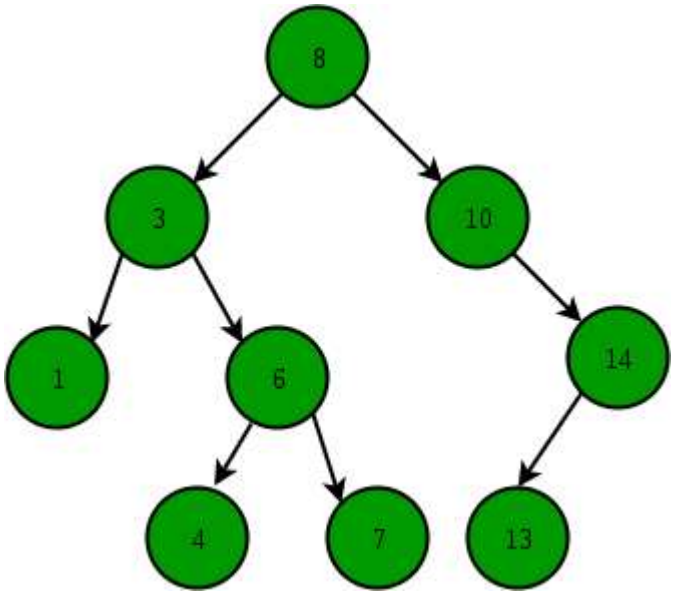
Comparison of General & Binary Tree

Parameter	General Tree	Binary Tree
Definition	A general tree is a data structure in which each node can have infinite number of children,	A Binary tree is a data structure in which each node has at most two nodes left and right.
Sub-tree	Sub-trees of general tree are not ordered.	Sub-trees of binary tree are ordered.
Empty	A General tree can't be empty.	A Binary tree can be empty.
Degree of root	In general tree, root have in-degree 0 and maximum out-degree n.	In binary tree, root have in-degree 0 and maximum out-degree 2.
Degree of other nodes	In general tree, each node has in-degree one and maximum out-degree n.	In binary tree, each node has in-degree one and maximum out-degree 2.

Comparison of General & Binary Tree



General Tree



Binary Tree

Converting Tree to Binary Tree

- A general tree can be changed into an equivalent binary tree. This conversion process or technique is called the natural correspondence between general and binary trees.

- The algorithm is written as below :

  (a) Insert the edges connecting siblings from left to right at the same level.

  (b) Erase all edges of a parent to its children except to its left most offspring.

  (c) Rotate the obtained tree 450 to mark clearly left and right subtrees.

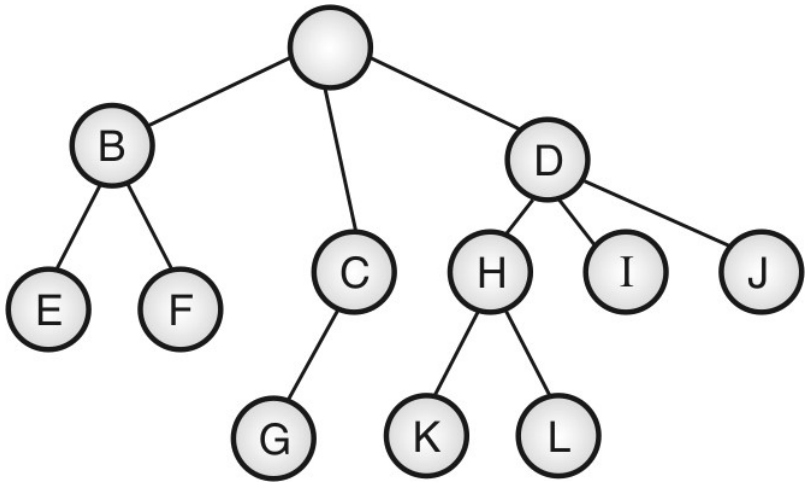
Converting Tree to Binary Tree

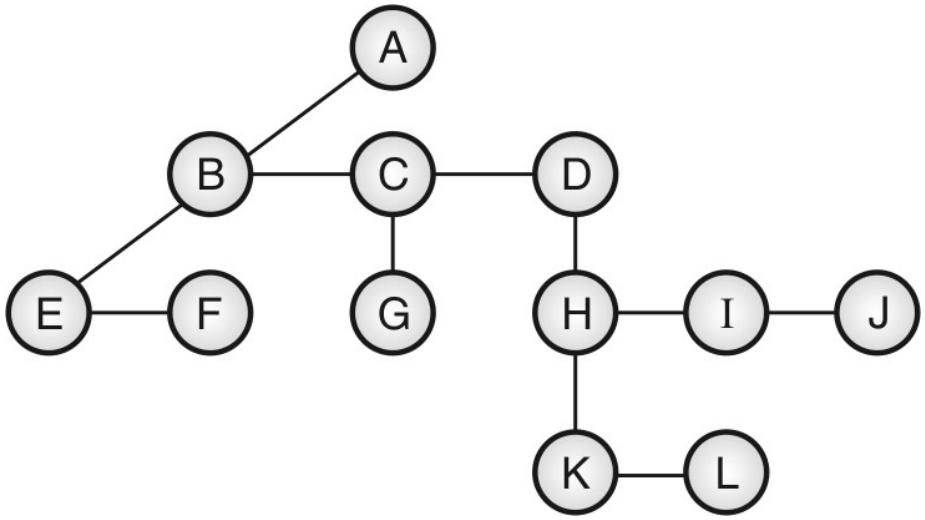
- The algorithm is written as below :

  (a) Insert the edges connecting siblings from left to right at the same level.

  (b) Erase all edges of a parent to its children except to its left most offspring.

  (c) Rotate the obtained tree 450 to mark clearly left and right subtrees.



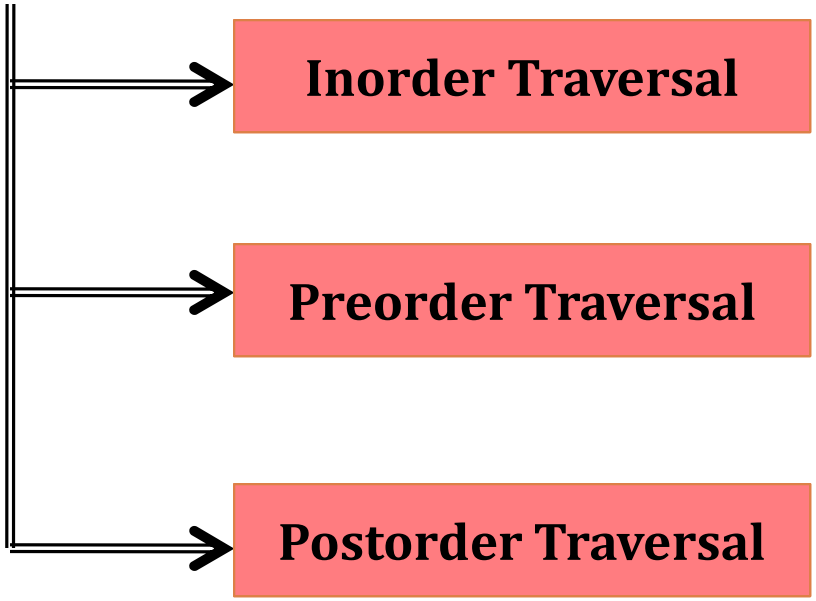


Binary Tree Traversal

- Traversing a tree is a process of visiting every nodes of tree at exactly once.
- For performing various operations on tree, we have to traverse the tree.
- **Linear Data Structure:** array, stacks, queues & linked list provide a single way to traverse the list.
- **Non-Linear Data Structure:** Tree which is hierarchical data structure can be traversed in different ways.

Binary Tree Traversal

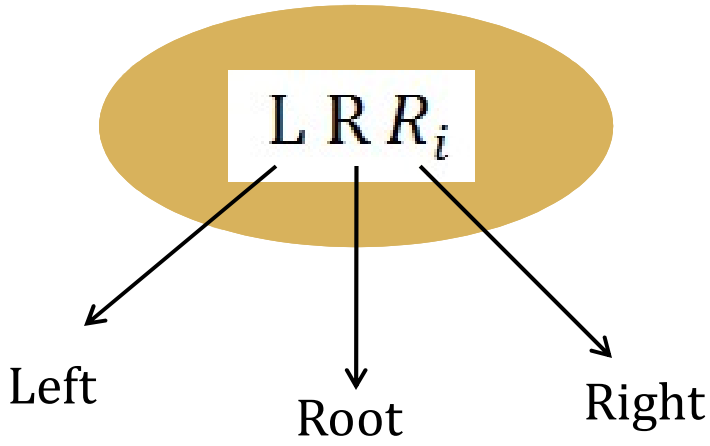
- Since, BT is defined in recursive manner, tree traversal too could be defined recursively.
- There are 3 different ways:



Binary Tree Traversal: Inorder

Recursive In-order Traversal

1. Traverse the left subtree, i.e. call Inorder (left-subtree)
2. Visit Root
3. Traverse the right subtree, i.e. call Inorder (right-subtree)

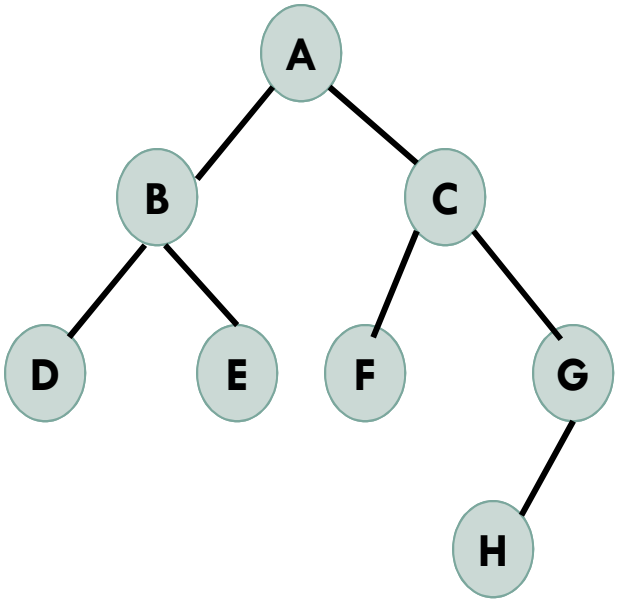


Binary Tree Traversal: Inorder

Recursive In-order Traversal

Example:

1. Start from Root
2. Goto left subtree as deep as possible.
3. Reach to leaf node print it.
4. Backtrack, print node which are visited 2^{nd} time.
5. Goto right subtree as deep as possible.
6. Reach to leaf node, print it.
7. Backtrack.



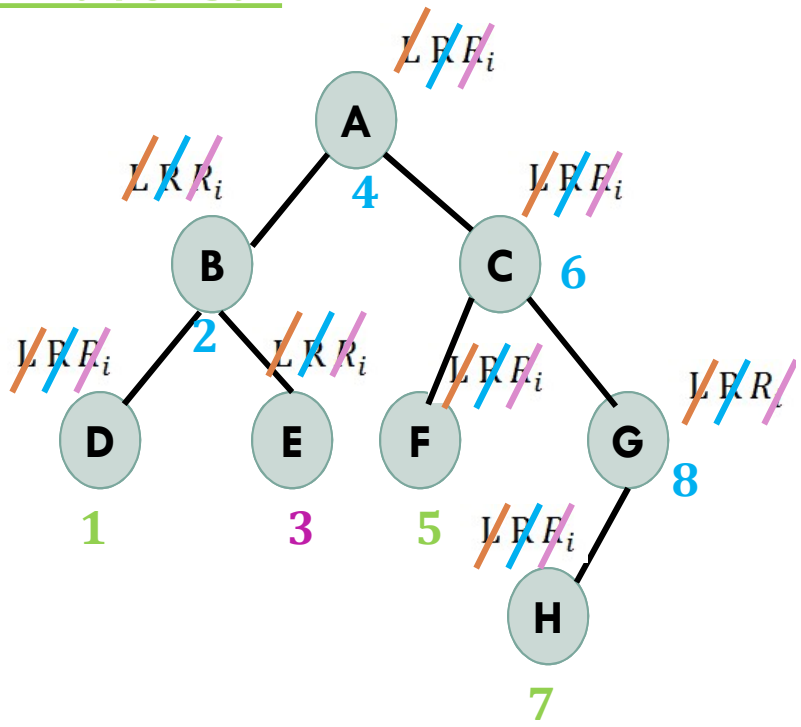
$$\mathrm{L}\ \mathrm{R}\ R_i$$

Binary Tree Traversal: Inorder

Recursive In-order Traversal

Example:

1. Start from Root.
2. Goto left subtree as deep as possible.
3. Reach to leaf node print it.
4. Backtrack, print node which are visited 2^{nd} time.
5. Goto right subtree as deep as possible.
6. Reach to leaf node, print it.
7. Backtrack.



