

# Non-Linear Data Structures: Trees & Graphs

Dr. Ayesha Hakim

# Outline

Tree – concept

General tree

Types of trees

Binary tree: representation, operation

Binary tree traversal

Binary search tree

BST- The data structure and implementation

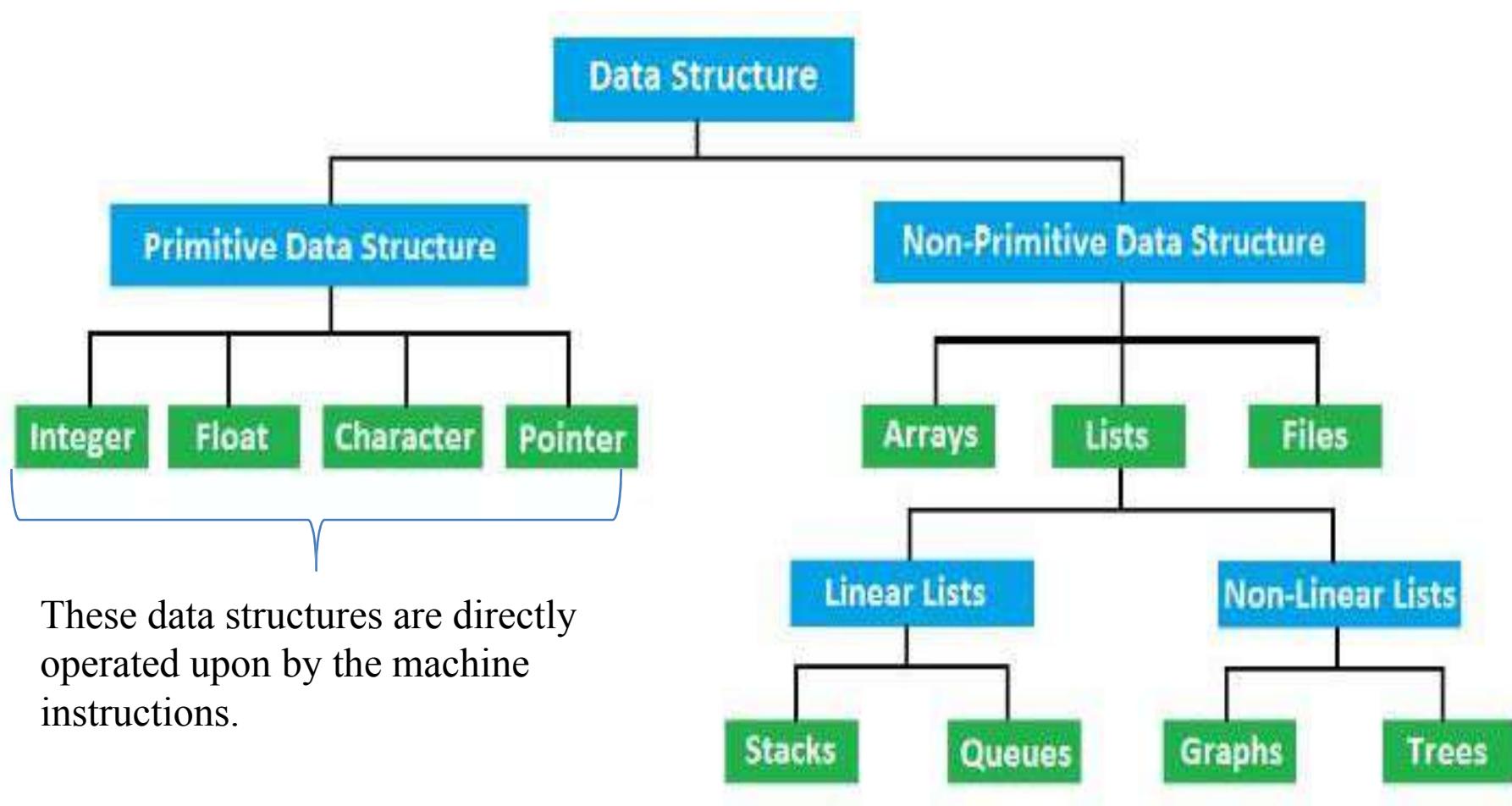
Threaded binary trees

Search Trees –

AVL tree, Multiway Search Tree, B Tree, B+ Tree, and Trie

Applications/Case study of trees

# Types of data structures



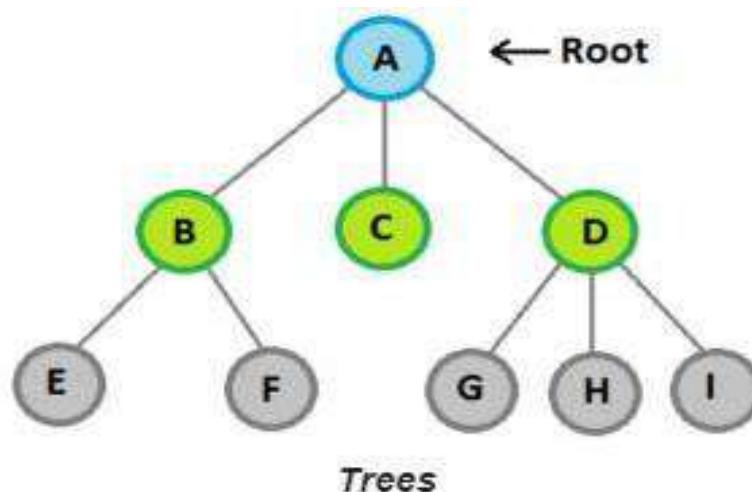
# Tree – a Hierarchical Data Structure

- Trees are non linear data structure that can be represented in a hierarchical manner.
  - A tree contains a finite non-empty set of elements.
  - Any two nodes in the tree are connected with a relationship of parent-child.
  - Every individual elements in a tree can have any number of sub trees.

# Types of Non-Linear Data Structures

- **Trees-**

- A tree stores a collection of items in an abstract **hierarchical** way.
- Each node is linked to other nodes and can have multiple sub-values also known as children.



# Why Tree is considered a non-linear data structure?

- The data in a tree are **not stored in a sequential manner** i.e, they are not stored linearly.
- Instead, they are arranged on multiple levels or we can say it is a hierarchical structure.
- For this reason, the tree is considered to be a non-linear data structure.

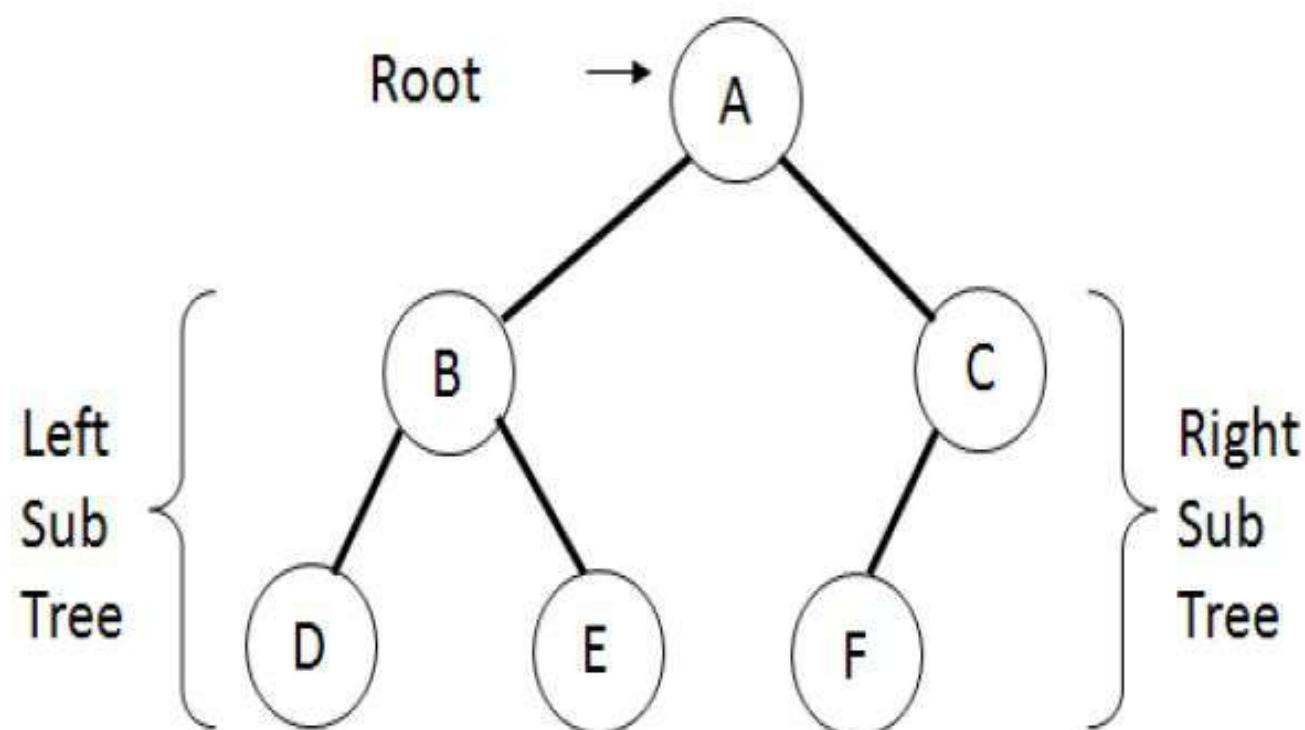
# Types of data structures

- *A Tree has the following characteristics :*
  - The top item in a hierarchy of a tree is referred as the **root** of the tree.
  - The remaining data elements are partitioned into a number of mutually exclusive subsets and they itself a tree and are known as the **subtree**.
  - Unlike natural trees trees in the data structure always grow in length towards the bottom.

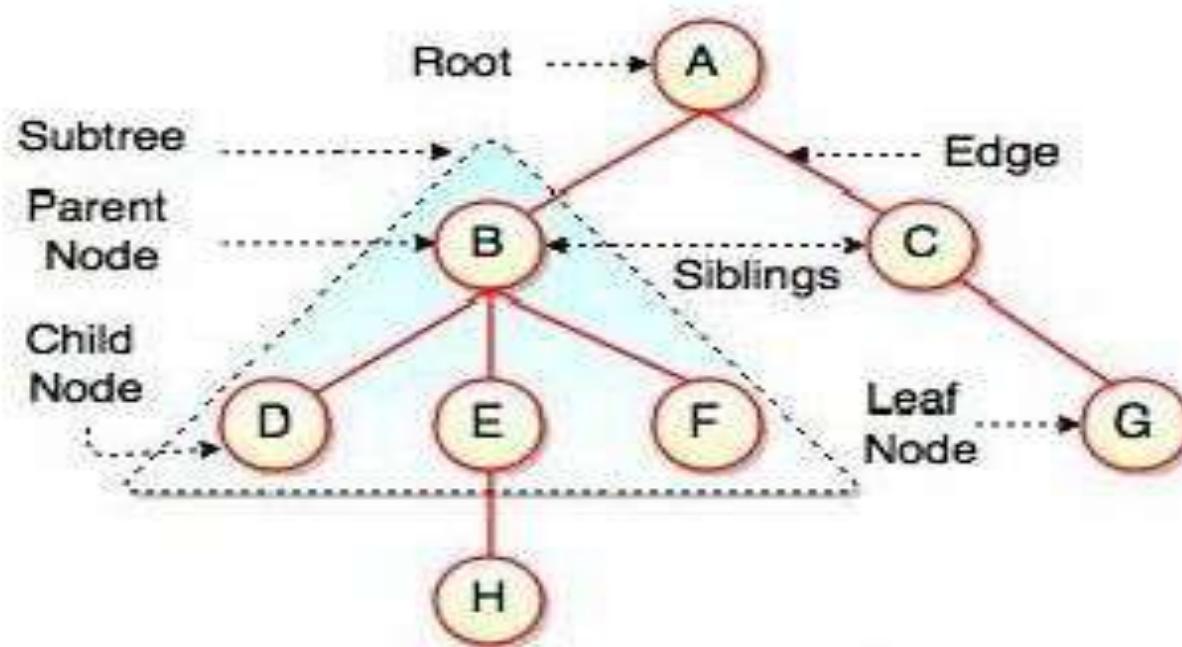
# What are trees?

- Tree is a hierarchical data structure which stores the information naturally in the form of hierarchy style.
- Tree is one of the most powerful and advanced data structures.
- It is a non-linear data structure compared to arrays, linked lists, stack and queue.
- It represents the nodes connected by edges.

# An Example of a Tree



# What are trees?



- The above figure represents structure of a tree. Tree has 2 subtrees.
- A is a parent of B and C.
- B is called a child of A and also parent of D, E, F.

# What are trees?

Field	Description
Root	Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent.
Parent Node	Parent node is an immediate predecessor of a node.
Child Node	All immediate successors of a node are its children.
Siblings	Nodes with the same parent are called Siblings.
Path	Path is a number of successive edges from source node to destination node.
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.
Height of Tree	Height of tree represents the height of its root node.
Depth of Node	Depth of a node represents the number of edges from the tree's root node to the node.
Degree of Node	Degree of a node represents a number of children of a node.
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.

# Tree – Basic Terminology

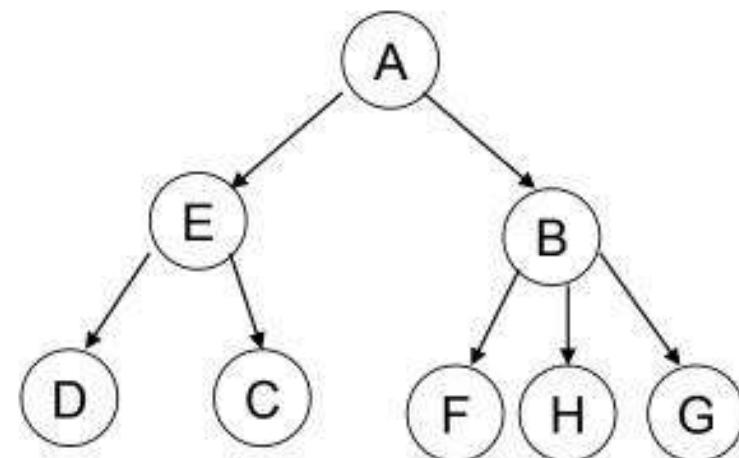
- **Root** : The basic node of all nodes in the tree. All operations on the tree are performed with passing root node to the functions.
- **Child** : a successor node connected to a node is called child. A node in binary tree may have at most two children.
- **Parent** : a node is said to be parent node to all its child nodes.
- **Leaf** : a node that has no child nodes.
- **Siblings** : Two nodes are siblings if they are children to the same parent node.

## Tree – Basic Terminology Contd...

- **Ancestor** : a node which is parent of parent node ( A is ancestor node to D,E and F ).
- **Descendent** : a node which is child of child node ( D, E and F are descendent nodes of node A )
- **Level** : The distance of a node from the root node, The root is at level – 0,( B and C are at Level 1 and D, E, F have Level 2 ( highest level of tree is called **height** of tree )
- **Degree** : The number of nodes connected to a particular parent node.

# Introduction

*A tree, is a finite set of nodes together with a finite set of directed edges (normally omits) that define parent-child relationships. Each directed edge connects a parent to its child. Example:*



Nodes={A,B,C,D,E,f,G,H}

Edges={(A,B),(A,E),(B,F),(B,G),(B,H),  
(E,C),(E,D)}



**SOMAIYA**

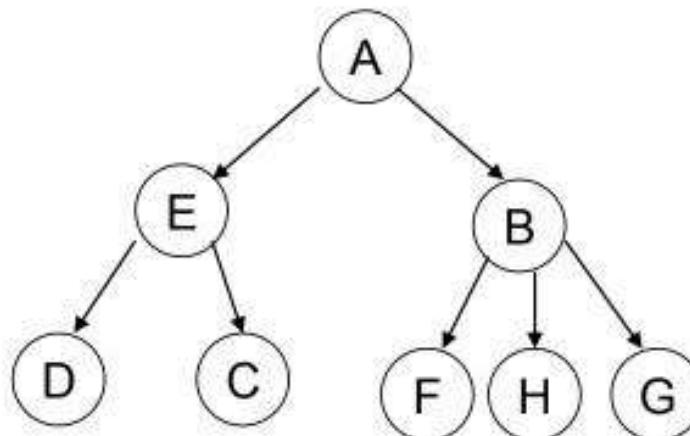
VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

# Introduction

- A directed path from node  $m_1$  to node  $m_k$  is a list of nodes  $m_1, m_2, \dots, m_k$  such that each is the parent of the next node in the list.
- The length of a path is  $k - 1$  i.e the number of edges on the path.

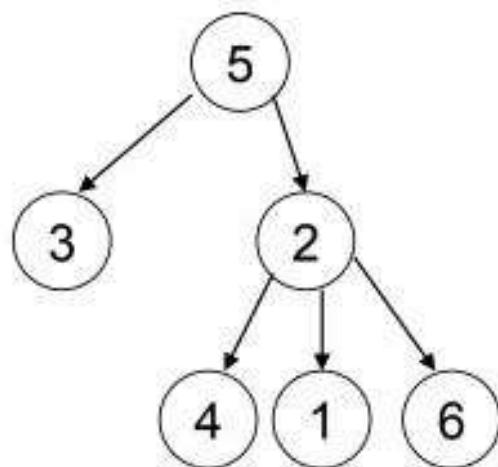
*Example:* A, E, C is a directed path of length 2.



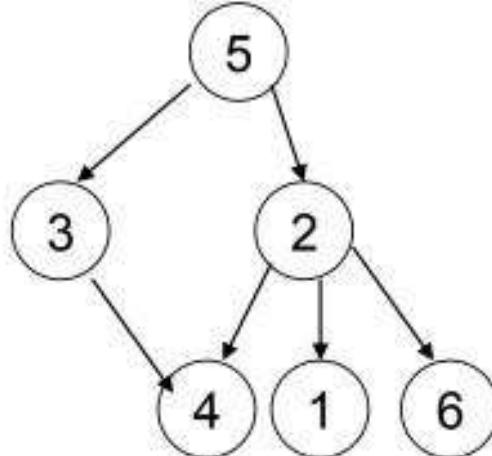
# Introduction

A tree satisfies the following properties:

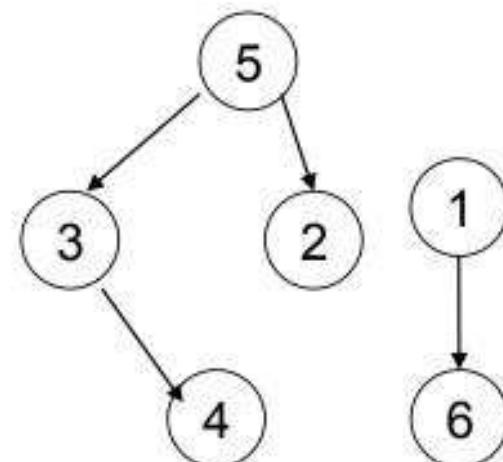
1. It has one designated node, called the root, that has no parent.
2. Every node, except the root, has exactly one parent.
3. A node may have zero or more children.
4. There is a unique directed path from the root to each node.



(a) tree

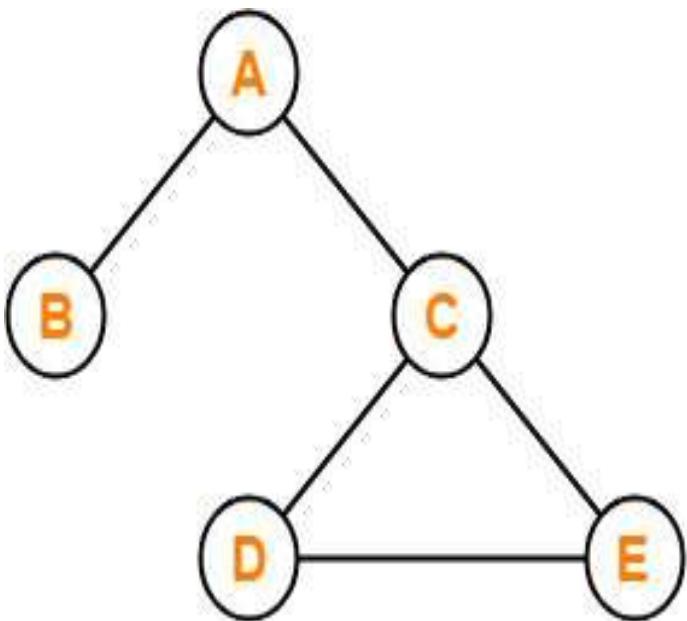


(b) Not a tree

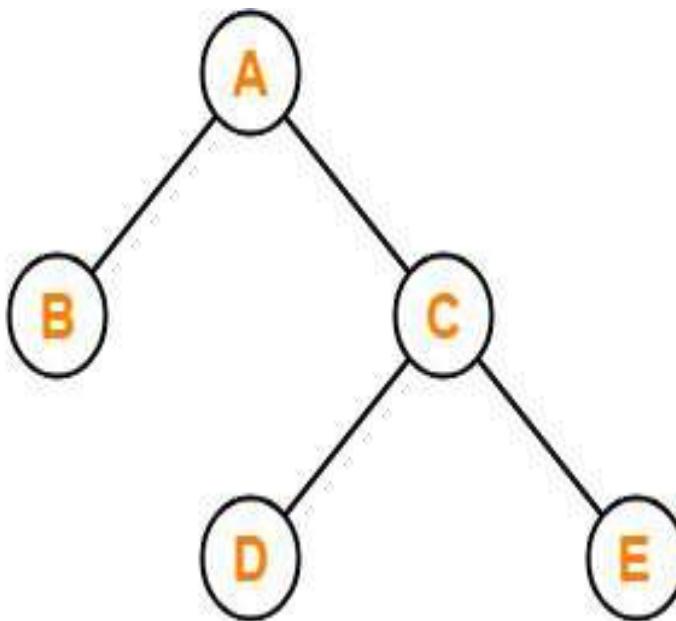


(c) Not a tree

# A tree is a connected graph without any circuits.

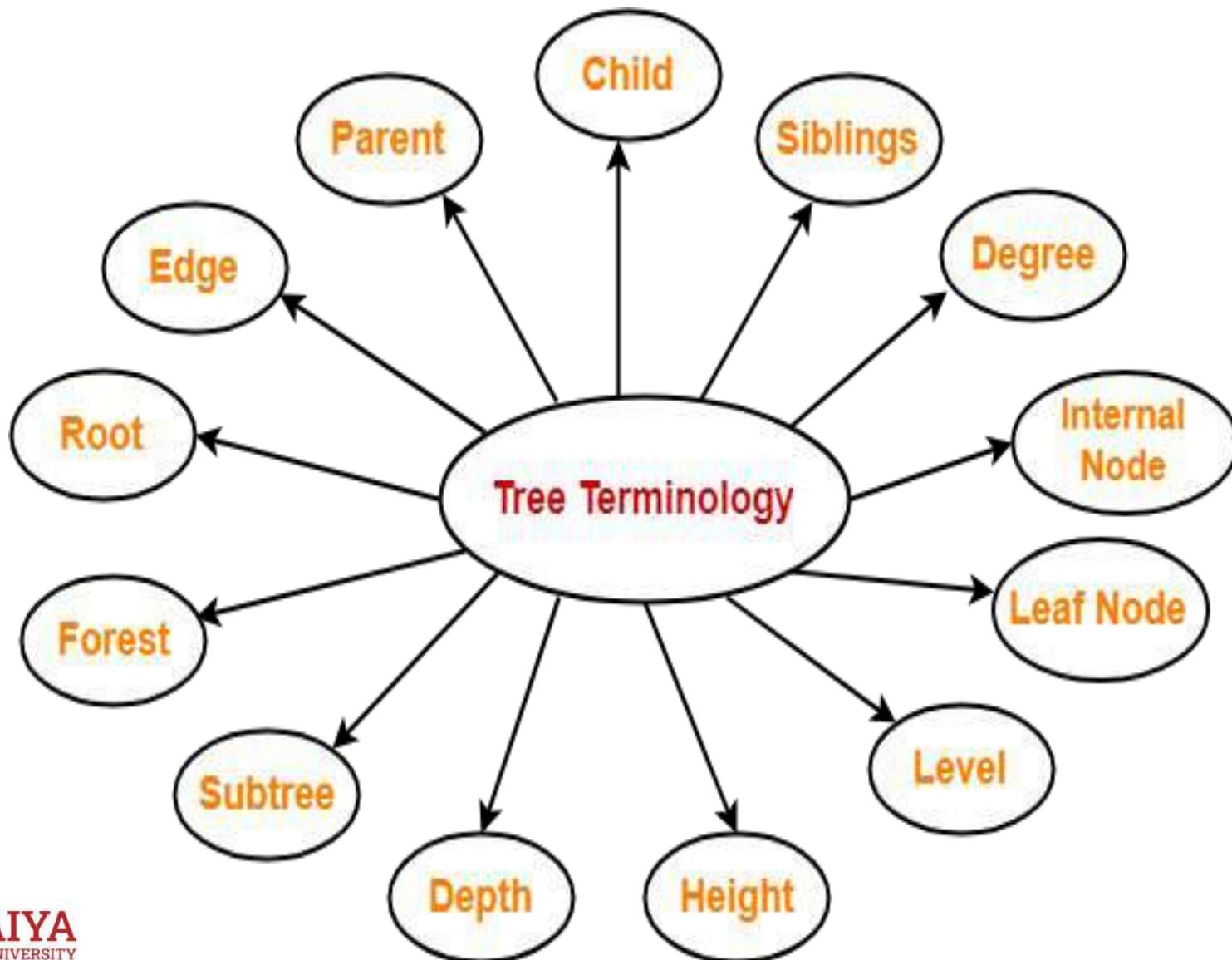


This graph is not a Tree



This graph is a Tree

# Tree terminology



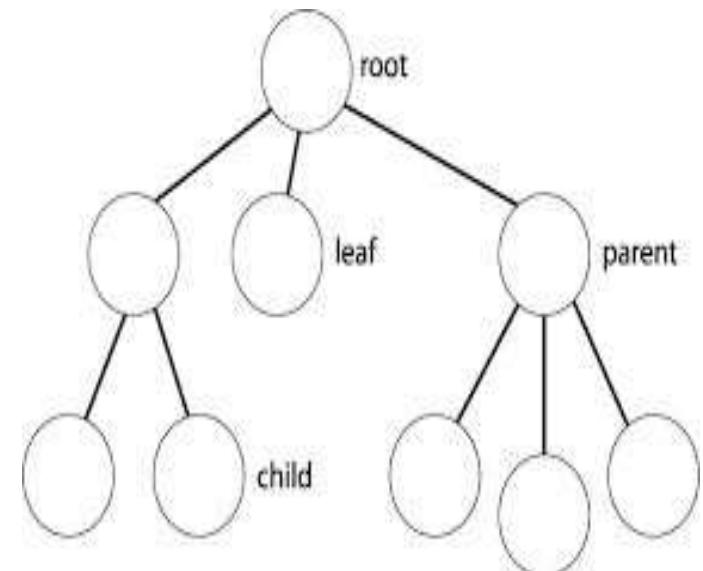
# Properties

The important properties of tree data structure are-

- There is one and only one path between every pair of vertices in a tree.
- A tree with  $n$  vertices has exactly  $(n-1)$  edges.
- A graph is a tree if and only if it is minimally connected.
- Any connected graph with  $n$  vertices and  $(n-1)$  edges is a tree.

# Tree :Basic Terminology-

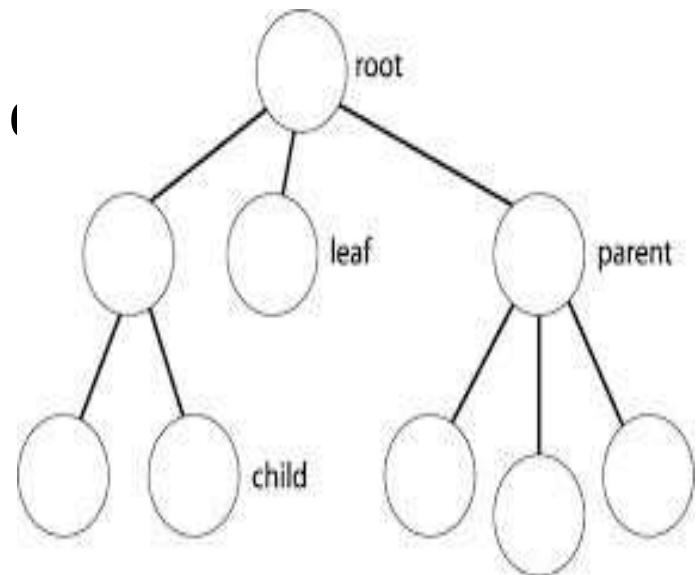
- Parent : Immediate predecessor of a node
- Child: Immediate successor of a node
- Siblings : Nodes with the same parent



# Tree :Basic Terminology-

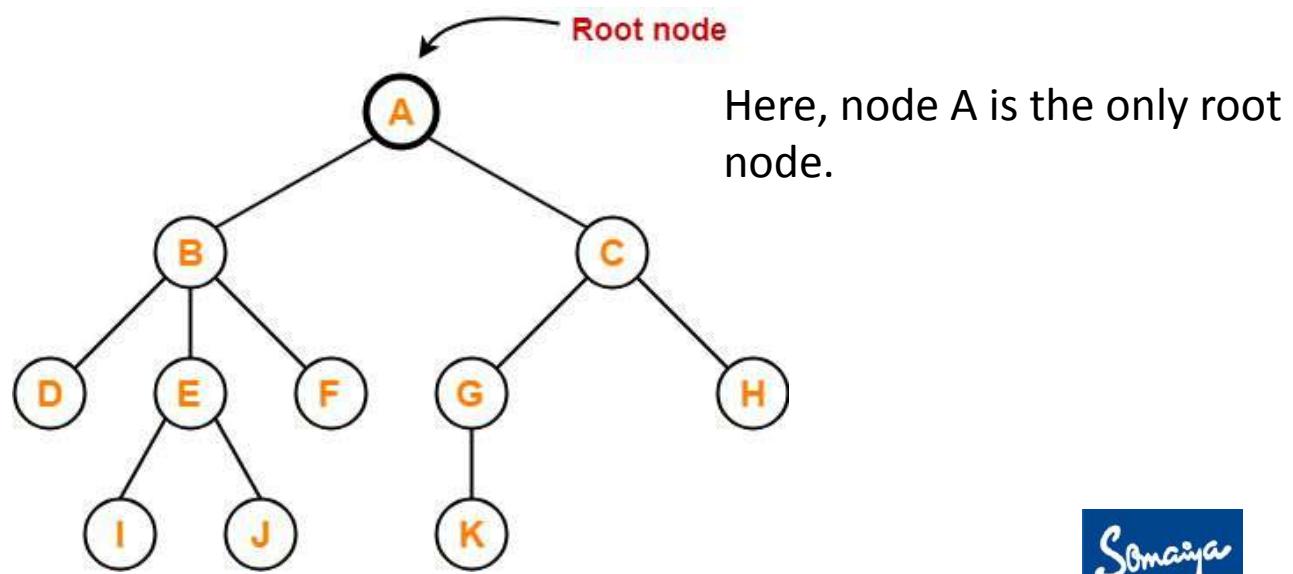
- **Node-**

- Each data item in a tree.
- Specifies the data and links to other nodes.