D B E A F C H G

Binary Tree Traversal: Inorder

Recursive In-order Traversal

```
void inorder ( struct node *p )
{
        if ( p != NULL )
        {
                inorder ( p -> leftchild ) ;
                cout<<p-> data ;
                inorder ( p -> rightchild ) ;
        }
        else
                return ;
}
```

Binary Tree Traversal: Inorder

Non-Recursive In-order Traversal

Step1: Start traversing from root (say T) traverse left and continue traversing left. All nodes traversed are push into stack S.

```
while(T!=NULL)
{
        s.push(T);
        T=T→left;
}
```

Binary Tree Traversal: Inorder

Non-Recursive In-order Traversal

Step2: If the stack S is empty then traversal is finished.

```
        Else
          {     // visit right subtree of the node popped from stack
                T=s.pop;
                visit(T);
                T=T→right;
               // start traversing from T, traverse left & continue traversing
                  left. All nodes traversed are pushed on S.
                while(T!=NULL)
                {
                        s.push(T);
                        T=T→left;
                }
          }
```

Binary Tree Traversal: Inorder

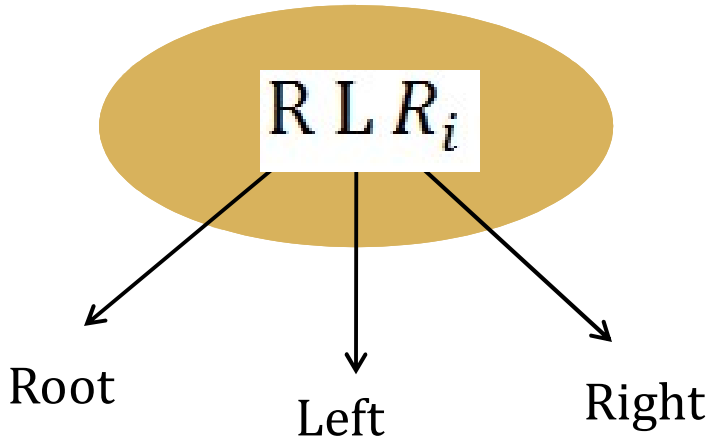
Non-Recursive In-order Traversal

```
void inorder Non-rec(node * T)
{
        stack s;
        while(T!=NULL)
        {
                s.push(T);
                T=T→left;
        }
        while (!s.Iempty())
        {       T=s.pop;
                print T->data;
                T=T→right;
                while(T!=NULL)
                {       s.push(T);
                        T=T→left;
                }
        }
}
```

Binary Tree Traversal: Preorder

Recursive Pre-order Traversal

1. Visit Root
2. Traverse the left subtree, i.e. call preorder (left-subtree)
3. Traverse the right subtree, i.e. call preorder (right-subtree)

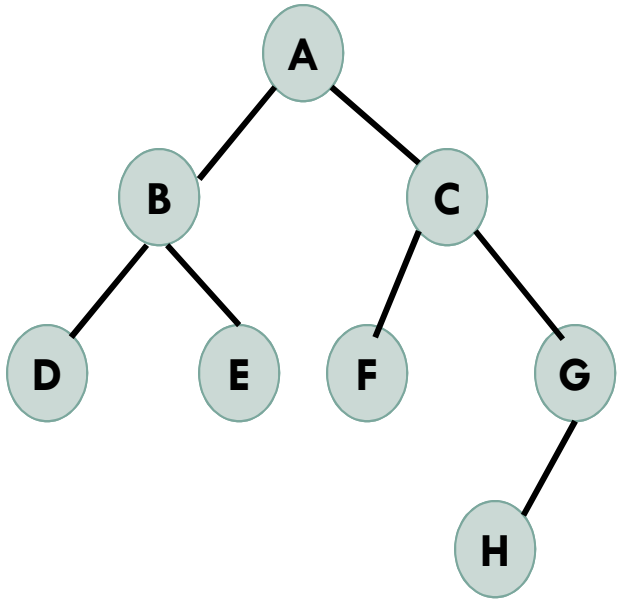


Binary Tree Traversal: Preorder

Recursive Pre-order Traversal

Example:

1. Start from Root, print it.
2. Goto left subtree as deep as possible.
3. While visiting each node print it.
4. Go up to leaf node.
5. Backtrack
6. Go to right subtree as deep as possible.
7. While visiting each node, print it.
8. Backtrack.

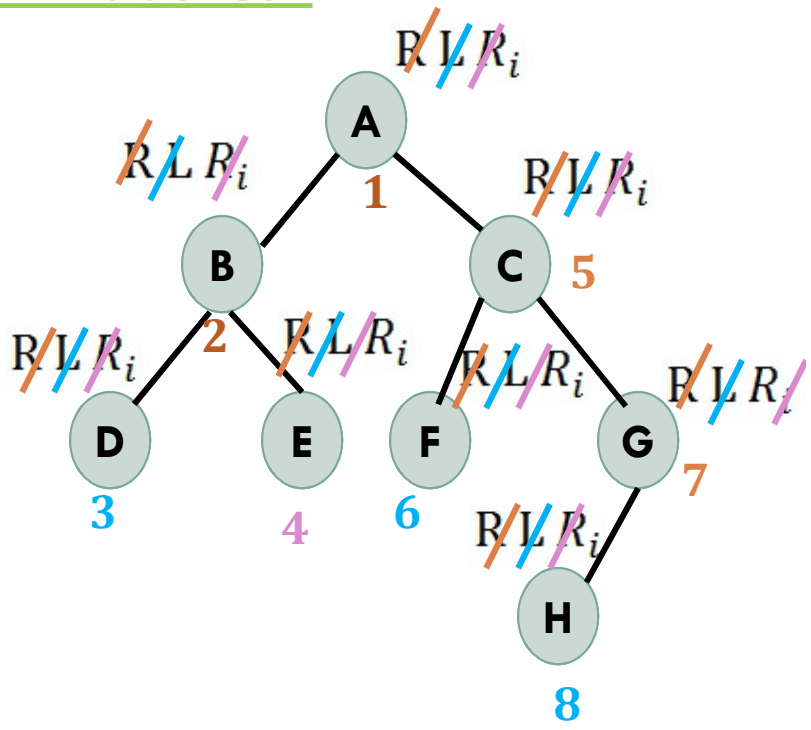


Binary Tree Traversal: Preorder

Recursive Pre-order Traversal

Example:

1. Start from Root, print it.
2. Goto left subtree as deep as possible.
3. While visiting each node print it.
4. Go up to leaf node.
5. Batrckack
6. Go to right subtree as deep as possible.
7. While visiting each node, print it.
8. Backtrack.



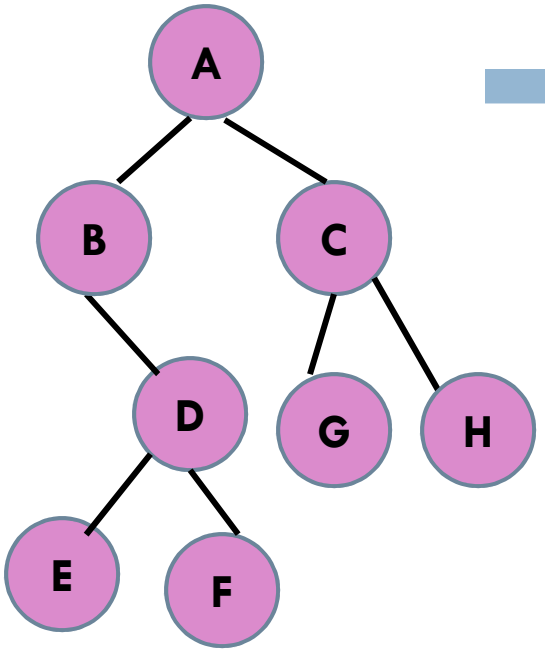
A B D E C F G H

Binary Tree Traversal: Preorder

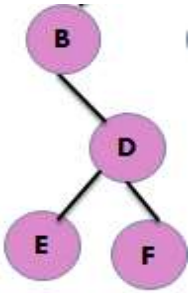
Recursive Preorder Traversal

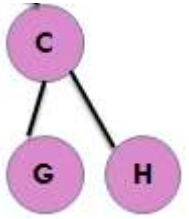
```
void preorder ( struct node *p )
{
        if ( p != NULL )
        {
                cout<<p-> data ;
                preorder ( p -> leftchild ) ;
                preorder ( p -> rightchild ) ;
        }
        else
                return ;
}
```

Example #1

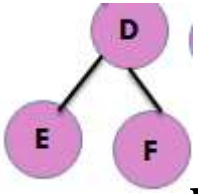


A + preorder on + preorder on





= A +(B+ preorder on)+(c + preorder on +preorder on)



= A +(B+D+ preorder on + preorder on) + (C + G + H)

= A +(B + D + E + F) + (C + G + H)

A B D E F C G H

Array Representation

Construct BT for the preorder and inorder traversal sequences.

Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F

Array Representation

Construct BT for the preorder and inorder traversal sequences.

Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F

Array Representation

Construct BT for the preorder and inorder traversal sequences.

Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F

Array Representation

Construct BT for the preorder and inorder traversal sequences.

Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F

Array Representation

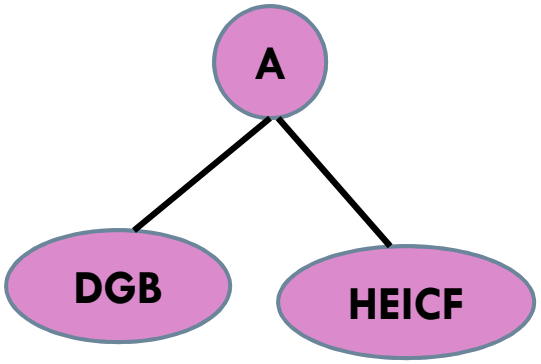
Construct BT for the preorder and inorder traversal sequences.

Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F

Array Representation

Construct BT for the preorder and inorder traversal sequences.

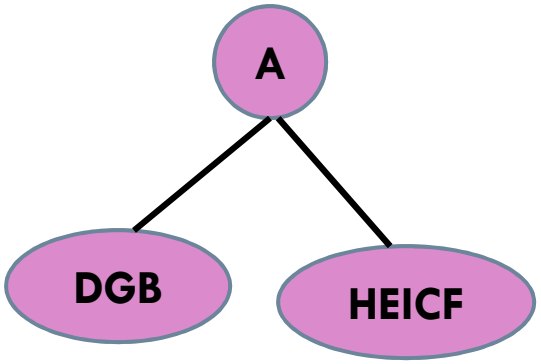
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

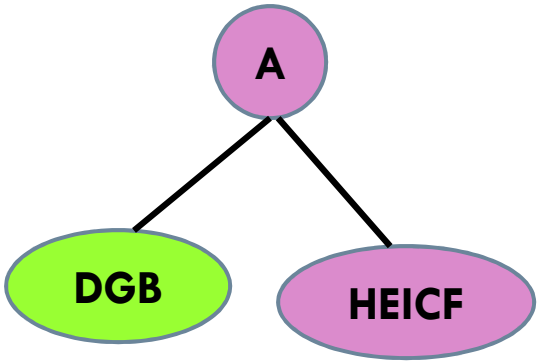
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

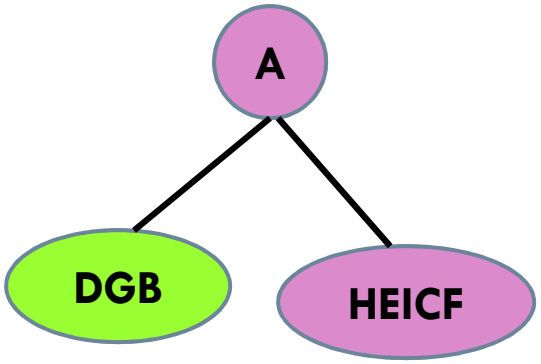
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

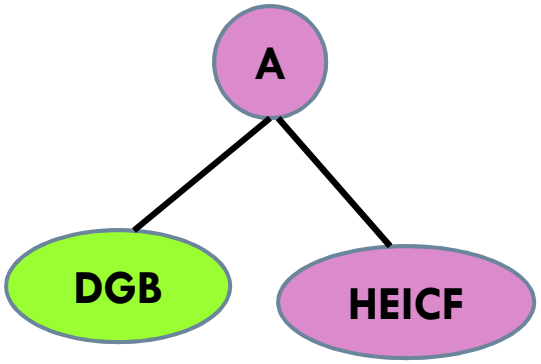
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

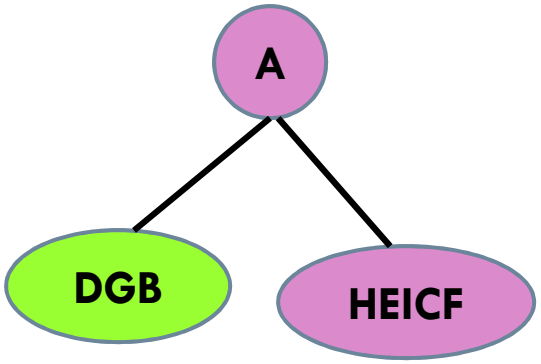
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

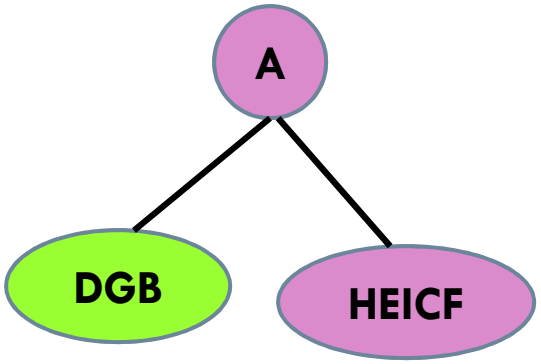
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

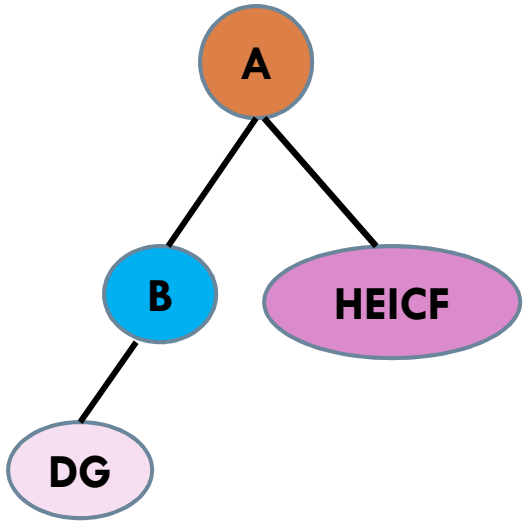
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