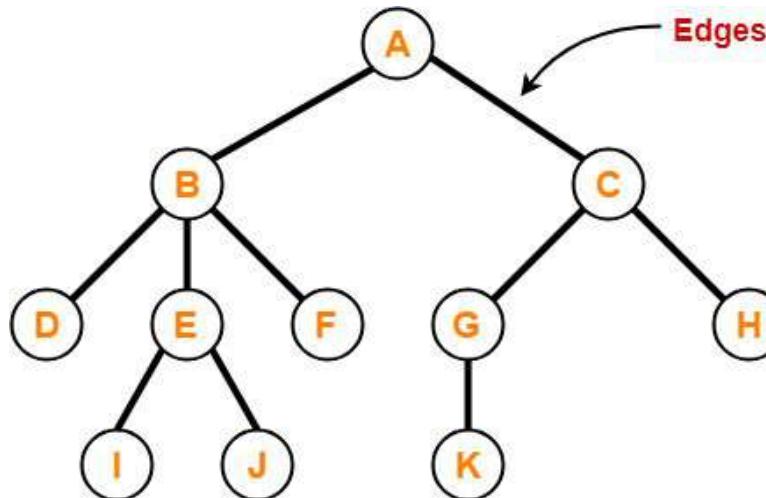
# Root

- The first node from where the tree originates is called as a **root node**.
- In any tree, there must be only one root node.
- Entire tree is referenced through it.
- We can never have multiple root nodes in a tree data structure.



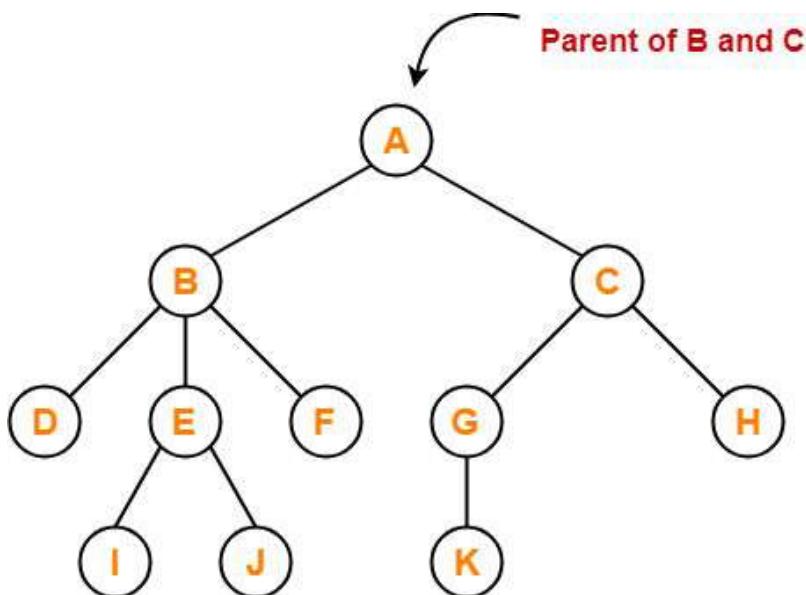
# Edge

- The connecting link between any two nodes is called as an **edge**.
- In a tree with  $n$  number of nodes, there are exactly  $(n-1)$  number of edges.



# Parent

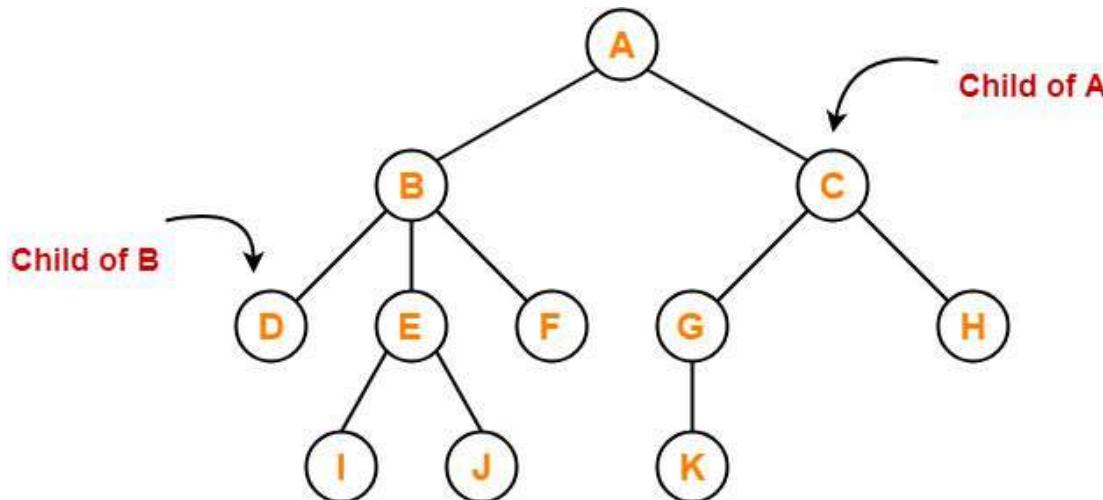
- The node which has a branch from it to any other node is called as a **parent node**.
- The node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.



Node A is the parent of nodes B and C  
Node B is the parent of nodes D, E and F  
Node C is the parent of nodes G and H  
Node E is the parent of nodes I and J  
Node G is the parent of node K

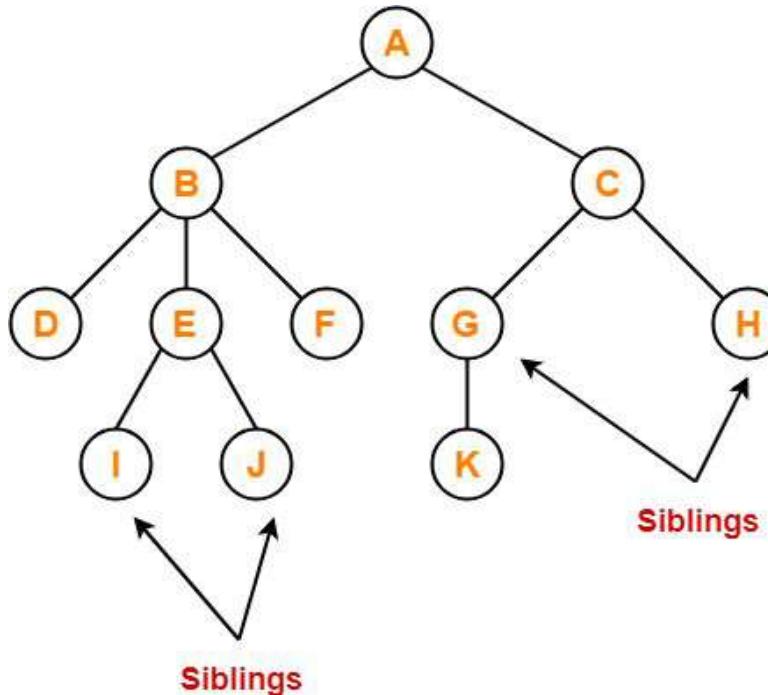
# Child

- The node which is a descendant of some node is called as a **child node**.
- All the nodes except root node are child nodes.



# Siblings

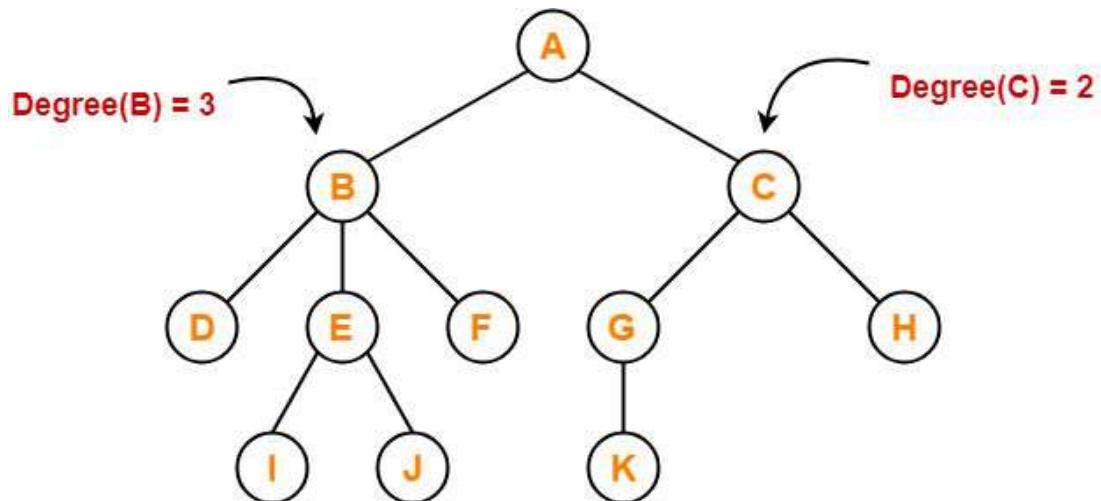
- Nodes which belong to the same parent are called as **siblings**.
- In other words, nodes with the same parent are sibling nodes.



Nodes B and C are siblings  
Nodes D, E and F are siblings  
Nodes G and H are siblings  
Nodes I and J are siblings

# Degree

- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.



Here **Degree of B is 3**

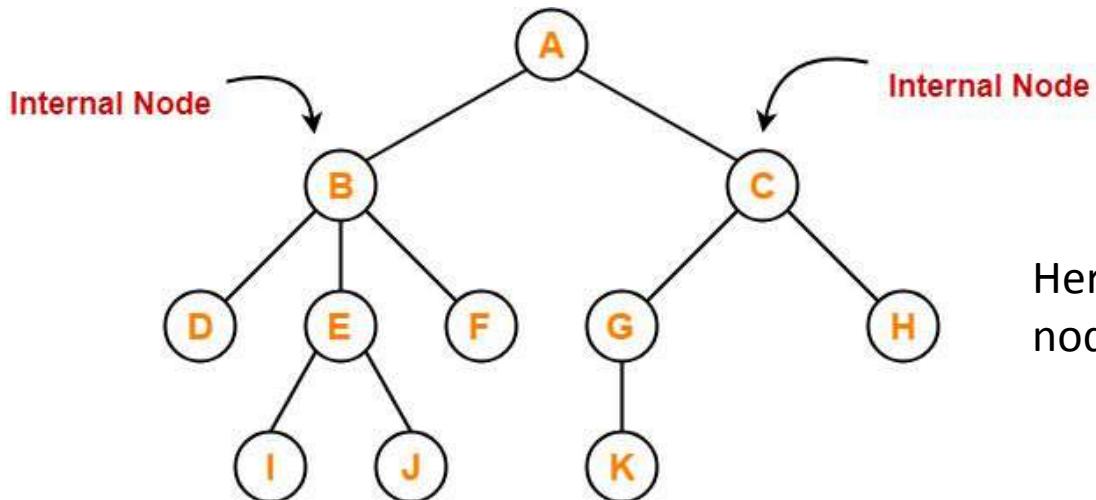
Here **Degree of A is 2**

Here **Degree of F is 0**

- In any tree, '**Degree**' of a node is total number of children it has.

# Internal node

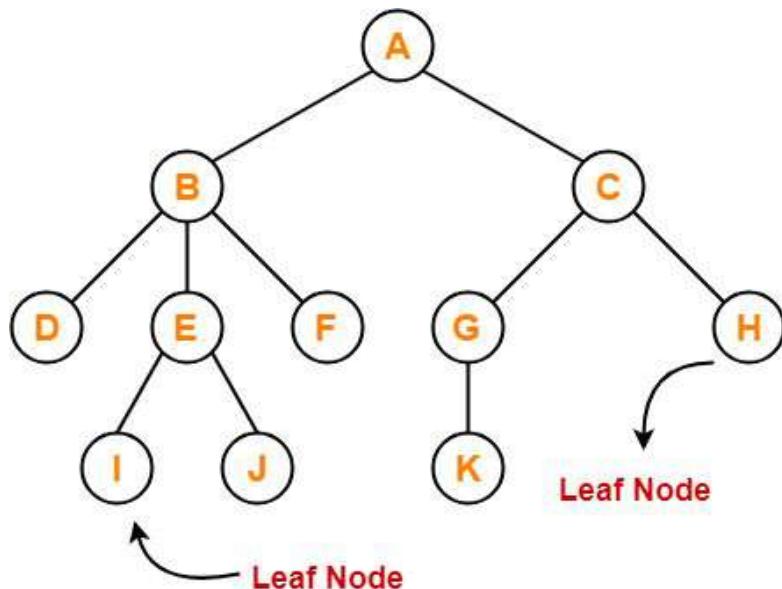
- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node.



Here, nodes A, B, C, E and G are internal nodes.

# Leaf node

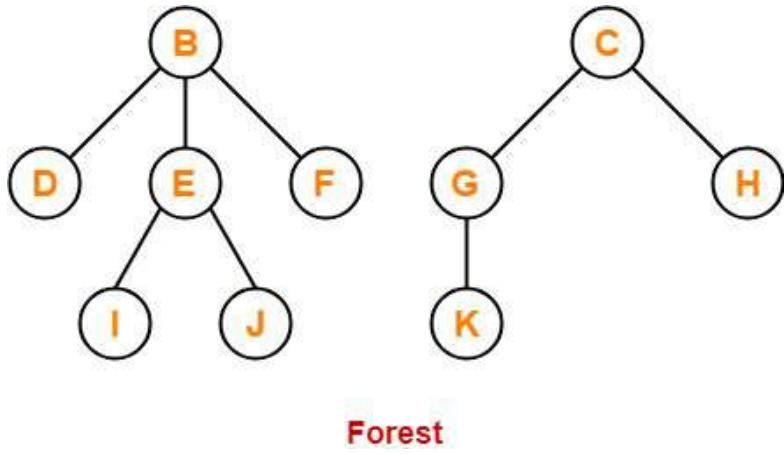
- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.



Here, nodes D, I, J, F, K and H are leaf nodes.

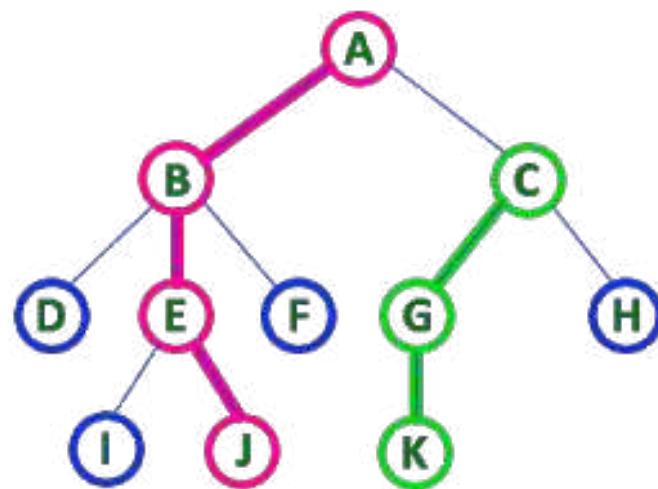
# Forest

- A forest is a set of disjoint trees.



# Path

- The sequence of consecutive edges from source node to destination node.



- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is

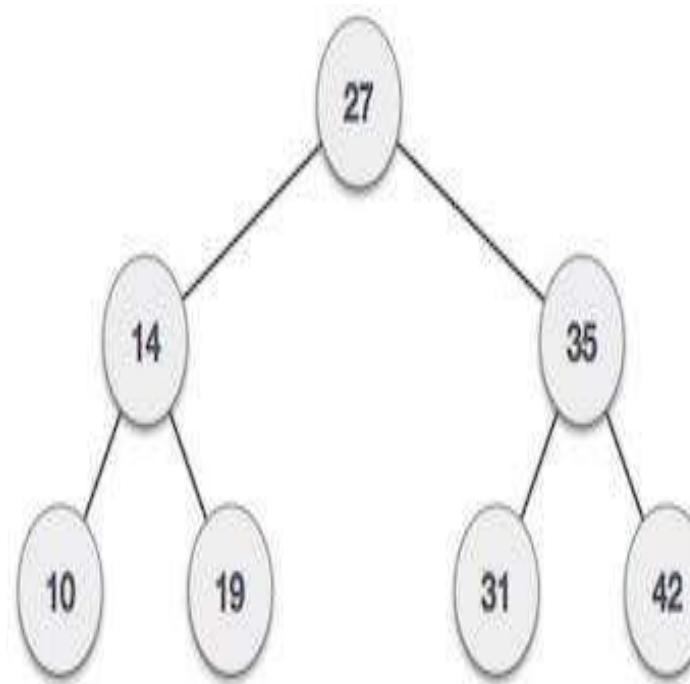
A - B - E - J

Here, 'Path' between C & K is

C - G - K

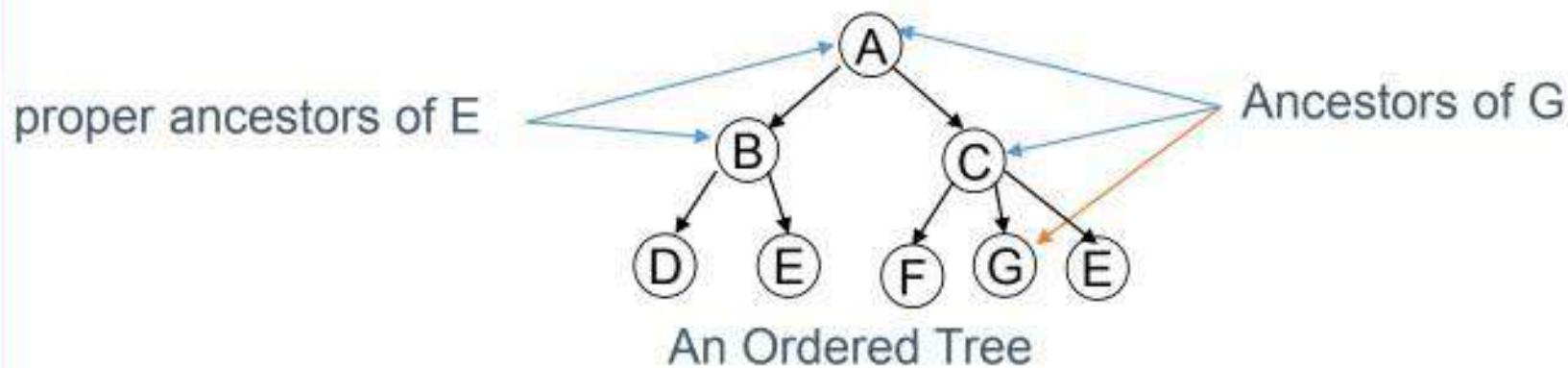
# Key

- Key represents a value of a node based on which a search operation is to be carried out for a node.



# Tree Terminology

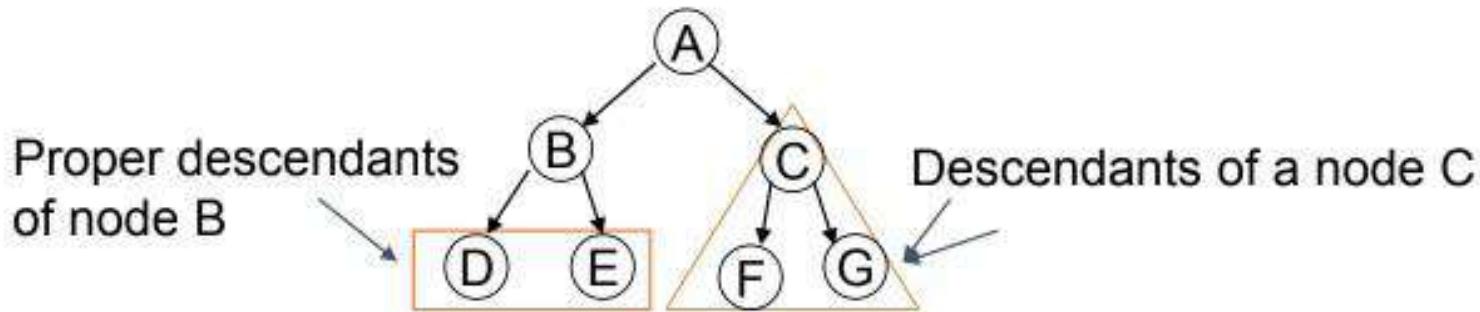
- Ordered tree: A tree in which the children of each node are linearly ordered (usually from left to right).



- Ancestor** of a node  $v$ : Any node, including  $v$  itself, on the path from the root to the node.
- Proper ancestor** of a node  $v$ : Any node, excluding  $v$ , on the path from the root to the node.

# Tree Terminology (Contd.)

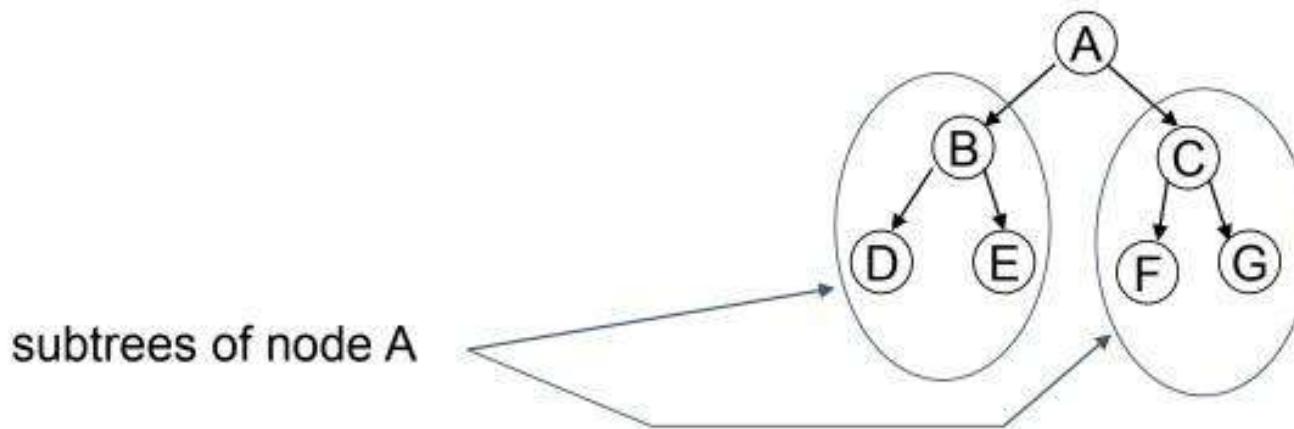
- **Descendant** of a node  $v$ : Any node, including  $v$  itself, on any path from the node to a leaf node (i.e., a node with no children).



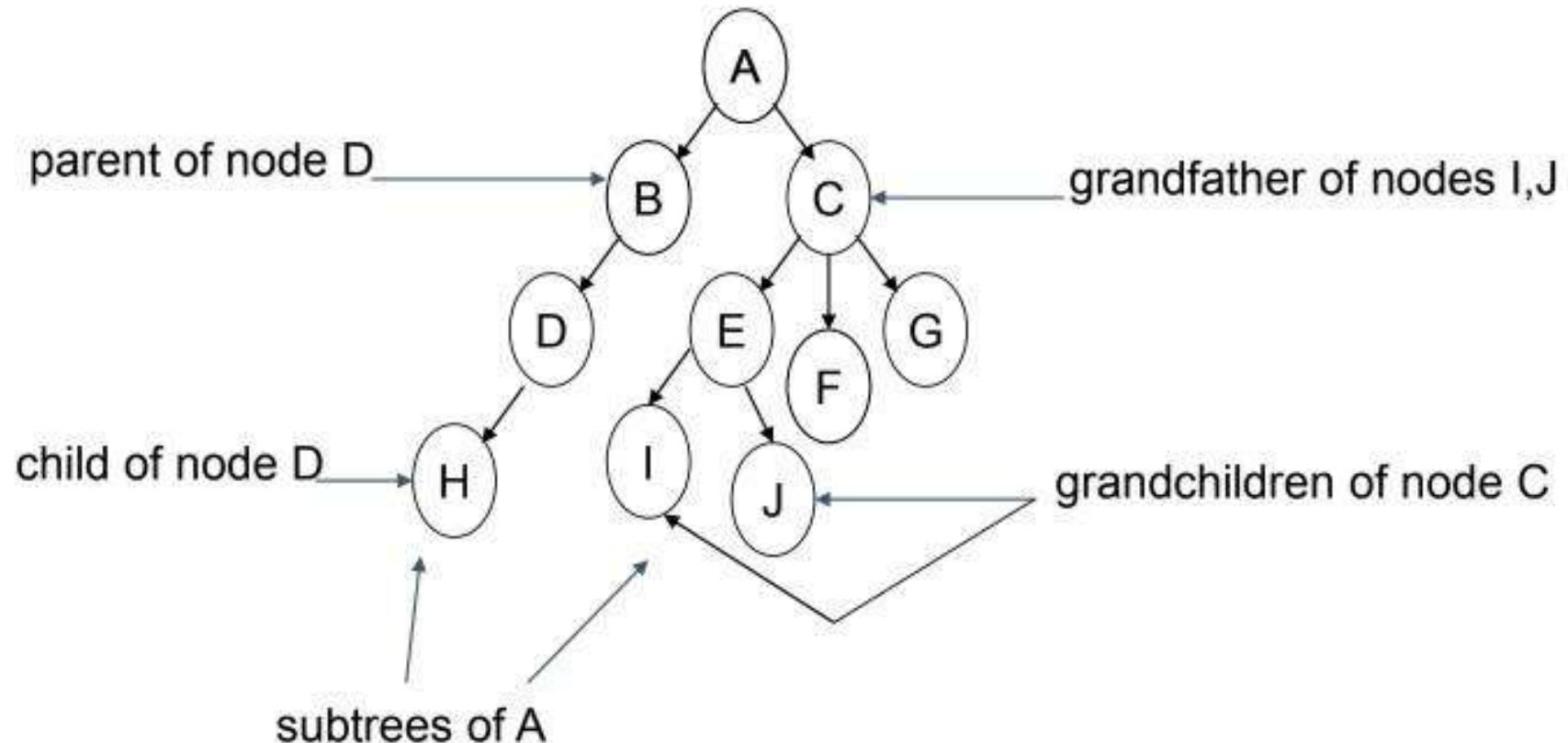
- **Proper descendant** of a node  $v$ : Any node, excluding  $v$ , on any path from the node to a leaf node.

# Tree Terminology (Contd.)

- **Subtree** of a node  $v$ : A tree rooted at a child of  $v$ .



# Tree Terminology (Contd.)



# Tree Terminology (Contd.)

- **Degree:** The number of subtrees of a node

*Each of node D and B has degree \_1\_*

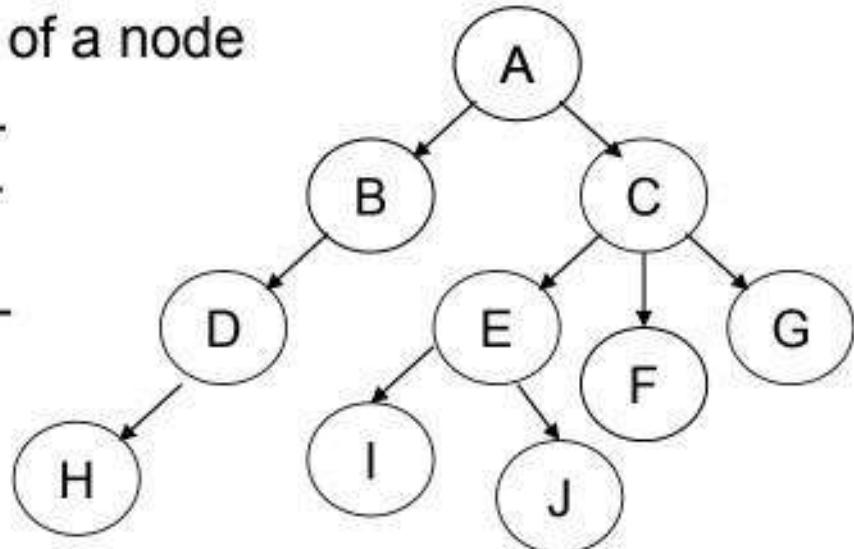
*Each of node A and E has degree \_2\_*

*Node C has degree \_3\_*

*Each of node F,G,H,I,J has degree \_0\_*

- **Leaf:** A node with degree 0.