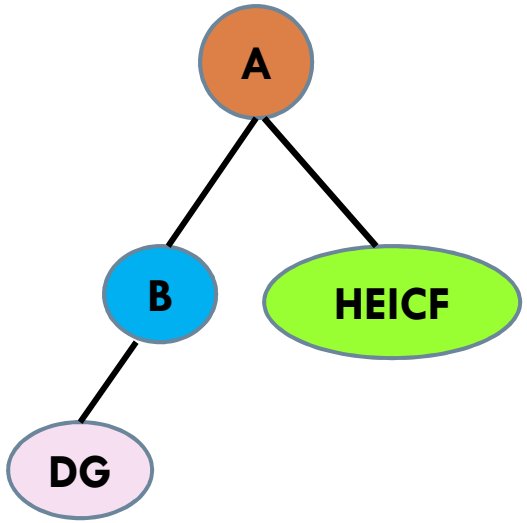
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

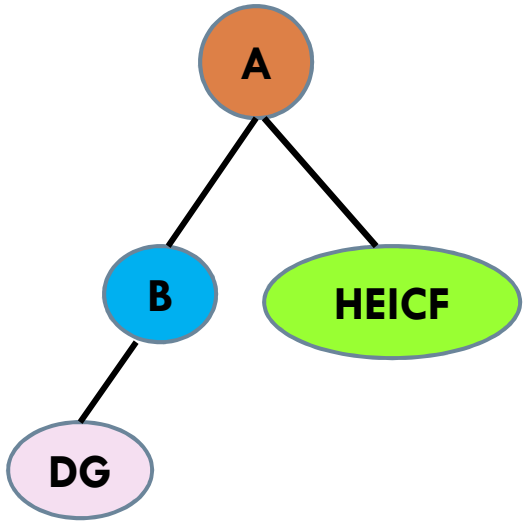
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

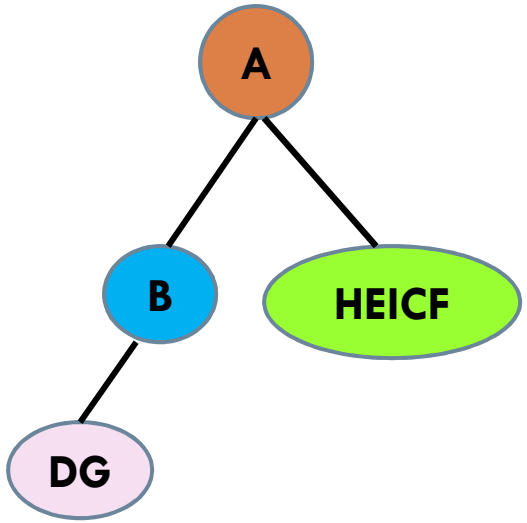
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

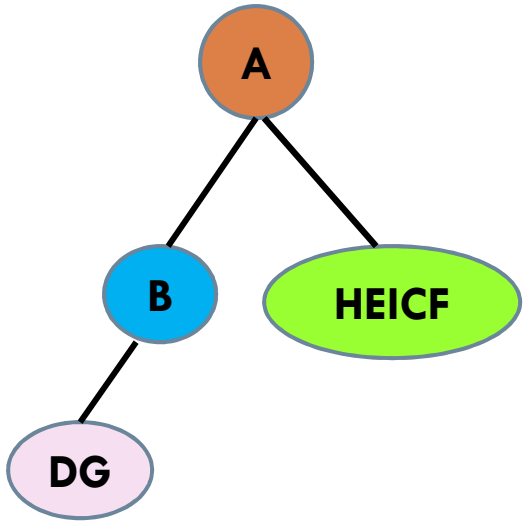
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

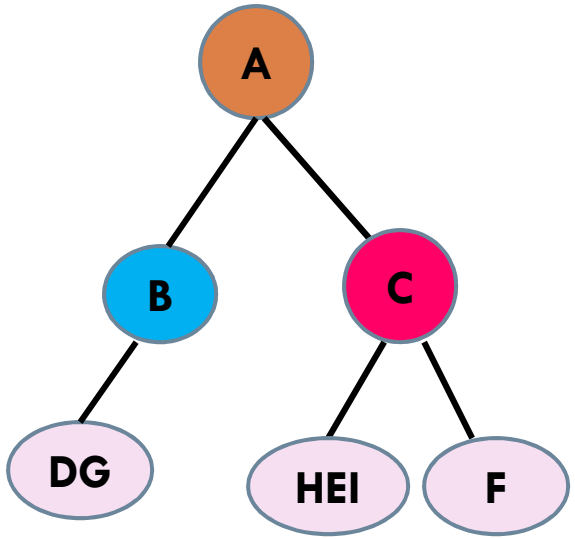
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

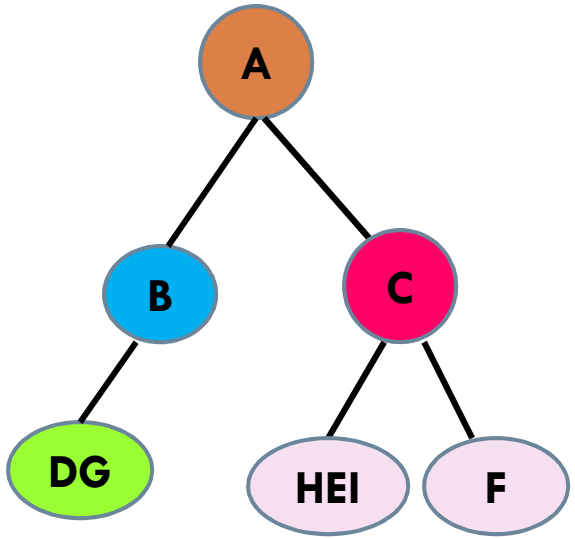
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

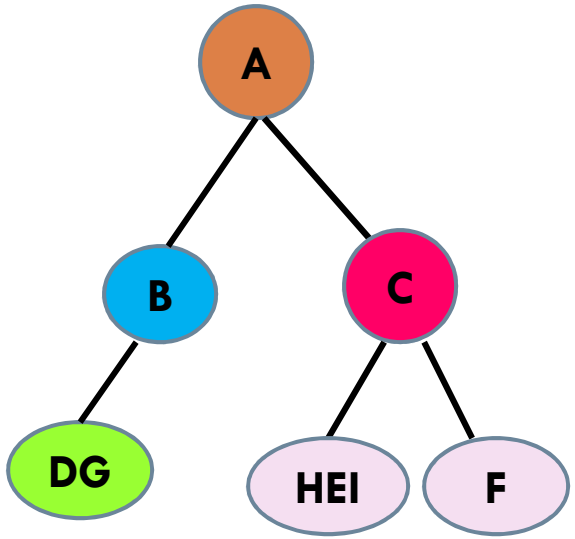
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

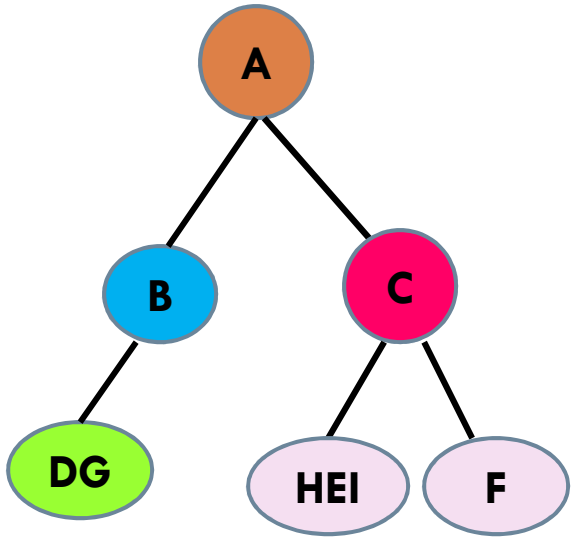
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

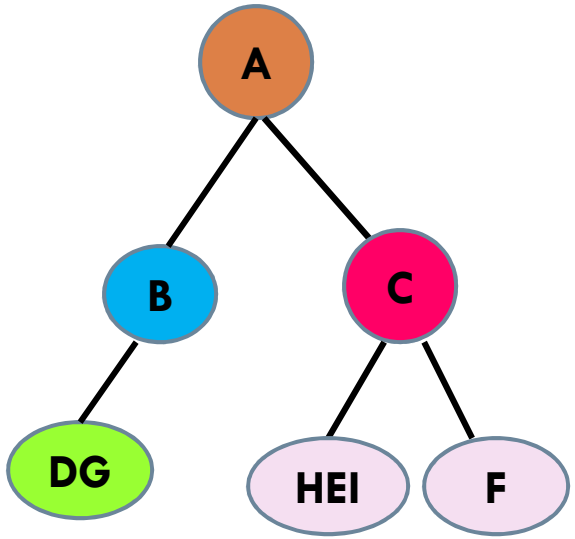
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

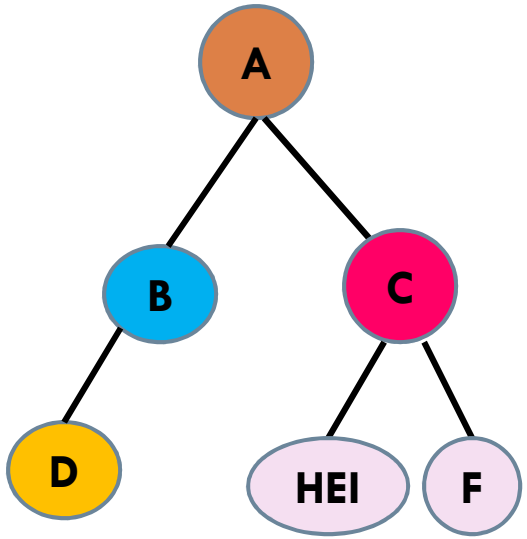
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

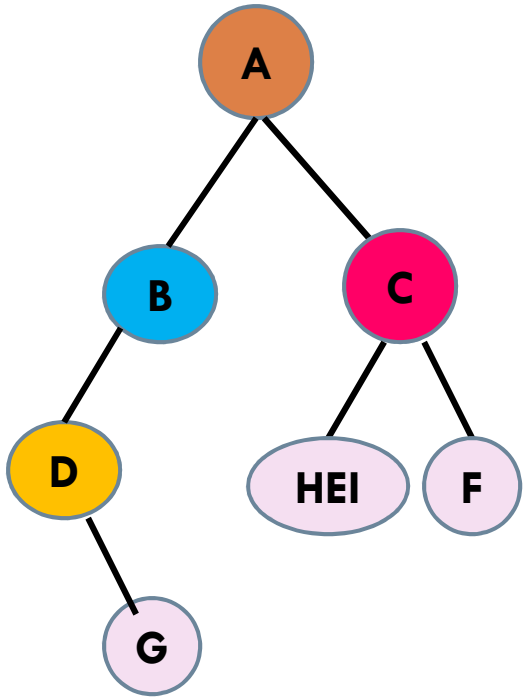
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

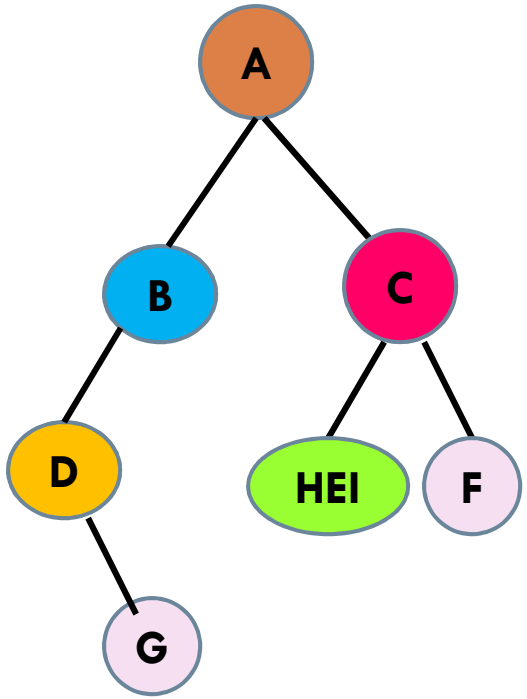
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

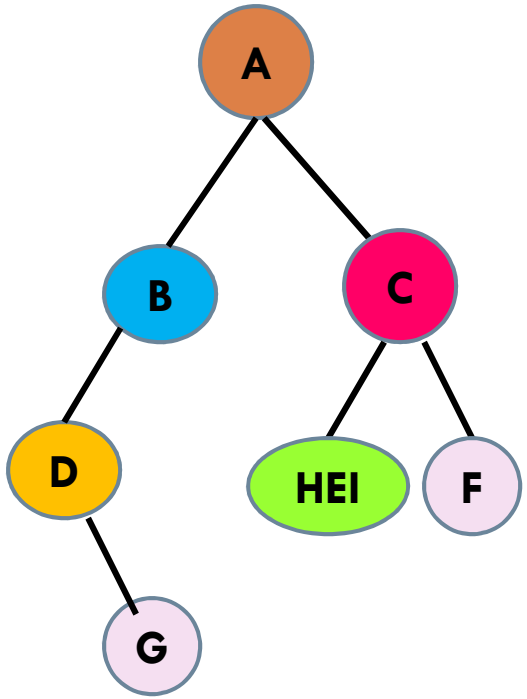
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

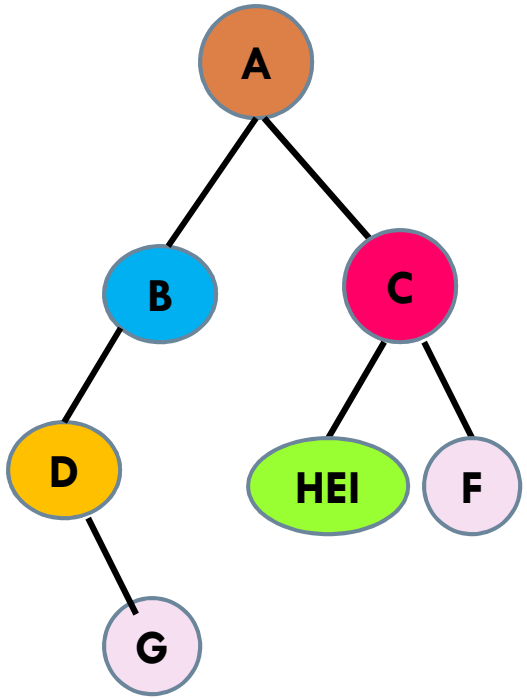
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