*Leaf nodes are \_H, I,J,F,G\_*



**Internal** or interior node: a node with degree greater than 0.

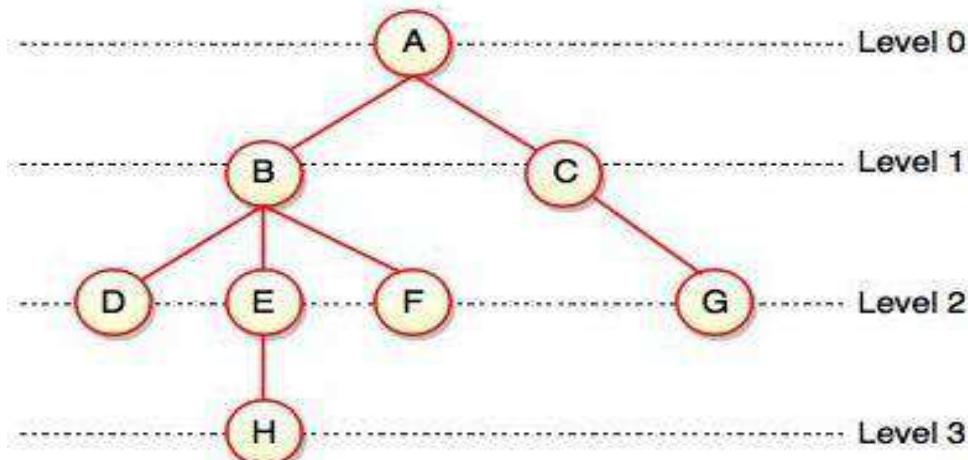
Internal Nodes are \_\_\_\_\_

- **Siblings:** Nodes that have the same parent.

Siblings of A \_\_\_\_\_ and Siblings of E \_\_\_\_\_ F, G \_\_\_\_\_

- **Size:** The number of nodes in a tree, size of this tree is \_\_\_\_\_ 10 \_\_\_\_\_

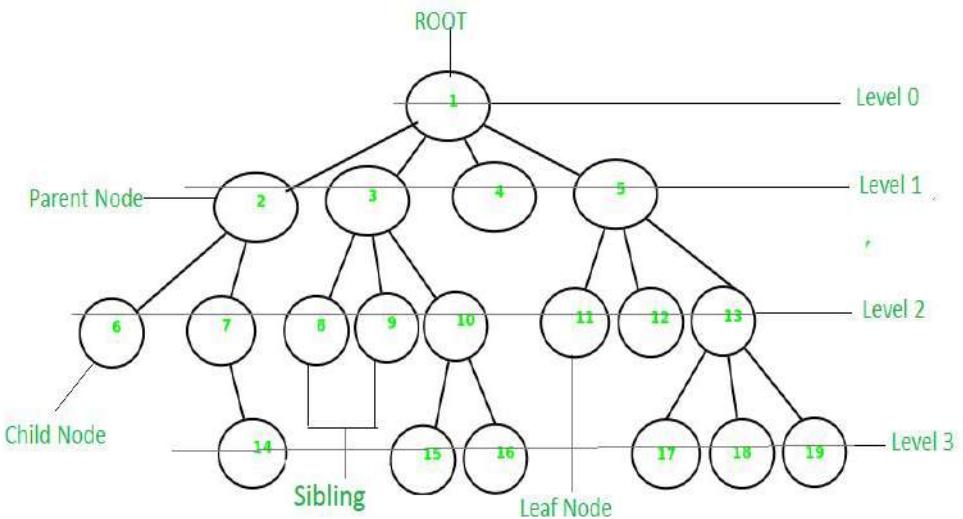
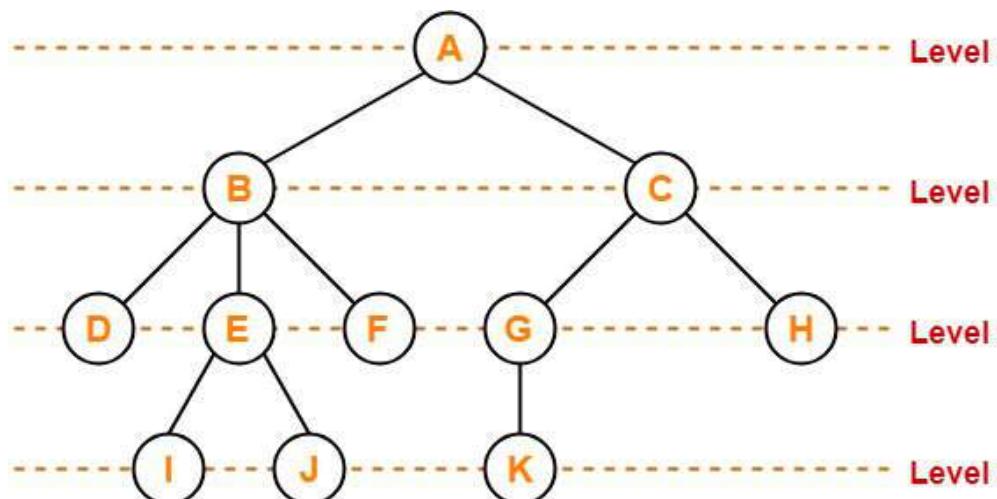
- **Levels of a node:** Levels of a node represents the number of connections between the node and the root. It represents generation of a node. If the root node is at level 0, its next node is at level 1, its grand child is at level 2 and so on. Levels of a node can be shown as follows:



- **Levels of a node:**
  - If node has no children, it is called **Leaves** or **External Nodes**.
  - Nodes which are not leaves, are called **Internal Nodes**. Internal nodes have at least one child.
  - A tree can be empty with no nodes or a tree consists of one node called the **Root**.

# Level

- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.

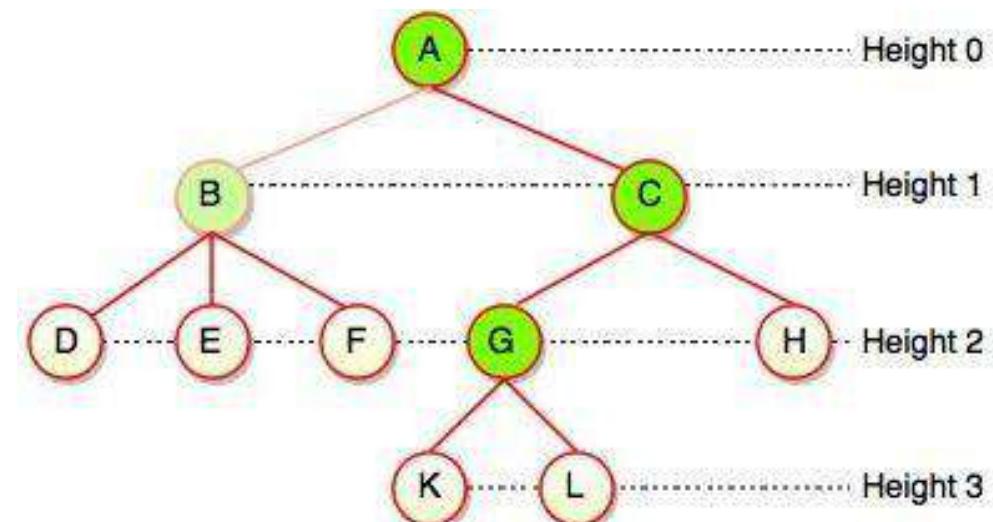


## Height of a Node

- height of a node is a number of edges on the longest path between that node and a leaf. Each node has height.
- In the above figure, A, B, C, D can have height. Leaf cannot have height as there will be no path starting from a leaf. Node A's height is the number of edges of the path to K not to D. And its height is 3.

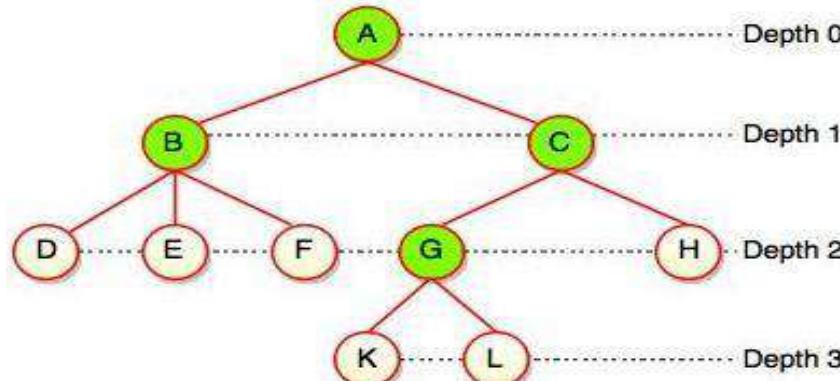
- **Height of a Node:**

- Height of a node defines the longest path from the node to a leaf.
- Path can only be downward.



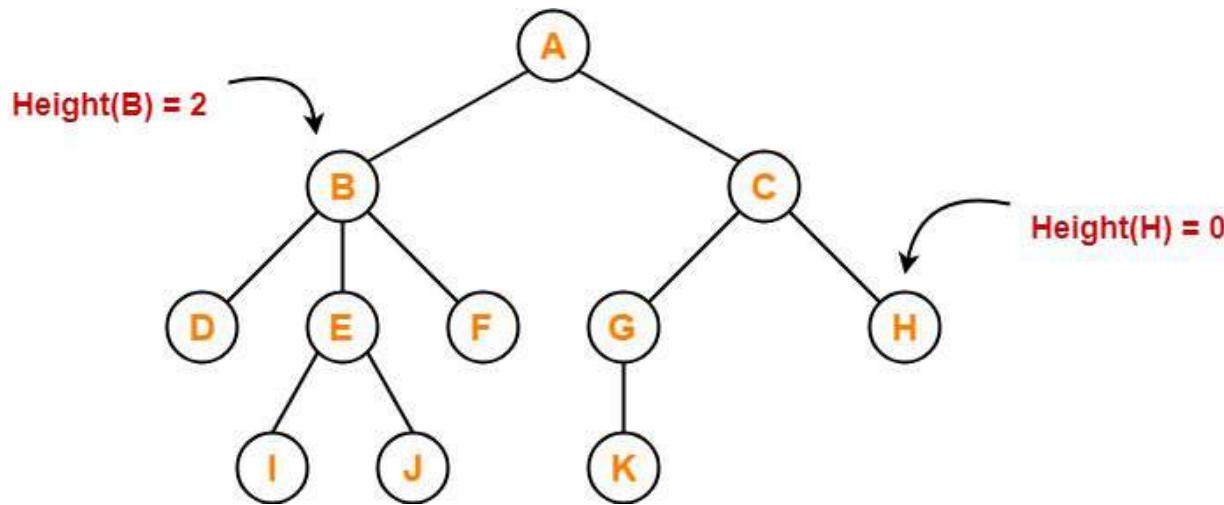
## • Depth of a Node

- While talking about the height, it locates a node at bottom where for depth, it is located at top which is root level and therefore we call it depth of a node.
- In the above figure, Node G's depth is 2. In depth of a node, we just count how many edges between the targeting node & the root and ignoring the directions.



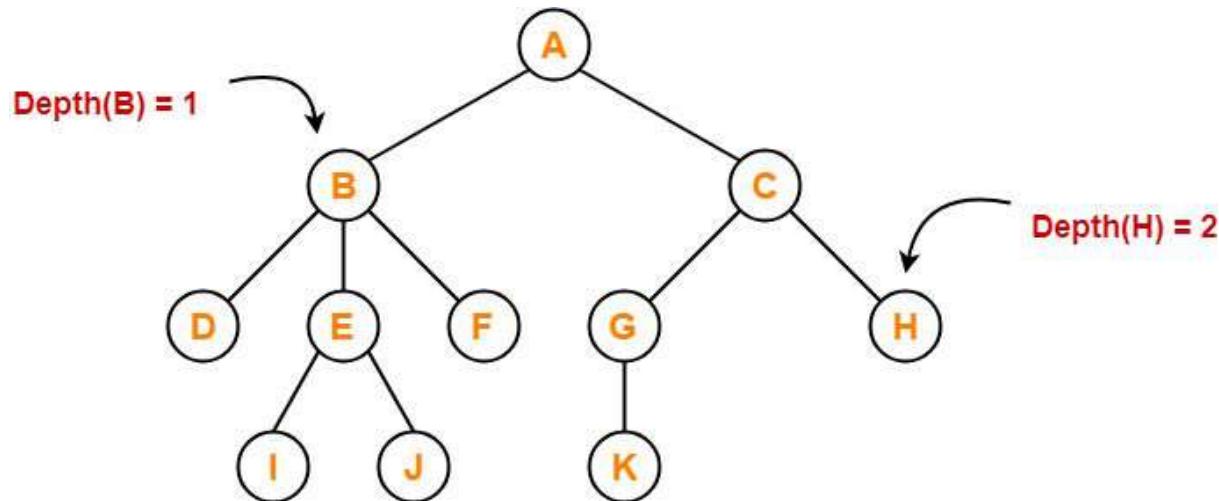
# Height

- Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.
- **Height of a tree** is the height of root node.
- Height of all leaf nodes = 0



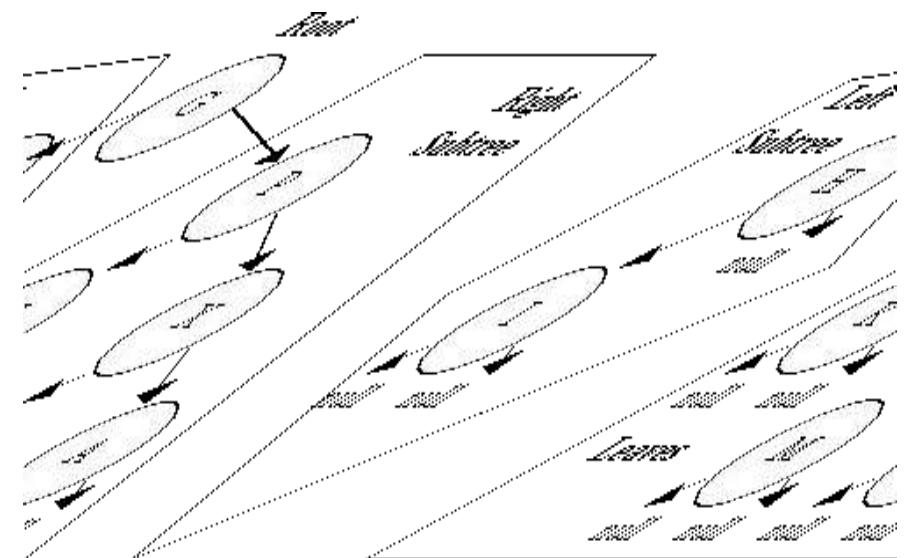
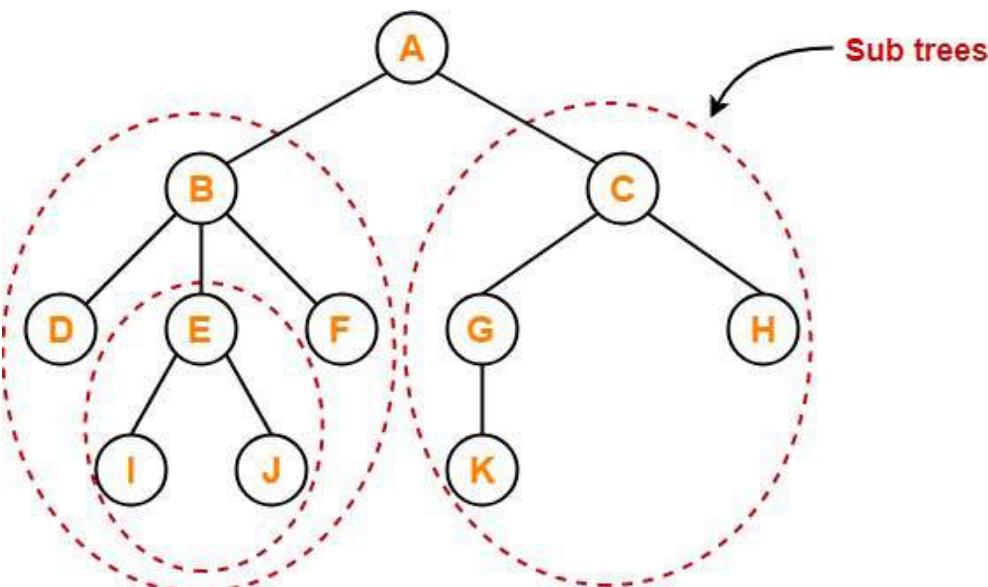
# Depth

- Total number of edges from root node to a particular node is called as **depth of that node**.
- **Depth of a tree** is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms “level” and “depth” are used interchangeably.



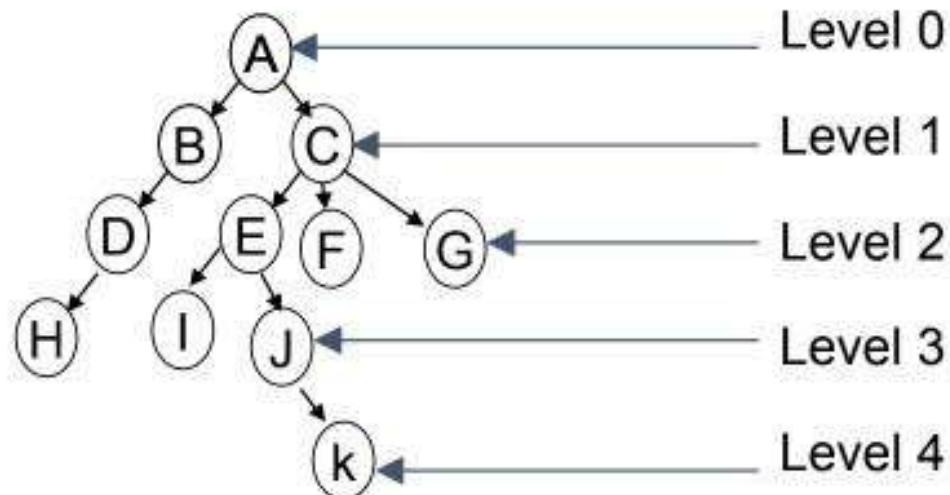
# Subtree

- There is a special data item called **root of the tree**. Remaining data items are partitioned into number of **mutually exclusive subsets**, each of which is itself a tree, called subtree.
- In a tree, each child from a node forms a **subtree** recursively.
- Every child node forms a subtree on its parent node.



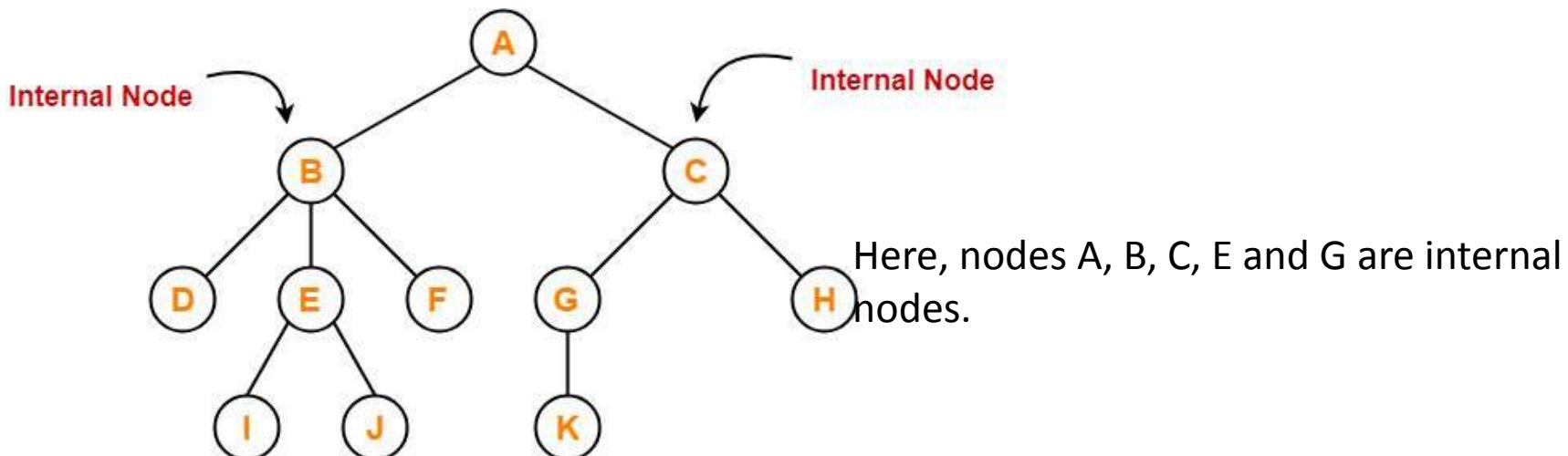
# Tree Terminology (Contd.)

- **Level** (or depth) of a node  $v$ : The length of the path from the root to  $v$ .  
Level of node A is 0 and node C is 1, So depth of k is 4
- **Height** of a node  $v$ : The length of the longest path from  $v$  to a leaf node.  
Height of node B is 2 and node E is 2 and so height of k is 0
- The height of a tree is the height of its root mode. So here it is 4
- By definition the height of an empty tree is -1.



# Internal node

- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node.



# Why Trees?

- ✓ Trees are very important data structures in computing.
- ✓ They are suitable for
  - Hierarchical structure representation, e.g.,
    - File directory.
    - Organizational structure of an institution.
    - Class inheritance tree.
  - Problem representation, e.g.,
    - Expression tree.
    - Decision tree.
  - Efficient algorithmic solutions, e.g.,
    - Search trees.
    - Efficient priority queues via heaps.

# General Trees and its Implementation

- In a general tree, there is no limit to the number of children that a node can have.
- Representing a general tree by linked lists:
- Each node has a linked list of the subtrees of that node.
- Each element of the linked list is a subtree of the current node

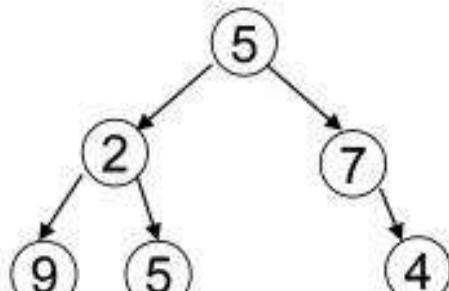
# Types of Trees

An N-ary tree is a tree that is either:

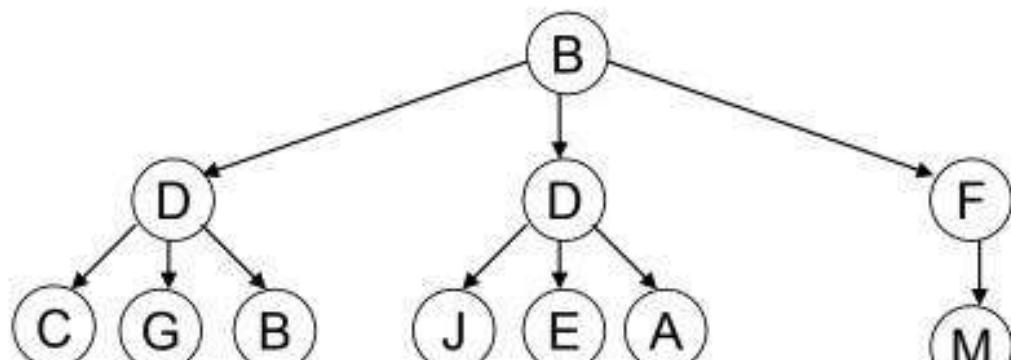
1. Empty, or
2. It consists of a root node and at most N non-empty N-ary subtrees.

It follows that the degree of each node in an N-ary tree is at most N.

Example of N-ary trees:



2-ary (binary) tree



3-ary (tertiary)tree

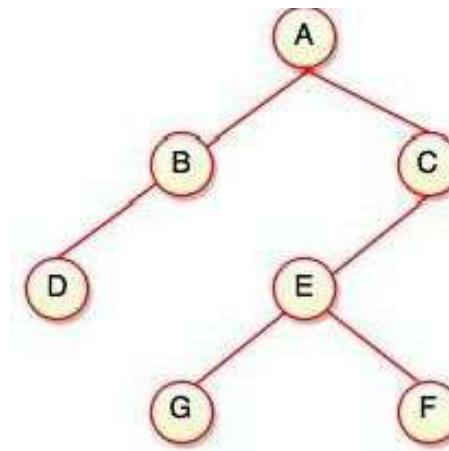


# Binary Tree

- Binary tree is a special type of data structure. In binary tree, every node can have a maximum of 2 children, which are known as **Left child** and **Right Child**.
- It is a method of placing and locating the records in a database, especially when all the data is known to be in random access memory (RAM)

# Binary Tree

- "A tree in which every node can have maximum of two children is called as Binary Tree."



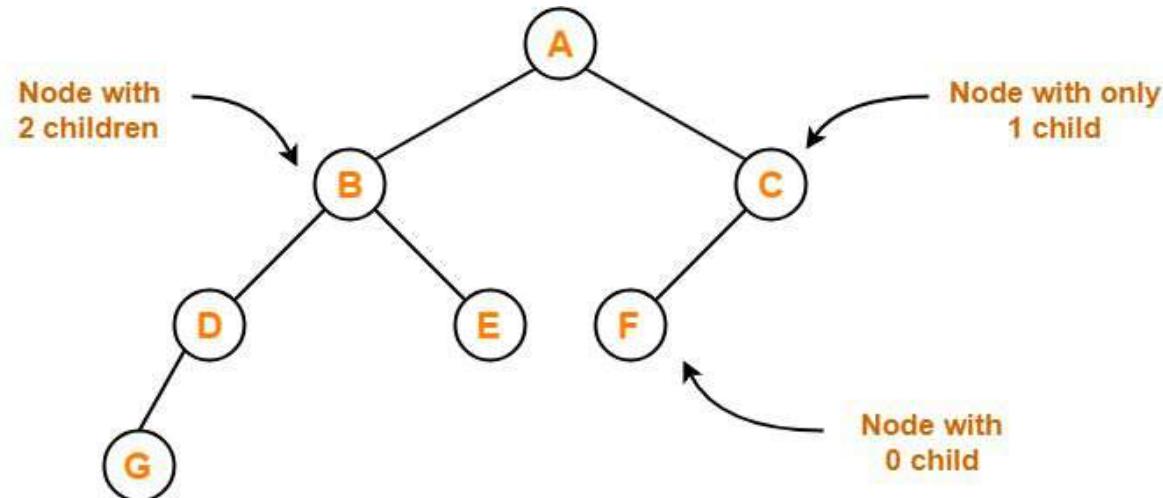
- The above tree represents binary tree in which node A has two children B and C. Each children have one child namely D and E respectively.

# Binary Tree

- A binary tree is a hierarchy of nodes, where every parent node has at most two child nodes. There is a unique node, called the root, that does not have a parent.
- A binary tree can be defined recursively as
- **Root node**
- **Left subtree:** left child and all its descendants
- **Right subtree:** right child and all its descendants

# Binary Tree

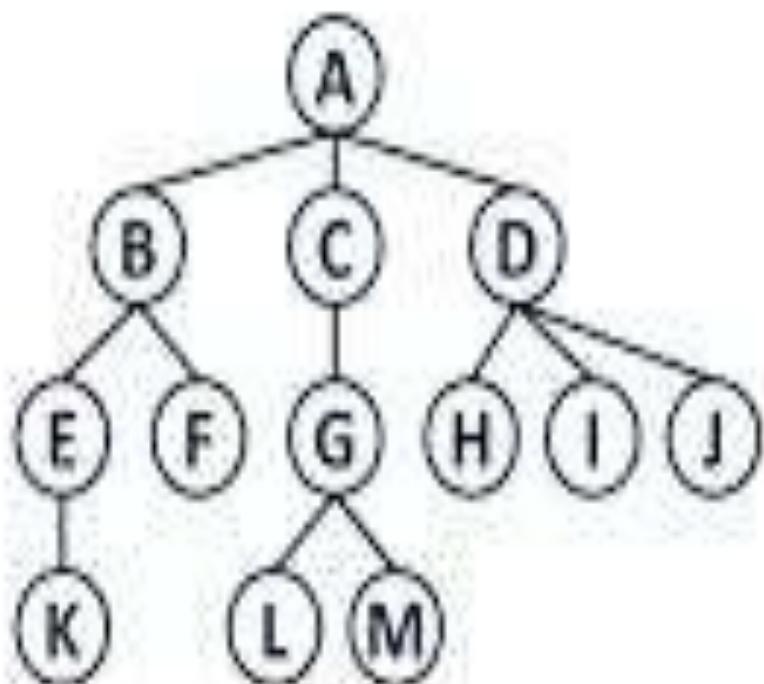
- Binary tree is a special tree data structure in which each node can have at most 2 children.
- Thus, in a binary tree, Each node has either 0 child or 1 child or 2 children.
  - It is either empty or
  - It consists a node called root
    - Left subtree
    - Right subtree