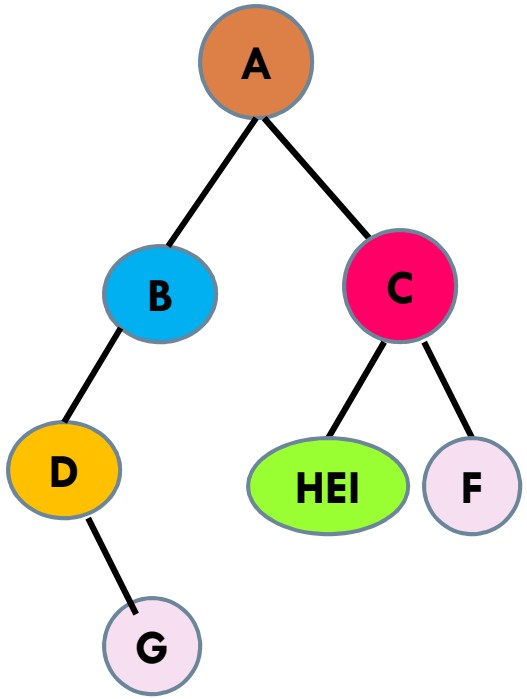
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

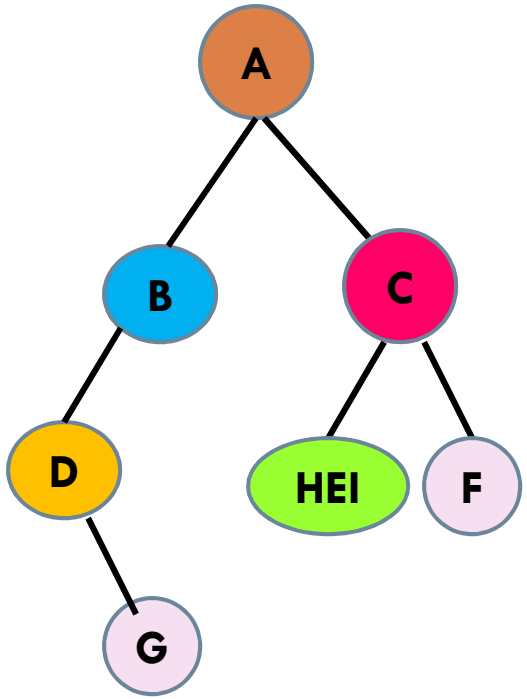
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

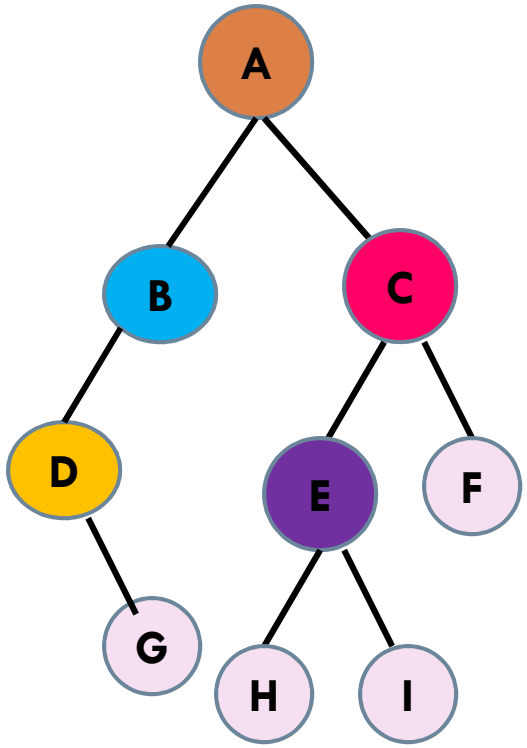
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

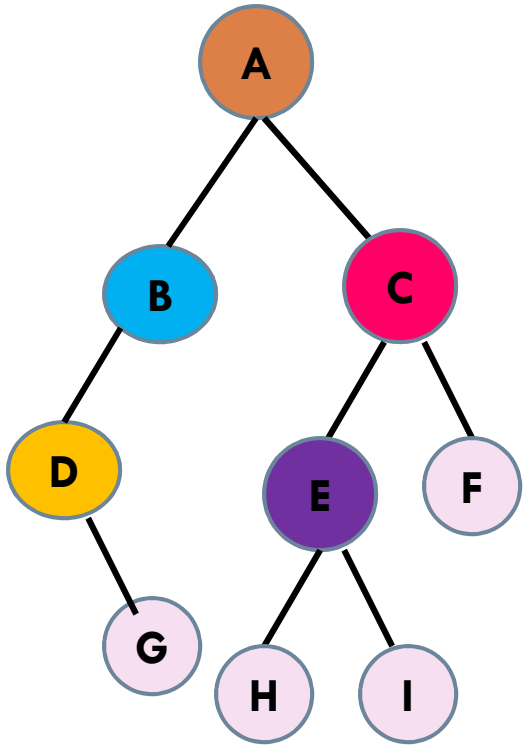
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

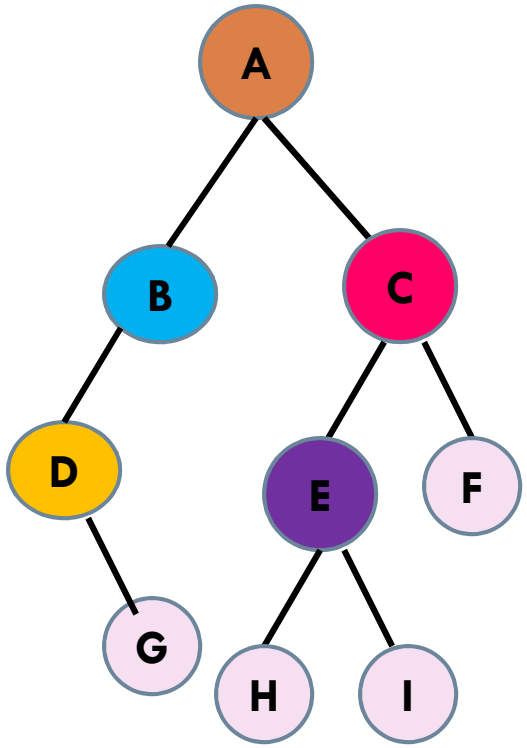
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

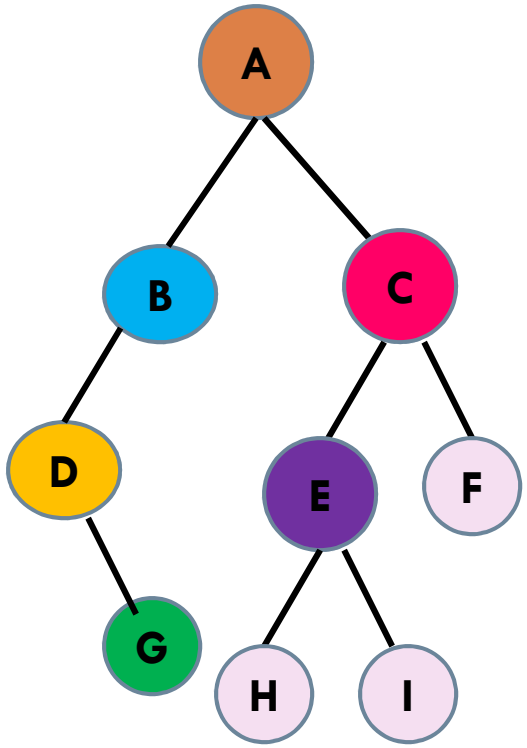
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

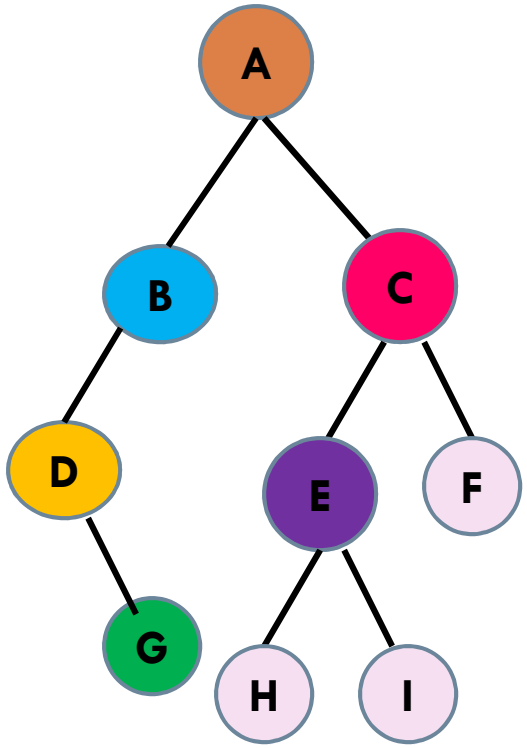
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

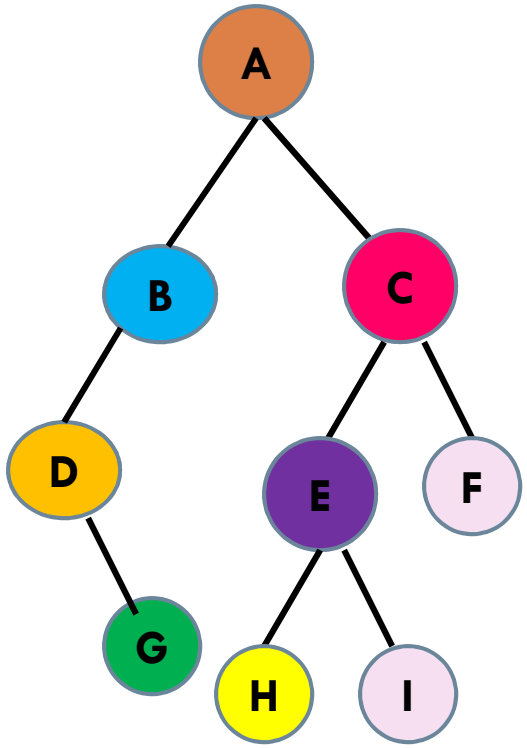
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

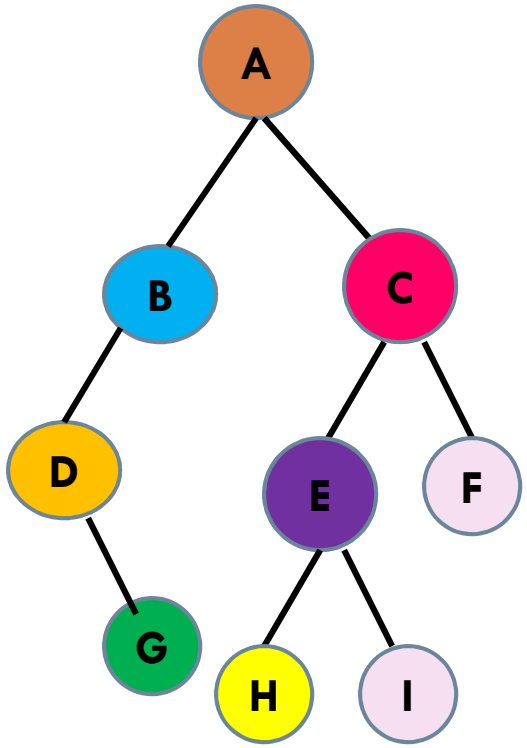
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

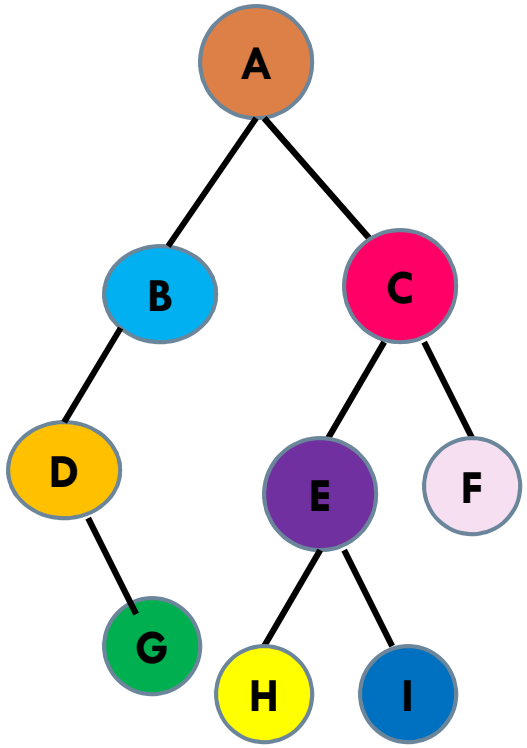
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

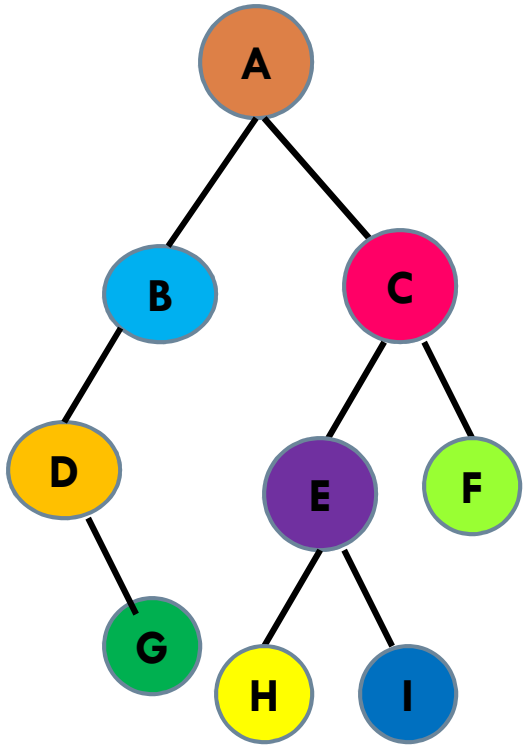
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

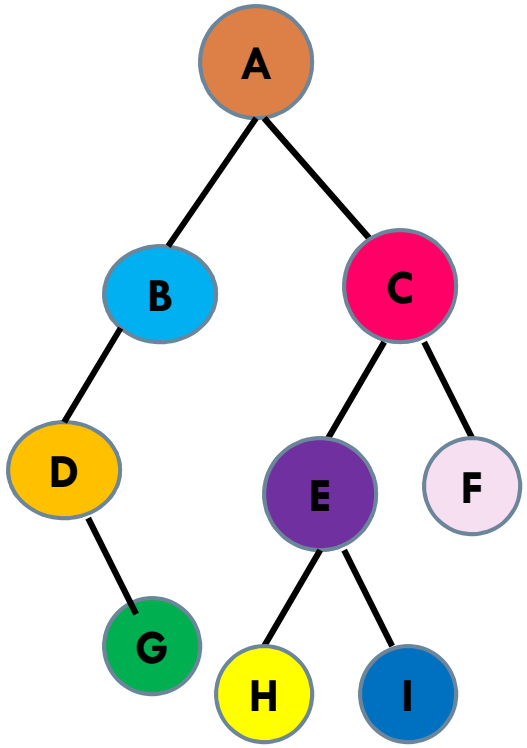
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Array Representation

Construct BT for the preorder and inorder traversal sequences.

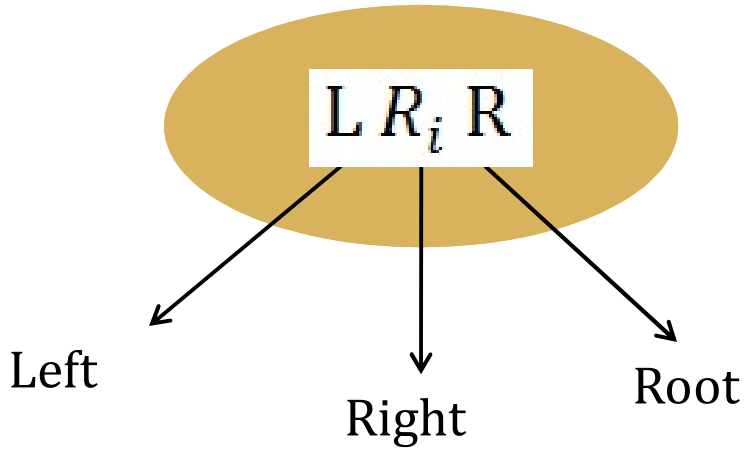
Preorder	A	B	D	G	C	E	H	I	F
Inorder	D	G	B	A	H	E	I	C	F



Binary Tree Traversal: Postorder

Recursive Post-order Traversal

1. Traverse the left subtree, i.e. call preorder (left-subtree)
2. Traverse the right subtree, i.e. call preorder (right-subtree)
3. Visit the Root(subtree)

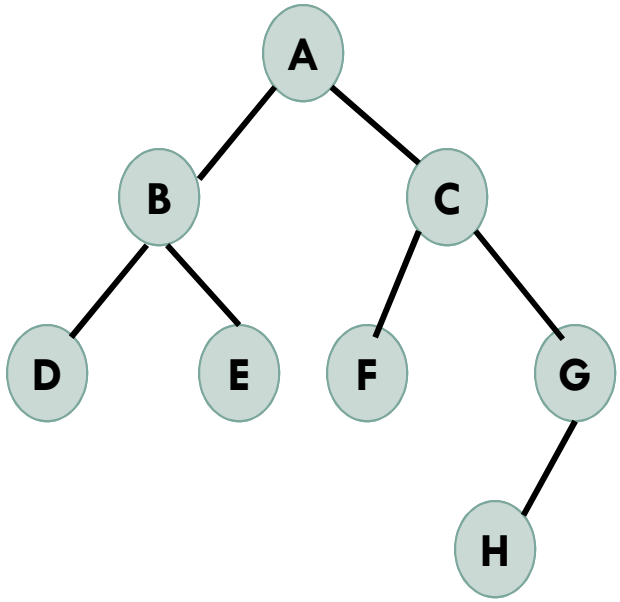


Binary Tree Traversal: Postorder

Recursive Post-order Traversal

Example:

1. Start from Root
2. Goto left subtree as deep as possible.
3. Go up to leaf node, print it
4. Go to right subtree as deep as possible.
5. Reach to leaf node, print it
6. Backtrack.
7. Print node while revisiting

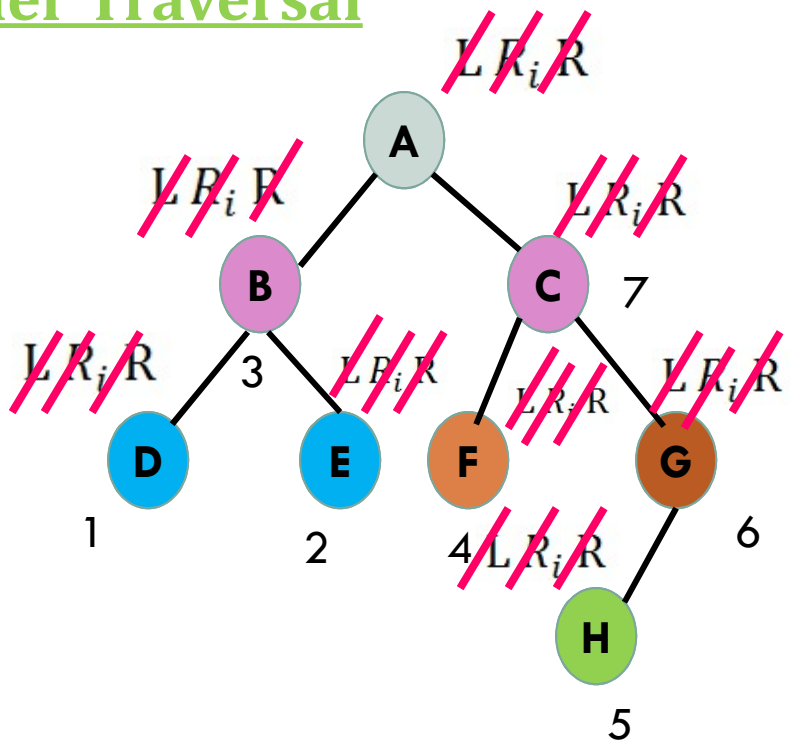


Binary Tree Traversal: Postorder

Recursive Post-order Traversal

Example:

1. Start from Root
2. Goto left subtree as deep as possible.
3. Go up to leaf node, print it
4. Go to right subtree as deep as possible.
5. Reach to leaf node, print it
6. Backtrack.
7. Print node while revisiting

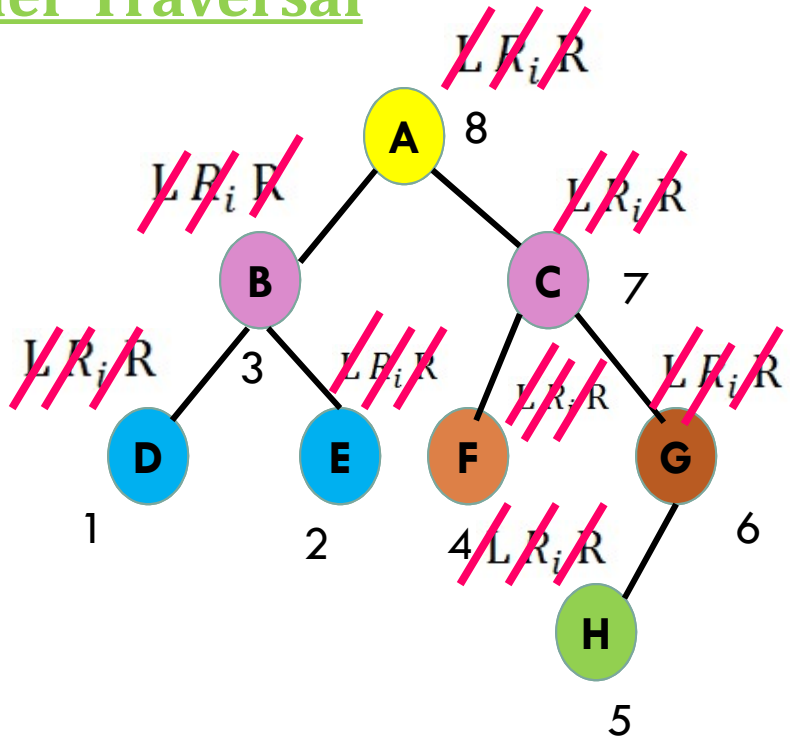


Binary Tree Traversal: Postorder

Recursive Post-order Traversal

Example:

1. Start from Root
2. Goto left subtree as deep as possible.
3. Go up to leaf node, print it
4. Go to right subtree as deep as possible.
5. Reach to leaf node, print it
6. Backtrack.
7. Print node while revisiting



D E B F H G C A

Binary Tree Traversal: Postorder

Recursive Postorder Traversal

```
void postorder ( struct node *p )
{
        if ( p != NULL )
        {
                postorder ( p -> leftchild ) ;
                postorder ( p -> rightchild ) ;
                cout<< p-> data ;
        }
        else
        return ;
}
```

Binary Search Tree (BST)

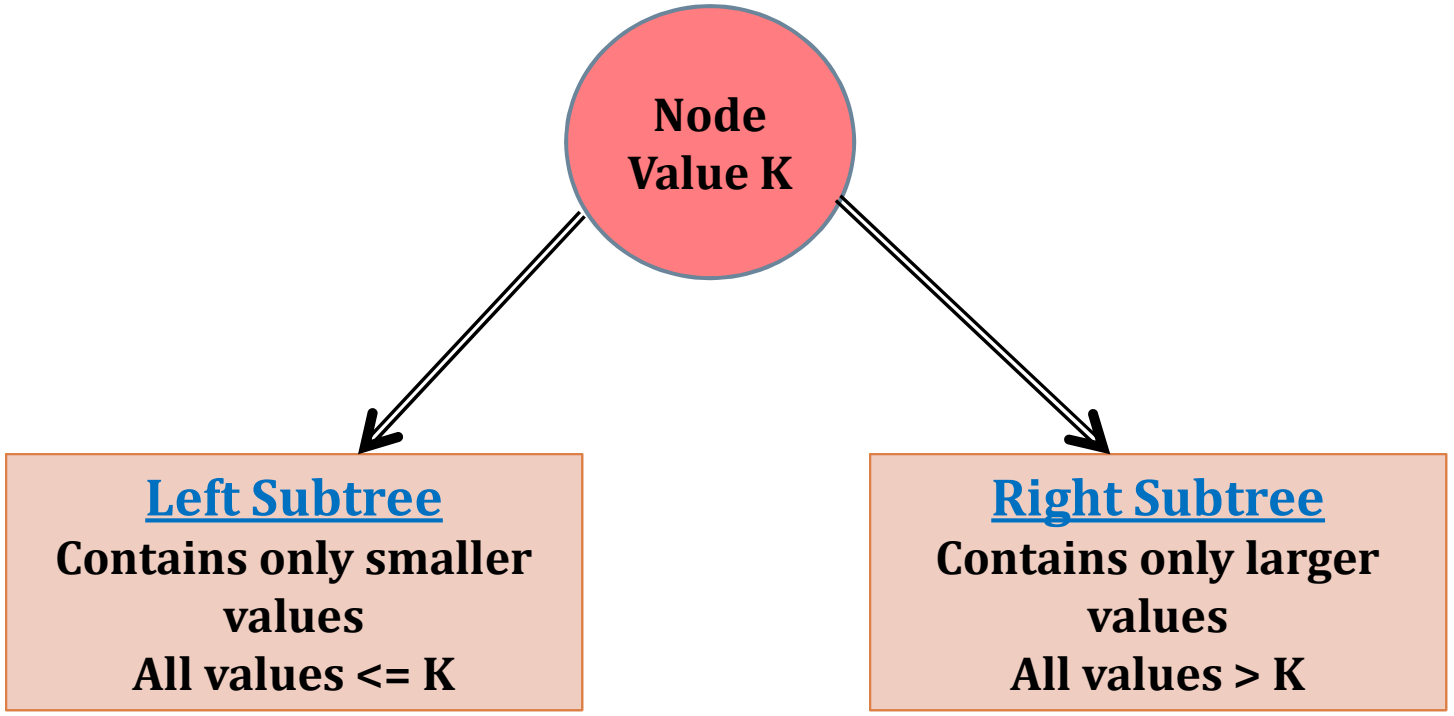
Binary Search Tree has the following mentioned properties :

- The left sub-tree of a node contains only nodes having value less than the node.
- The right sub-tree of a node contains only nodes having value greater than the node.
- In the left and right sub-tree all the nodes must also be a binary search tree.

A binary tree in which the data of all the nodes in the left sub-tree of the root node is less than the data of the root and the data of all the nodes in the right sub-tree of the root node is more than the data of the root, is called as Binary Search Tree.