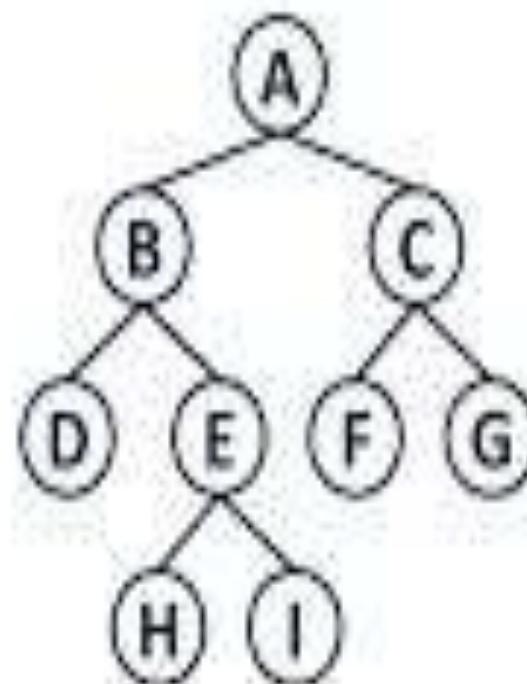
Binary Tree Example

# Binary Tree

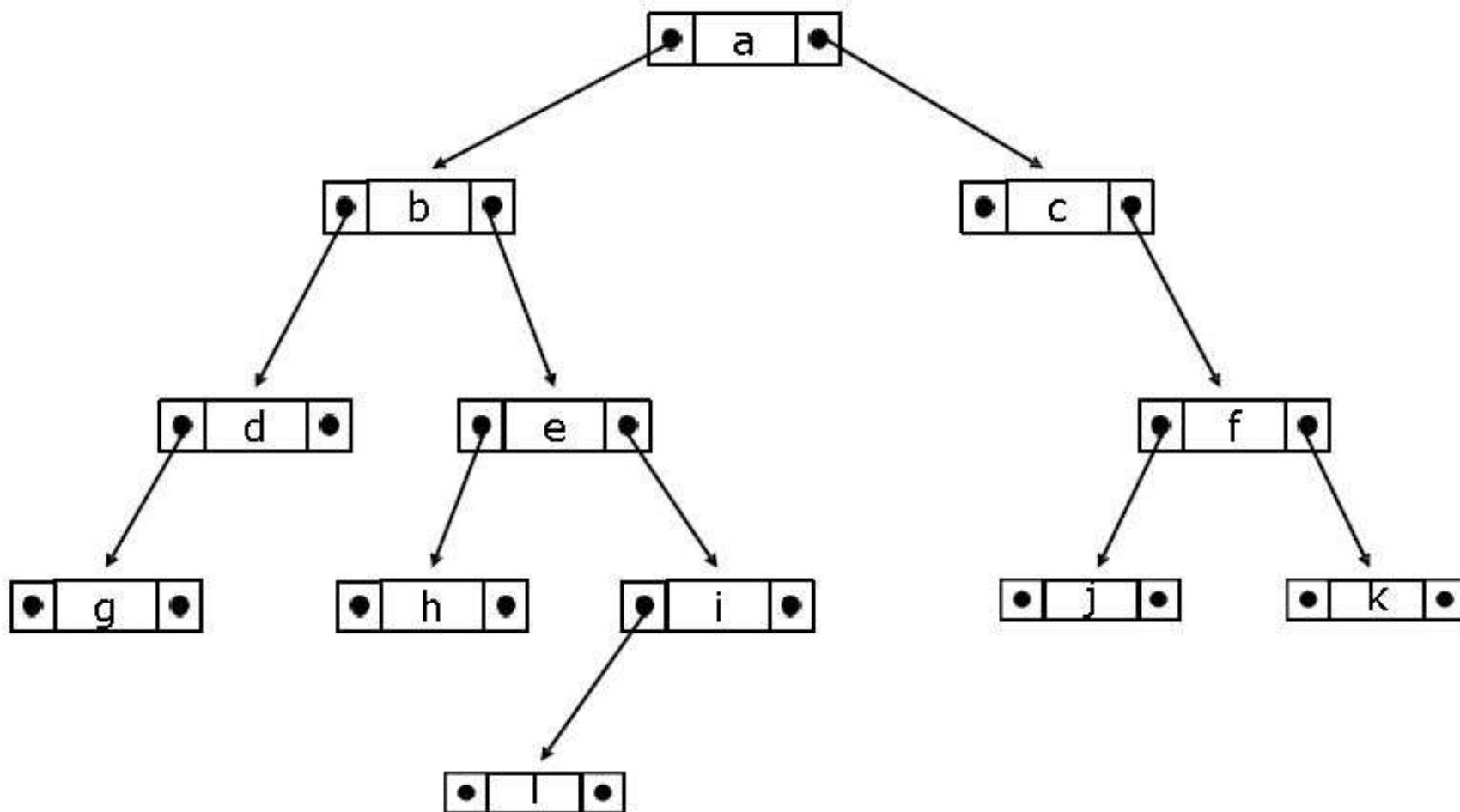
Tree



Binary Tree



# Binary Tree



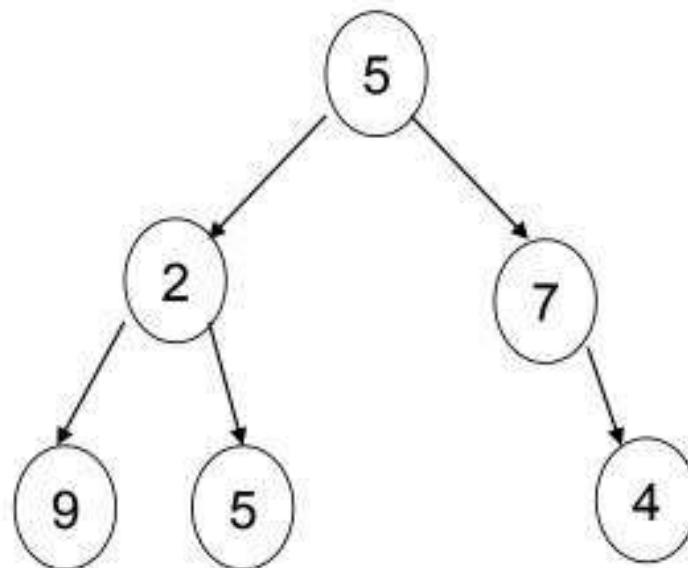
# Binary Trees

A binary tree is an N-ary tree for which  $N = 2$ .

Thus, a binary tree is either:

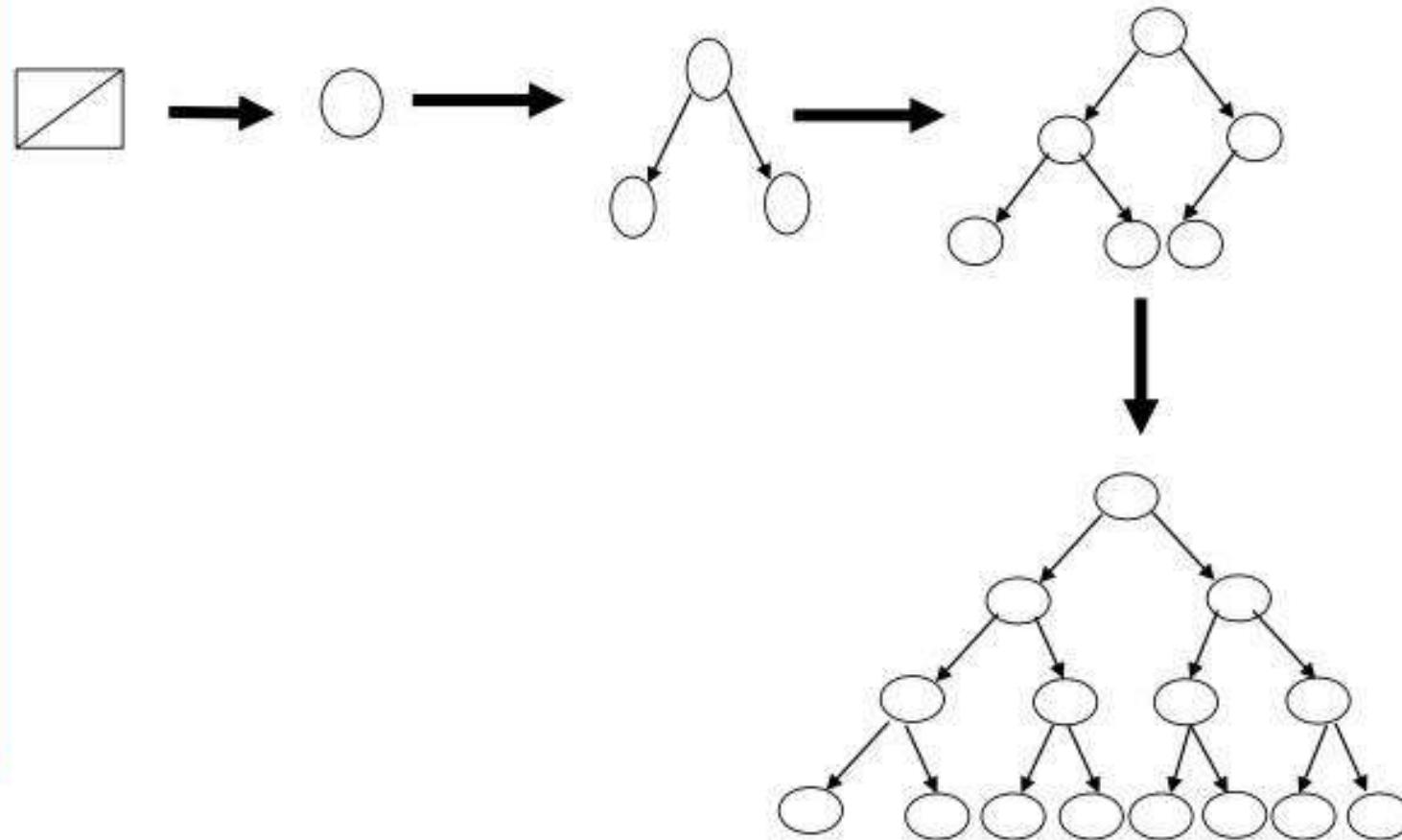
1. An empty tree, or
2. A tree consisting of a root node and at most two non-empty binary subtrees.

Example:



# Binary Trees (Contd.)

A **full binary** tree is either an empty binary tree or a binary tree in which each level  $k$ ,  $k \geq 0$ , has  $2^k$  nodes.



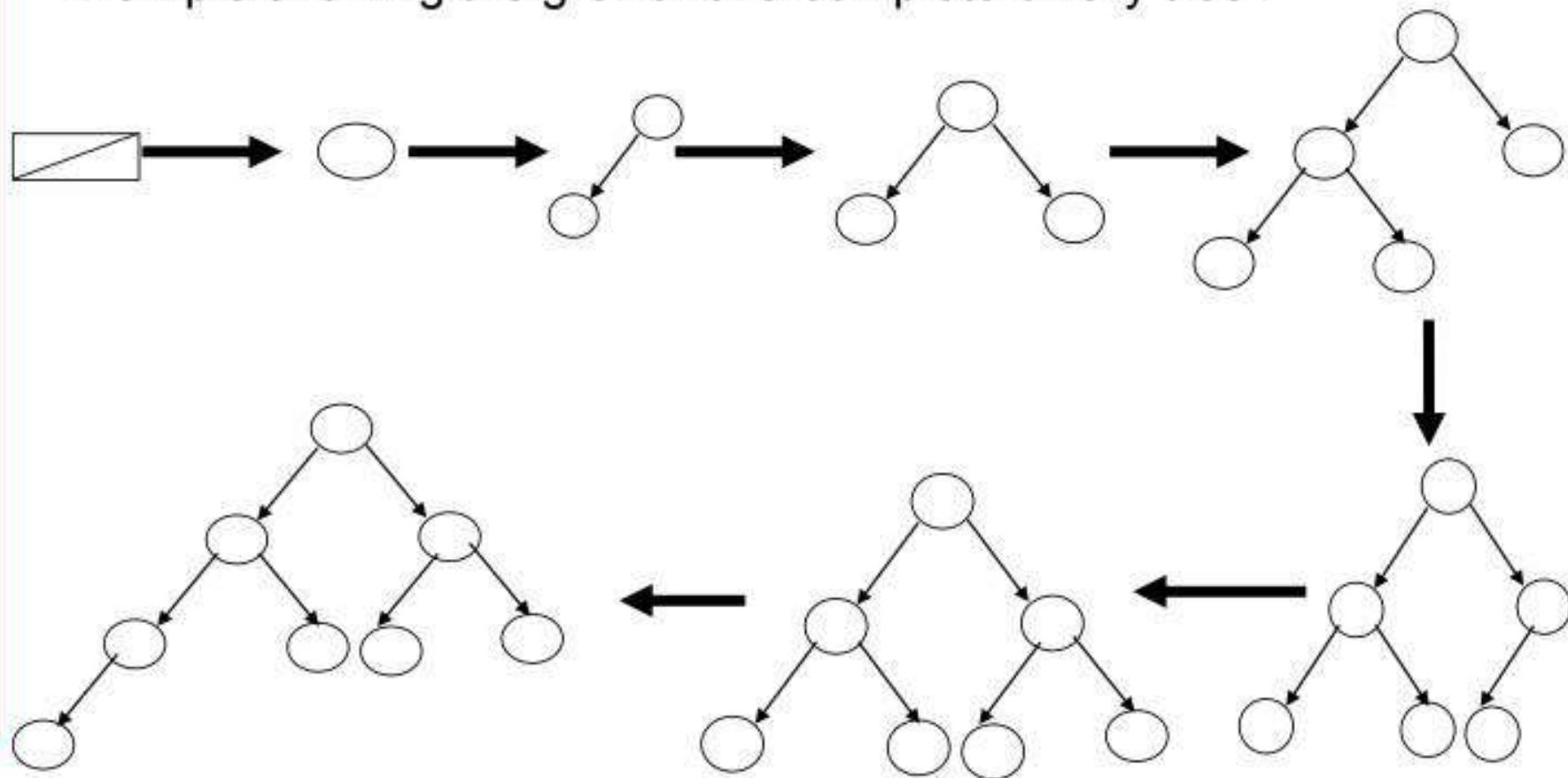
# Binary Trees - Types

- A *complete binary* tree is either an empty binary tree or a binary tree in which:
  1. Each level  $k$ ,  $k \geq 0$ , other than the last level contains the maximum number of nodes for that level, that is  $2^k$ .
  2. The last level may or may not contain the maximum number of nodes.
  3. If a slot with a missing node is encountered when scanning the last level in a left to right direction, then all remaining slots in the level must be empty.