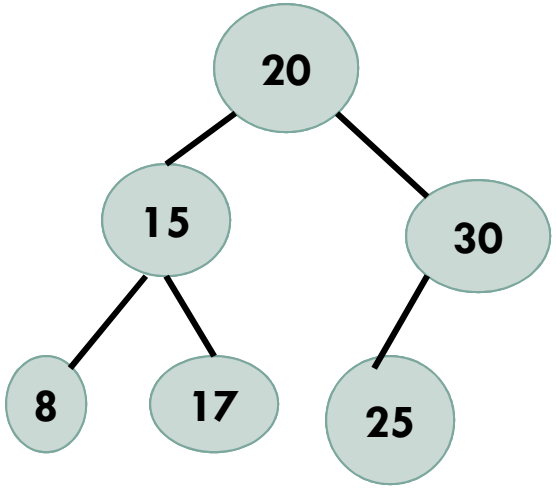
Binary Search Tree (BST)

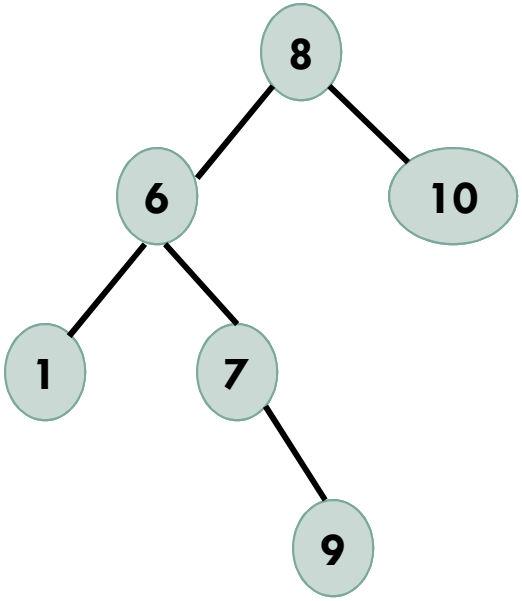
Node Structure of BST



Binary Search Tree (BST)

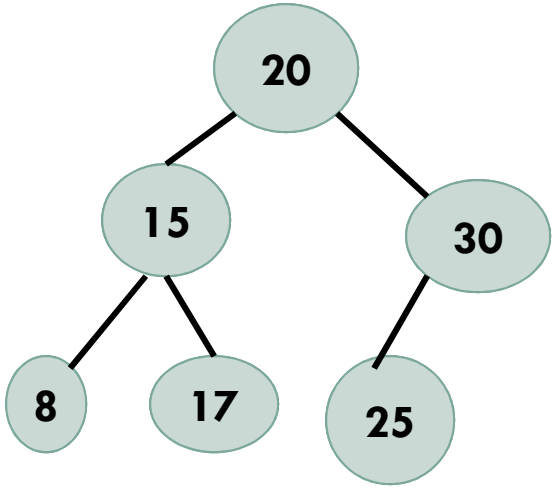
Example:

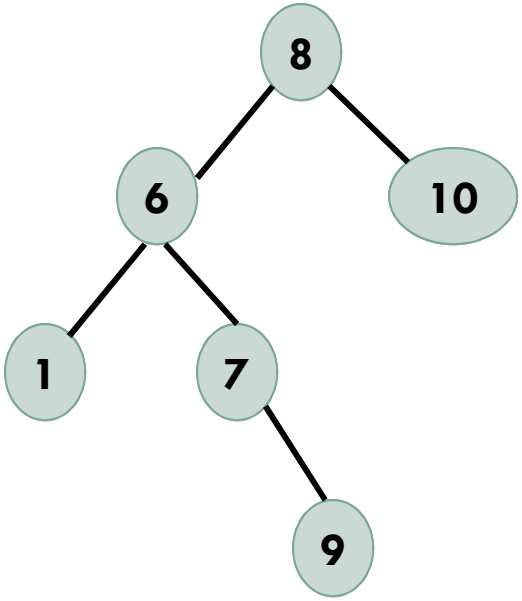




Binary Search Tree (BST)

Example:

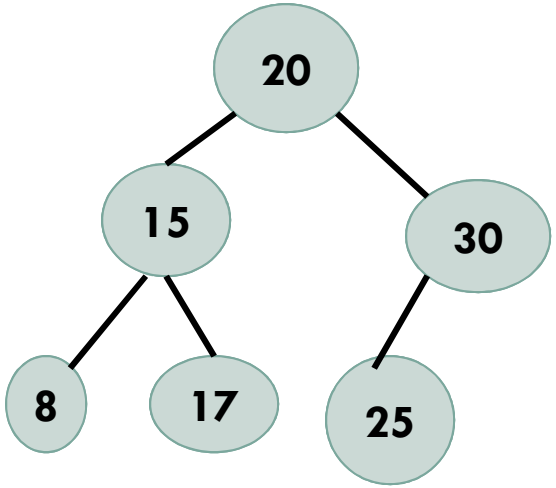


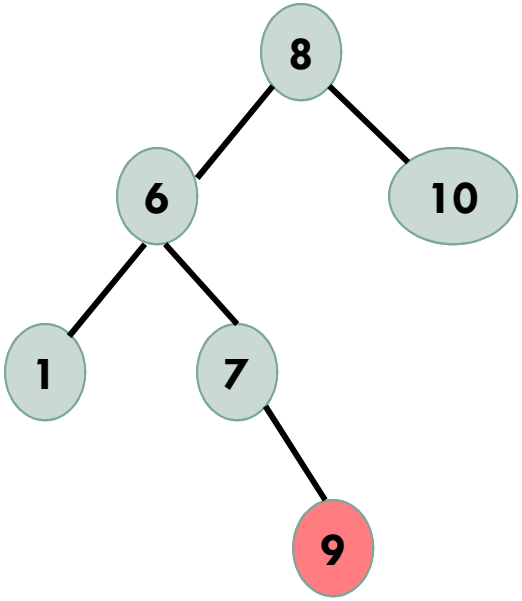


BST

Binary Search Tree (BST)

Example:

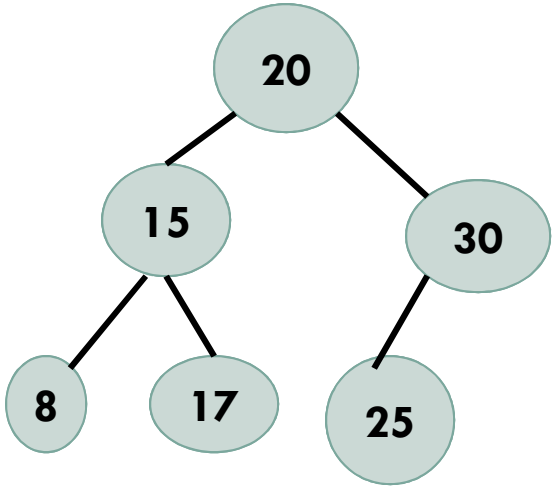




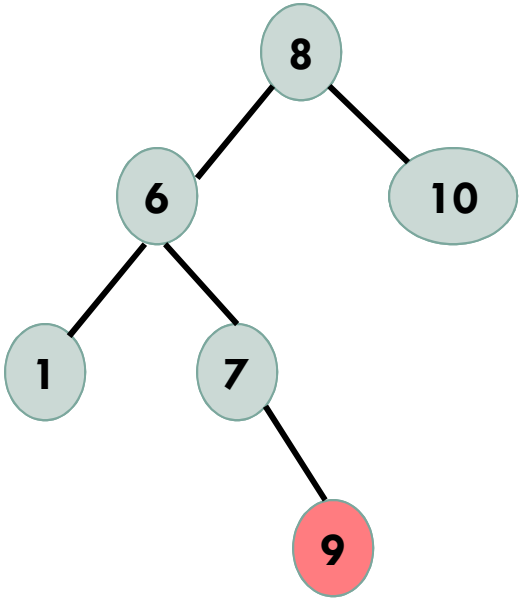
BST

Binary Search Tree (BST)

Example:



BST

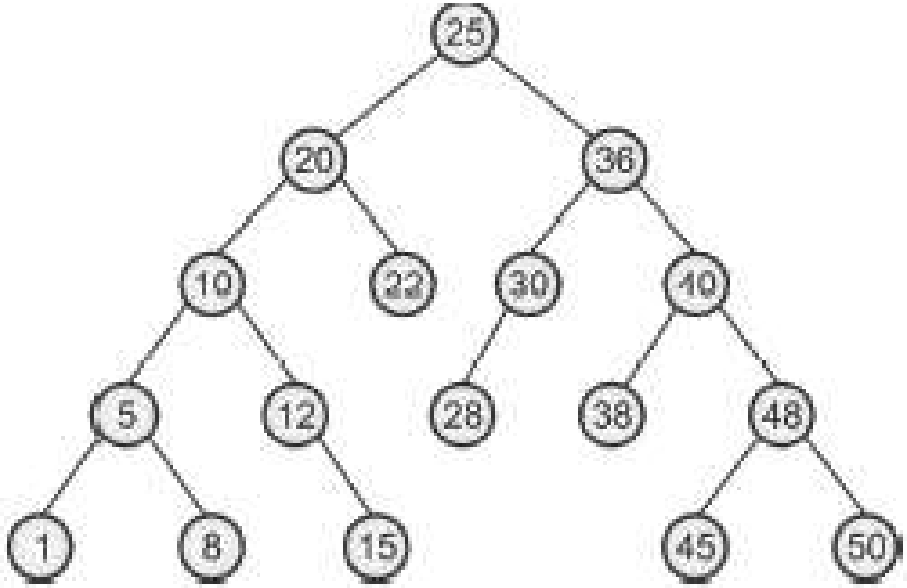


Not a BST

Binary Search Tree (BST)

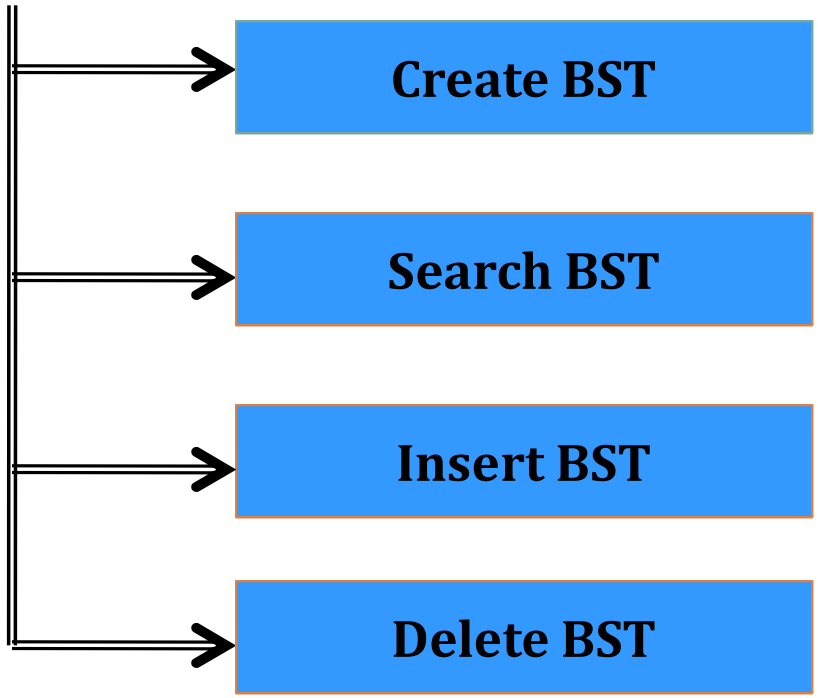
Example:

- The tree shown below is an example of Binary Search Tree.
- In this tree, we can observe that left sub-tree of every node has nodes with smaller values while right subtree of every node has larger values than the root.



Operations on BST

- There are various types of operations which can be performed on Binary Search Tree.



Other Operations:
- Initialize
- Make empty
- Find min
- Find max

Operations on BST

Create BST

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

Operations on BST

Initialize

Initially as there is node i.e. tree is empty, the reference pointer will be kept as NULL.

```
node * Initialize ()
{
        return NULL;
}
```

Operations on BST

Search node in BST

Step 1 : Read the data (search element) from user to search.
Step 2 : Compare, the search element with the value of root node in the tree.
Step 3 : If both are equal, then display "element found" and exit from the function
Step 4 : If both are not equal, then check whether search element is smaller or larger than that node value.
Step 5 : If search element is smaller, then continue the search process in left sub-tree.
Step 6 : If search element is larger, then continue the search process in right sub-tree.
Step 7 : Repeat the same until we found exact element or we completed with a leaf node.
Step 8 : If we reach to the node with search value, then display "**Element found**" and exit from the function.
Step 9 : If we reach to a leaf node and it is also not matching, then display "**Element not found**" and exit from the function.

Operations on BST

Search node in BST: Recursive

```
node * find (node * root, int x)
{
        if(root==NULL)
                return NULL;
        if(root→ data == x)
                return root;
        if(x > root→ data)
                return(find(root→right,x));
        return(find(root→left, x));
}
```

Operations on BST

Search node in BST:
Non-Recursive

```
node * NRfind (node * root, int x)
{
        while(root!=NULL)
        {
                if(x== root→ data)
                        return root;
                 if(x>root→ data)
                        root =root →right;
                else
                        root=root → left;
        }
        return(NULL);
}
```

Operations on BST

Insert node in BST

In a binary search tree, the insertion operation is performed with O(log n) time complexity.

In binary search tree, new node is always inserted as a leaf node.

The insertion operation is performed as follows :

Binary Search Tree (BST)

Insert node in BST

Step 1 : Create a newNode with given value and set its left and right to NULL.

Step 2 : Check whether tree is Empty.

Step 3 : If the tree is Empty, then set root to newNode.

Step 4 : If the tree is Not Empty, then check whether value of newNode is smaller or larger than the node (here it is root node).

Step 5 : If newNode is smaller than or equal to the node, then move to its left child. If newNode is larger than the node, then move to its right child.

Step 6 : Repeat the above step until we reach to a leaf node

Step 7 : After reaching a leaf node, insert the newNode as left child if newNode is smaller or equal to that leaf else insert it as right child.

Binary Search Tree (BST)

Insert node in BST

```
Node * Insert(node *T, int x)
{
        if (T==NULL)
        {
                T= new node;
                T→ data =x
                T→ left=NULL
                T→ right=NULL
                return T;
        }
```

Binary Search Tree (BST)

Insert node in BST

```
Node * Insert(node *T, int x)
{
        if (T==NULL)
        {
                T= new node;
                T→ data =x
                T→ left=NULL
                T→ right=NULL
                return T;
        }
```

```
If(x > T→data)
{
        T→right=Insert(T→right,x)
        Return (T);
}

T→left=Insert(T→left,x)

Retuen (T);
}
```

In Order Traversal of In-Order Threaded Binary Tree

- The main motivation behind using a threaded binary tree over a simpler and smaller standard binary tree is to increase the speed of an in-order traversal of the tree.

- An in-order traversal of a binary tree visits the nodes in the order in which they are stored, which matches the underlying ordering of a binary search tree.

- This means most threaded binary trees are also binary search trees. The idea is to visit all the left children of a node first, then visit the node itself, and then visit the right children last.

In Order Traversal of In-Order Threaded Binary Tree

- An in-order traversal of any binary tree generally goes as follows :

```
func traverse(n: Node)
{
        if (n == nil)
        {
                return
        }
        else
        {
                traverse(n.left)
                visit(n)
                traverse(n.right)
        }
}
```

In Order Traversal of In-Order Threaded Binary Tree

- Where n is a node in the tree, each node stores its children as left and right, and "visiting" a node can mean performing any desired action on it.
- this algorithm uses stack space proportional to the height of the tree due to its recursive nature.
- If the tree has n nodes, this usage can range anywhere from O(log n) for a fairly balanced tree, to O(n) to a very unbalanced tree.
- A threaded binary tree fixes this problem.

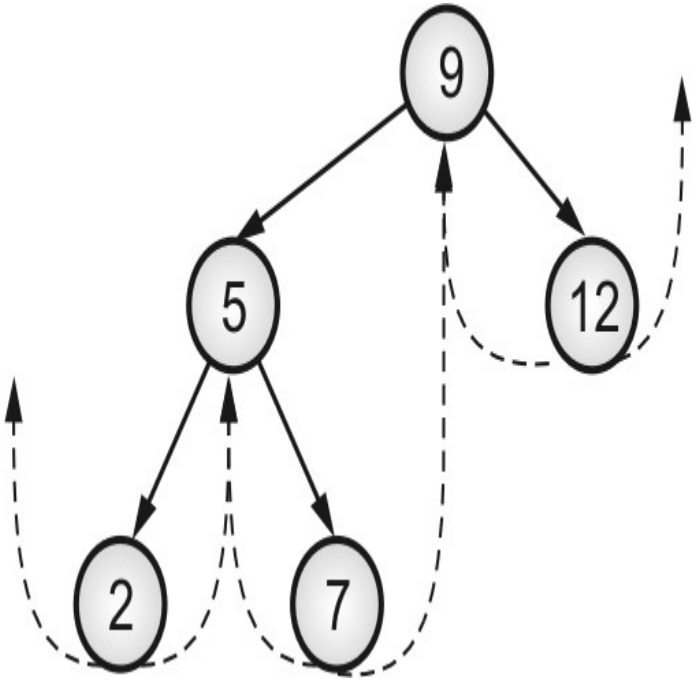
Threaded Binary Tree: Predecessors and successors

- An in-order traversal of a tree yields a linear ordering of the nodes.
- Thus each node has both a predecessor and a successor (except for the first and last nodes, which only have a successor or a predecessor respectively).
- Before we go into detail about the methods of a threaded binary tree, lets see how the tree itself is represented :

Threaded Binary Tree: Predecessors and successors

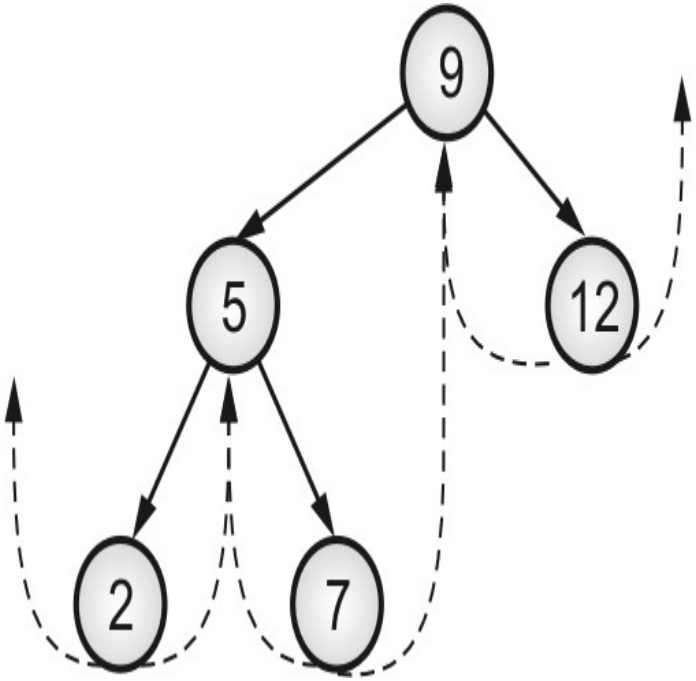
- value : T is the value of this node (e.g. 1, 2, A, B, etc.)
- parent : is the parent of this node.
- left : is the left child of this node.
- right : is the right child of this node.
- leftThread : is the in-order predecessor of this node.
- rightThread : is the inorder successor of this node.

Threaded Binary Tree: Predecessors and successors



- We start at the root of the tree, 9. Note that we don't visit(9) yet.
- From there we want to go to the minimum() node in the tree, which is 2 in this case. We then visit(2) and see that it has a rightThread, and thus we immediately know what its successor() is.
- We follow the thread to 5, which does not have any leaving threads. Therefore, after we visit(5), we go to the minimum() node in its right subtree, which is 7.

Threaded Binary Tree: Predecessors and successors



- We then visit(7) and see that it has a rightThread, which we follow to get back to 9. Now we visit(9), and after noticing that it has no rightThread, we go to the minimum() node in its right subtree, which is 12.

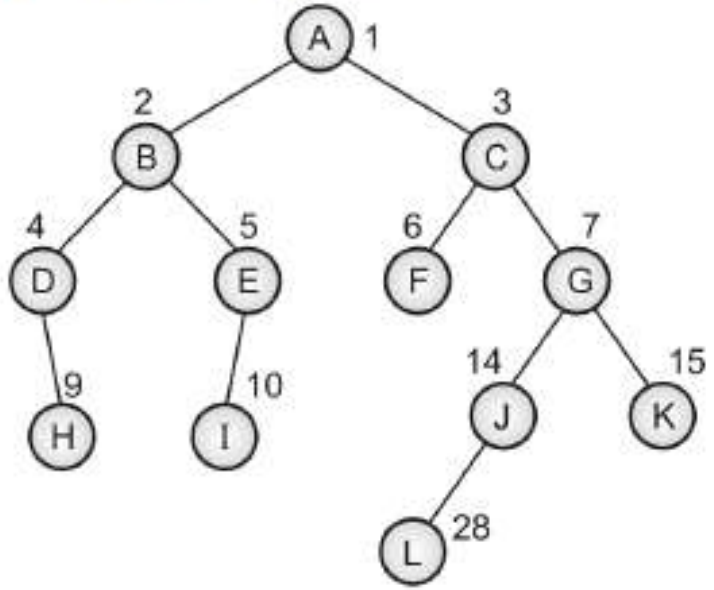
- This node has a rightThread that leads to nil, which signals that we have completed the traversal! We visited the nodes in order 2, 5, 7, 9, 12, which intuitively makes sense, as that is their natural increasing order.

Threaded Binary Tree: Predecessors and successors

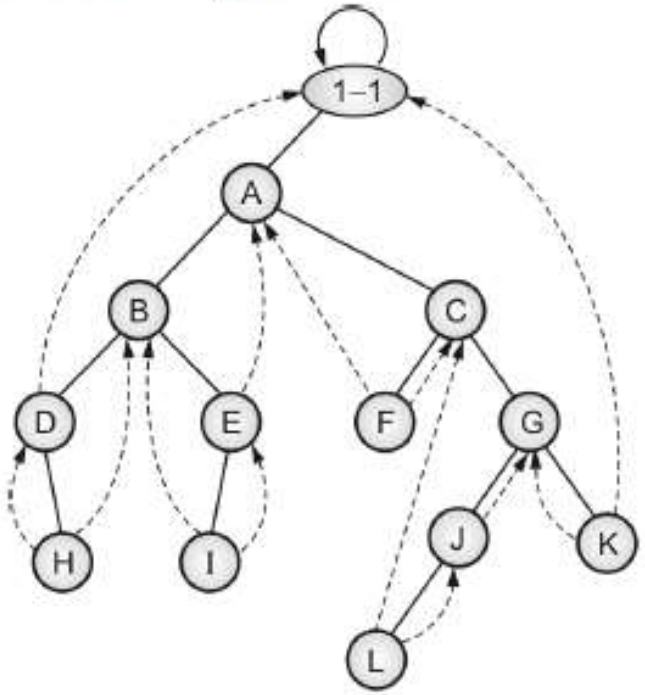
For the binary tree represented as an array, perform in-order threading on the tree :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
A	B	C	D	E	F	G		H	I				J	K													L

Step 1 : Tree for the given data.

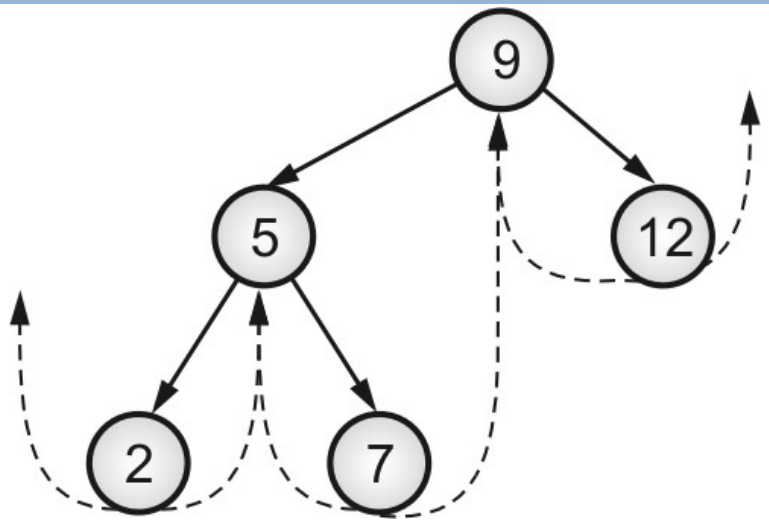


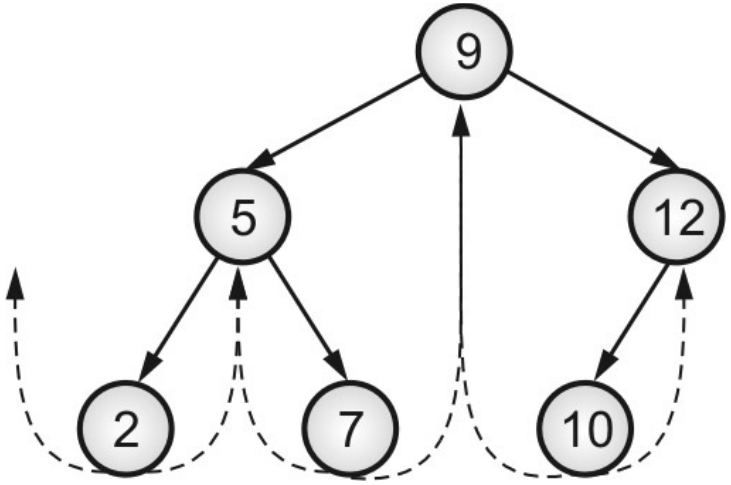
Step 2 : Threading of the tree.



Insertion and Deletion of Nodes: TBT

Inserting/deleting nodes becomes more complicated, as we have to continuously manage the leftThread and rightThread variables.





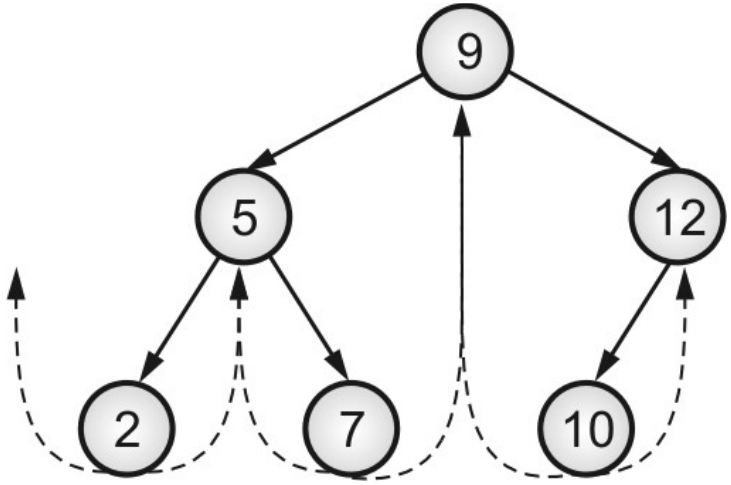
Suppose we insert 10 into this tree. The resulting graph would look like this :

Insertion and Deletion of Nodes: TBT

Here we have to maintain the threads between nodes. So we know that we want to insert 10 as 12's left child. The first thing we do is set 12's left child to 10, and set 10's parent to 12.

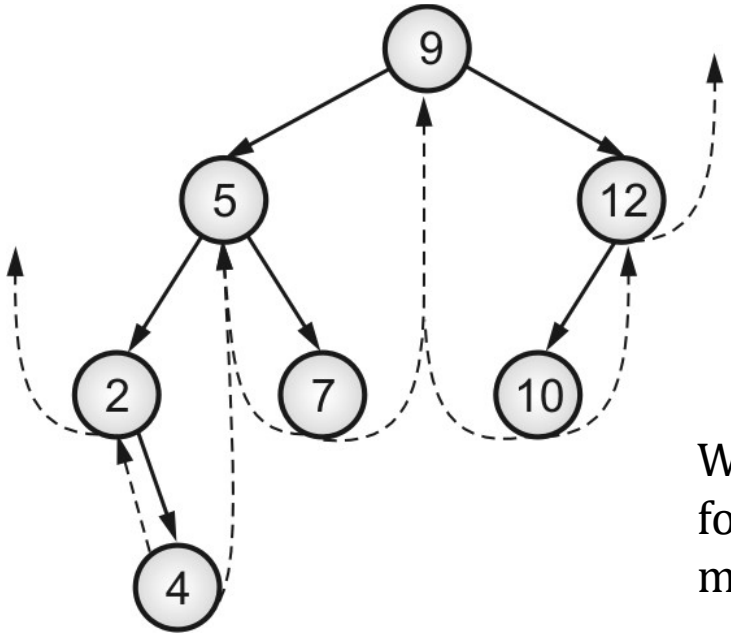
Because 10 is being inserted on the left, and 10 has no children of its own, we can safely set 10's rightThread to its parent 12.

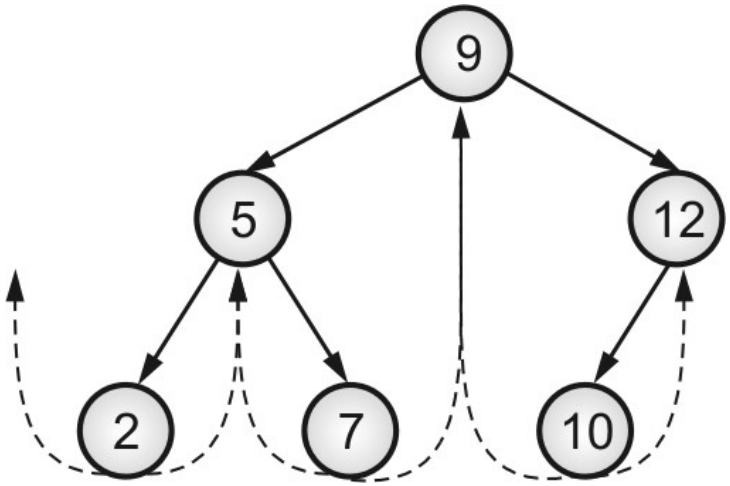
What about 10's leftThread? Because we know that 10 < 12, and 10 is the only left child of 12, we can safely set 10's leftThread to 12's (now outdated) leftThread. Finally we set 12's leftThread = nil, as it now has a left child.



Insertion and Deletion of Nodes: TBT

Let's now insert another node, **4, into the tree:**

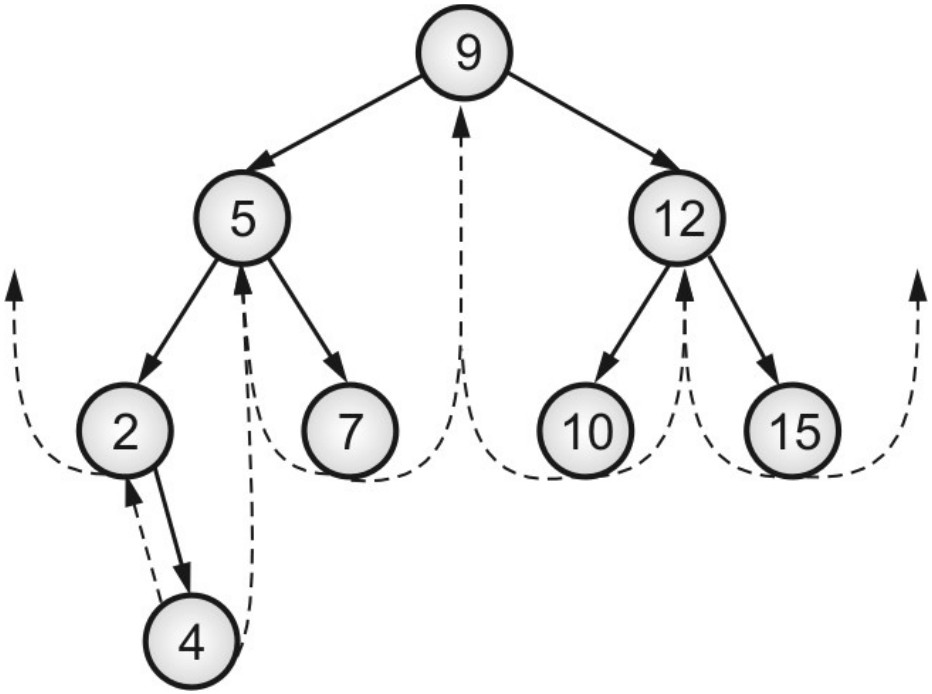


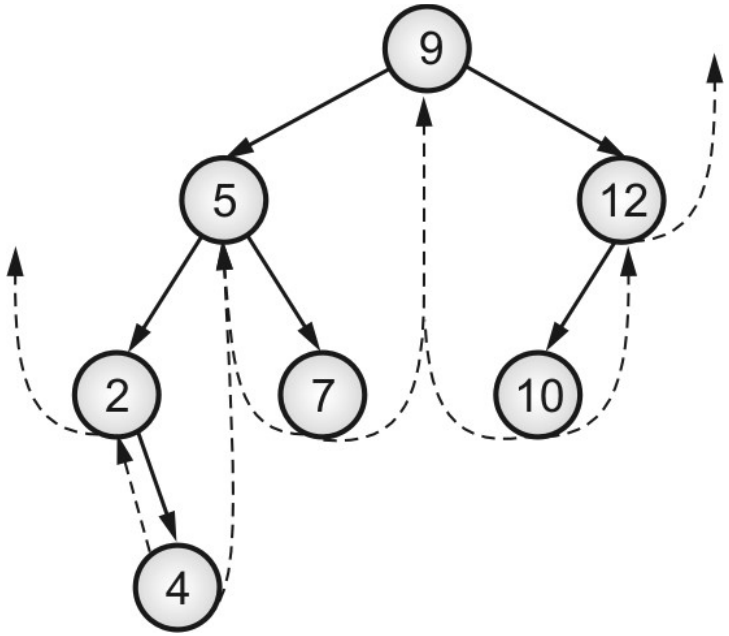


While we are inserting 4 as a right child, it follows the exact same process as above, but mirrored (swap left and right).

Insertion and Deletion of Nodes: TBT

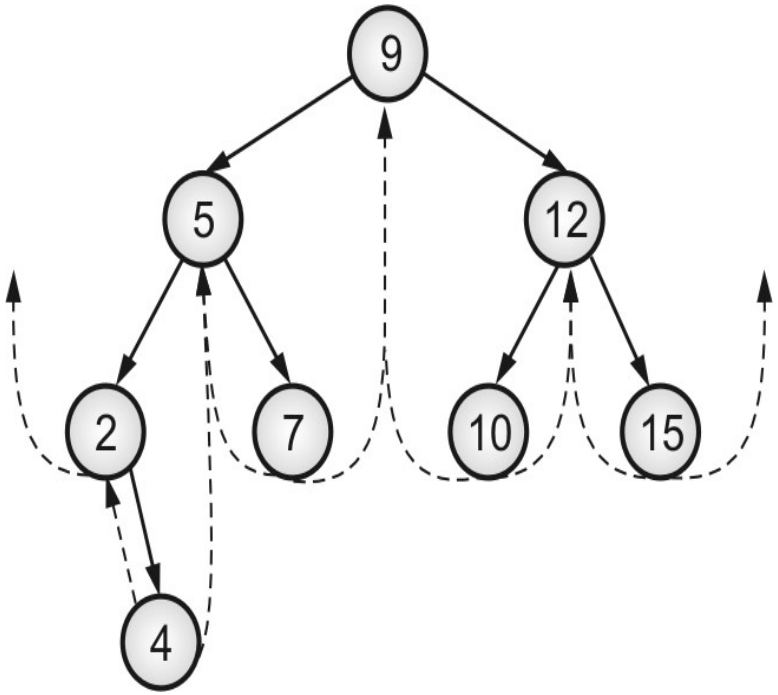
insert one final node, 15:





Insertion and Deletion of Nodes: TBT

Compared to insertion, deletion is a little more complicated. Let's start with something simple, like removing 7, which has no children :



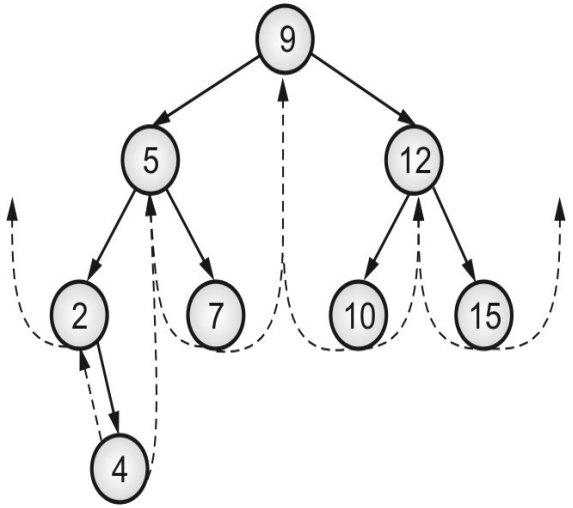
Before we can just throw 7 away, we have to perform some clean-up. In this case, because 7 is a right child and has no children itself, we can simply set the rightThread of 7's parent(5) to 7's (now outdated) rightThread.

Then we can just set 7's parent, left, right, leftThread, and rightThread to nil, effectively removing it from the tree.

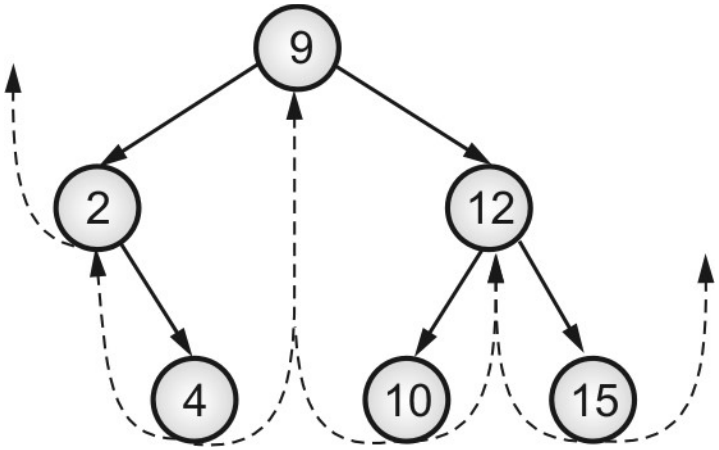
We also set the parent's rightChild to nil, which completes the deletion of this right child.

Insertion and Deletion of Nodes: TBT

Say we remove 5 from the tree:



5 has some children that we have to deal with. The core idea is to replace 5 with its first child, 2. To accomplish this, we of course set 2's parent to 9 and set 9's left child to 2. Note that 4's rightThread used to be 5, but we are removing 5, so it needs to change.



Two important properties of threaded binary trees:

• For the rightmost node m in the left subtree of any node n, m's rightThread is n.

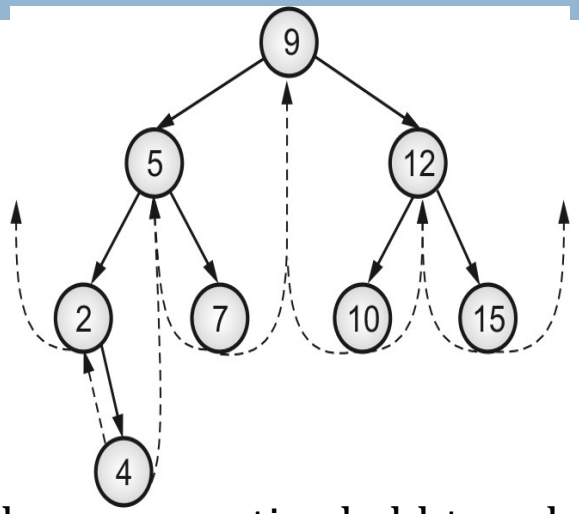
• For the leftmost node m in the right subtree of any node n, m's leftThread is n.

Insertion and Deletion of Nodes: TBT

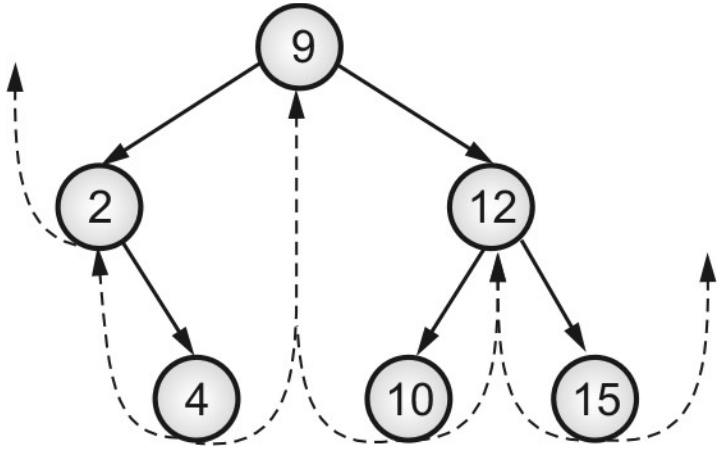
Two important properties of threaded binary trees:

• For the rightmost node m in the left subtree of any node n, m's rightThread is n.

• For the leftmost node m in the right subtree of any node n, m's leftThread is n.



• Note how these properties held true before the removal of 5, as 4 was the rightmost node in 5's left subtree.



• In order to maintain this property, we must set 4's rightThread to 9, as 4 is now the rightmost node in 9's left subtree

.

• To completely remove 5, all we now have to do is set 5's parent, left, right, leftThread, and rightThread to nil.