*Thus, every full binary tree is a complete binary tree, but the opposite is not true.*

# Complete Binary Trees

Example showing the growth of a complete binary tree:

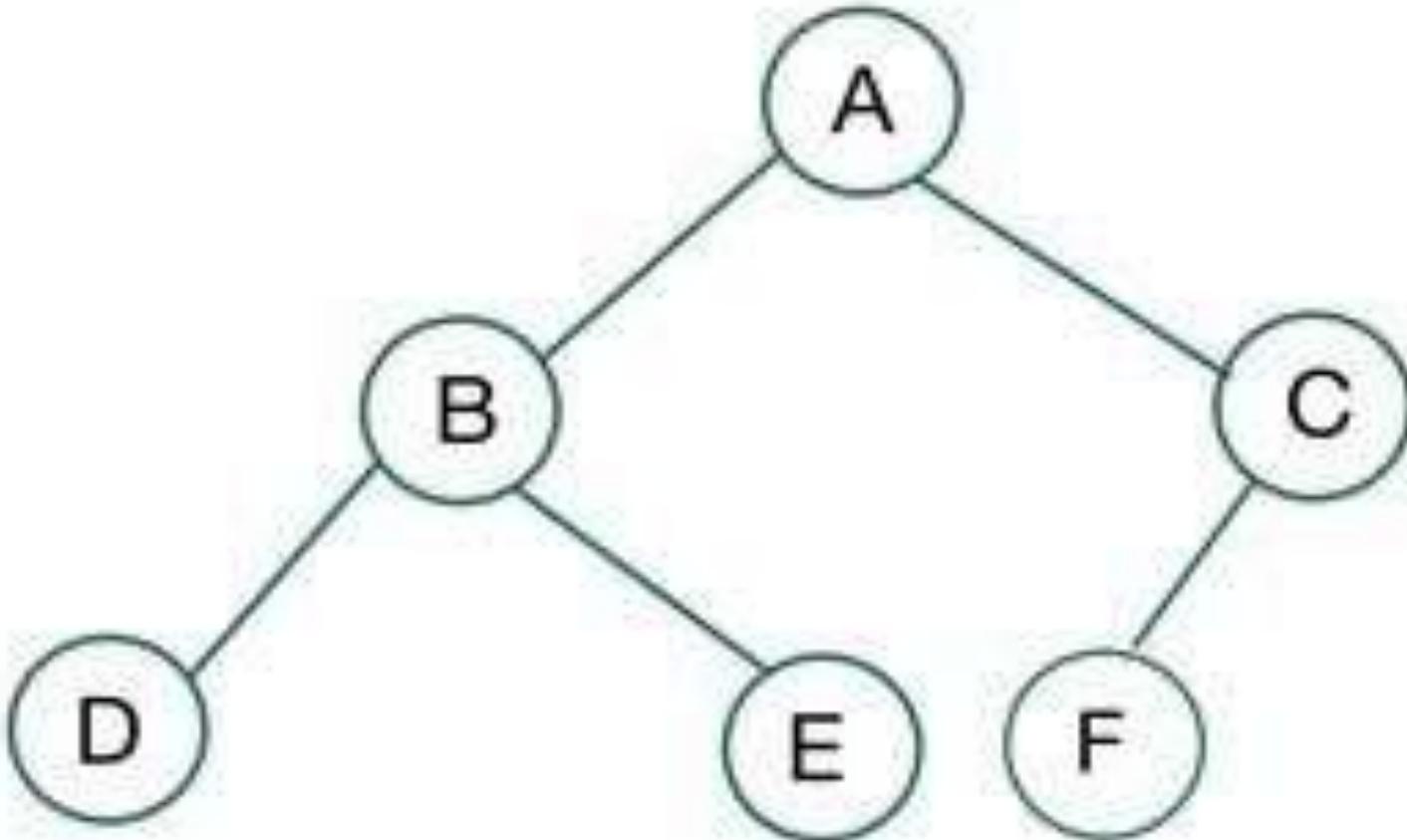


**SOMAIYA**

VIDYAVIHAR UNIVERSITY

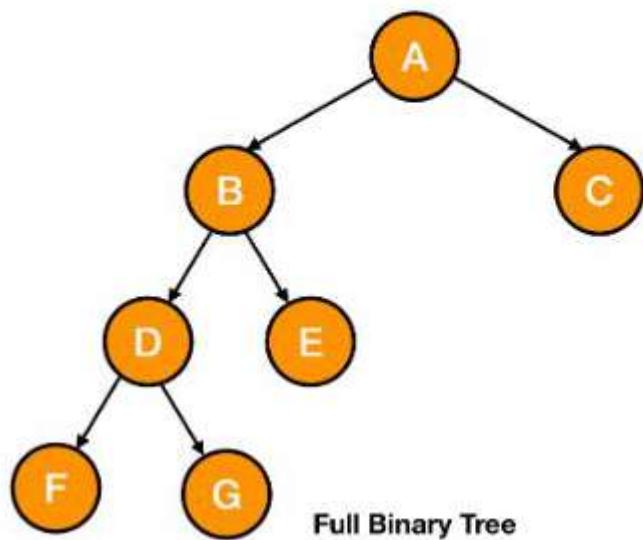
K J Somaiya College of Engineering

# COMPLETE BINARY TREE

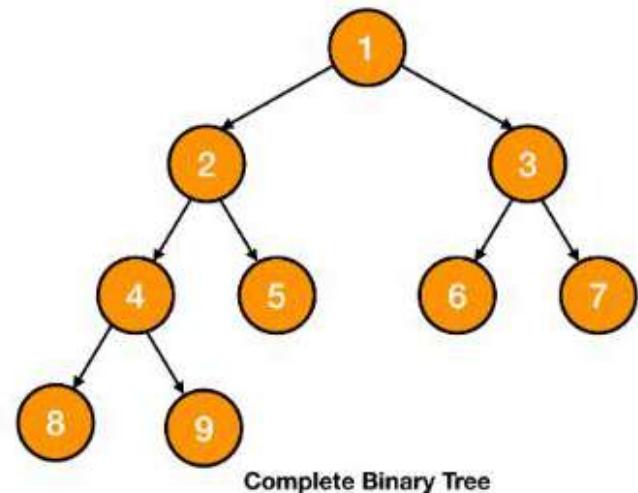


A binary tree is said to be a **complete binary tree** if all its levels, except possibly the last level, have the maximum number of possible nodes, and all the nodes in the last level appear as far left as possible.

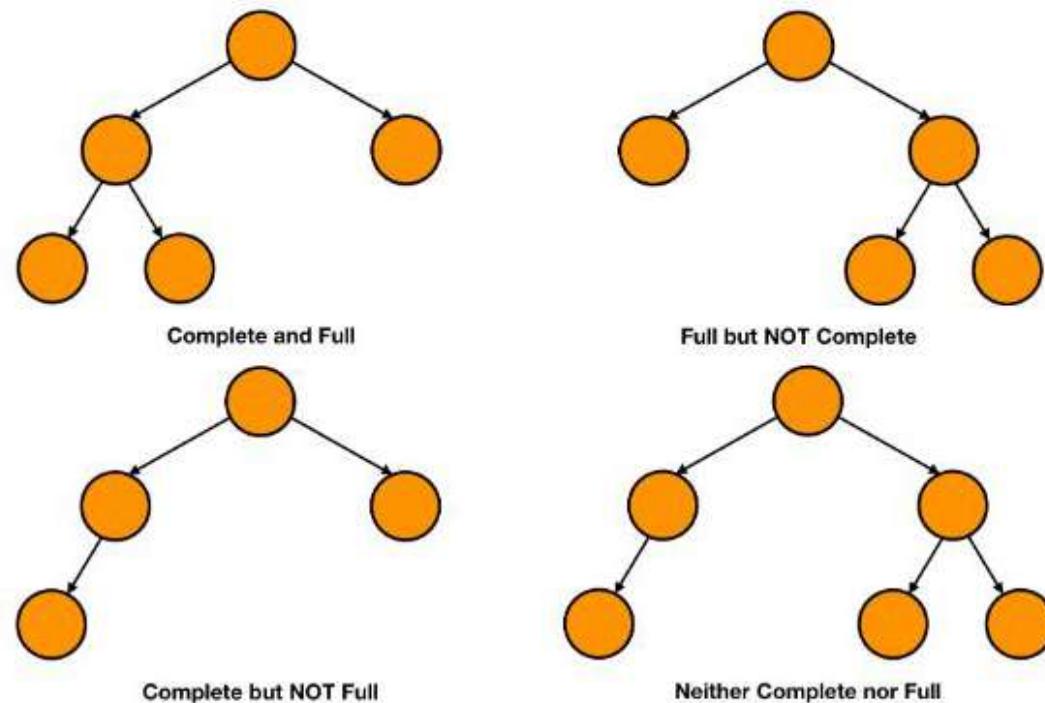
**Full Binary Tree** → A binary tree in which every node has 2 children except the leaves is known as a full binary tree.



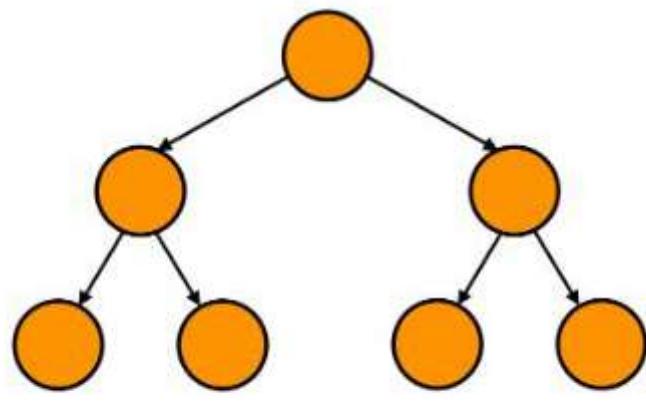
**Complete Binary Tree** → A binary tree which is completely filled with a possible exception at the bottom level i.e., the last level may not be completely filled and the bottom level is filled from left to right.



Let's look at this picture to understand the difference between a full and a complete binary tree.



**Perfect Binary Tree** → In a perfect binary tree, each leaf is at the same level and all the interior nodes have two children.



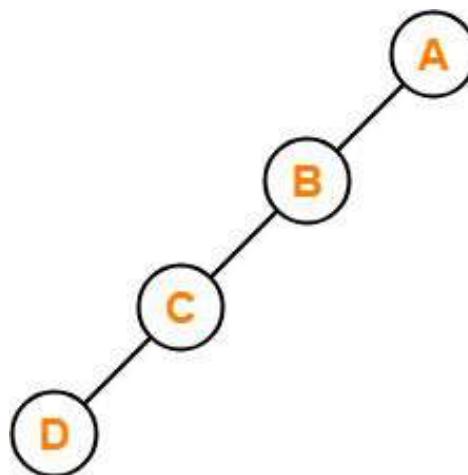
<https://www.codesdope.com/course/data-structures-binary-trees/>

# Skewed Binary Tree

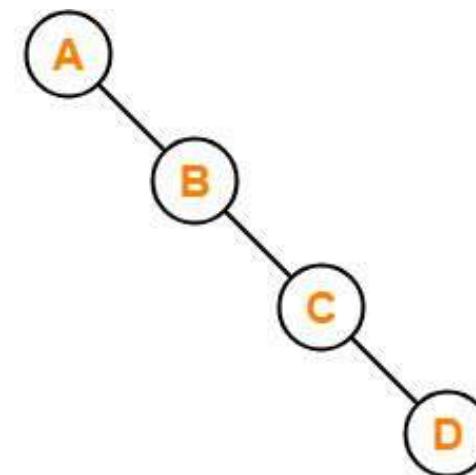
- All the nodes except one node has one and only one child.
- The remaining node has no child.

OR

- A **skewed binary tree** is a binary tree of  $n$  nodes such that its depth is  $(n-1)$ .



Left Skewed Binary Tree



Right Skewed Binary Tree

# Binary Trees Implementation

Using DLL-

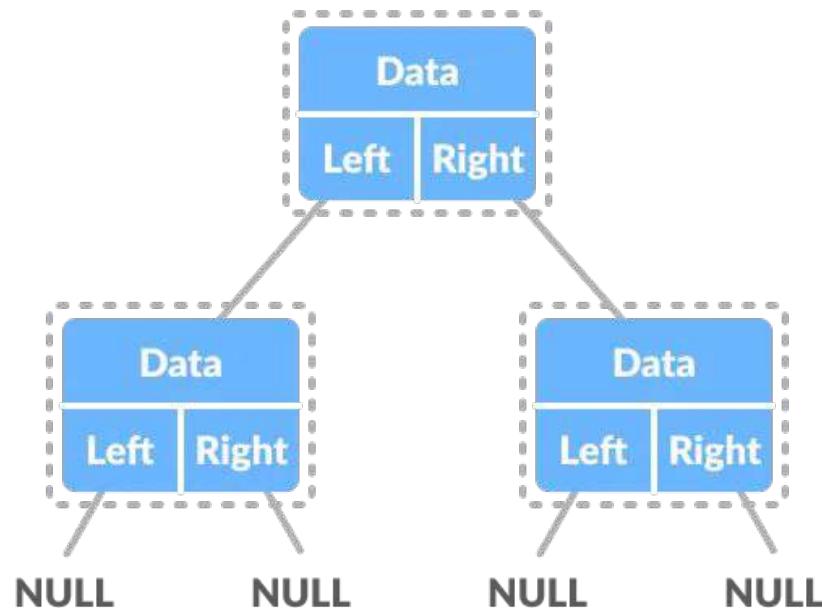
Struct BTnode

{

```
    int data;
    struct BTnode *left;
    struct BTnode *right;
```

}

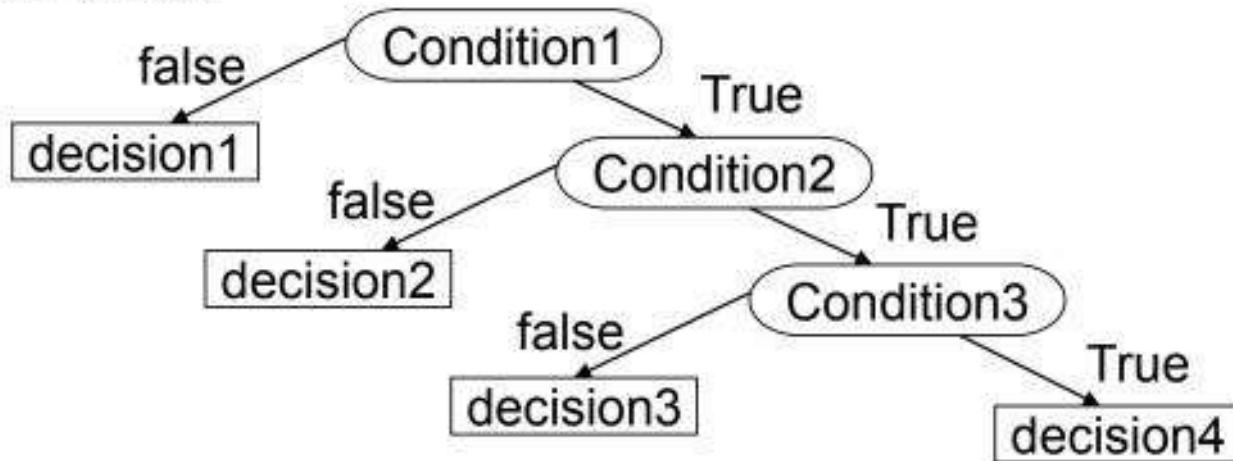
left	key	right
------	-----	-------



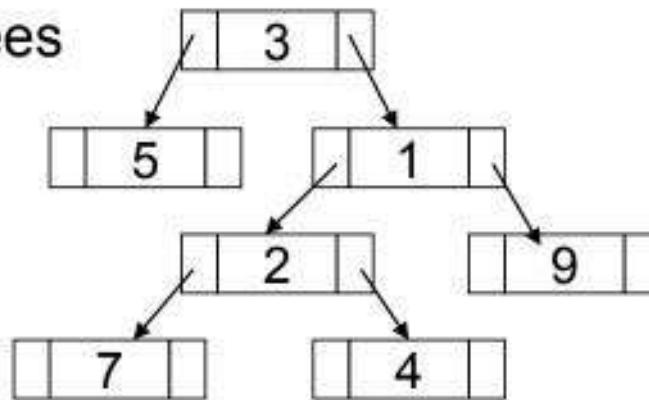
# Application of Binary Trees

Binary trees have many important uses. Two examples are:

1. Binary decision trees.



2. Binary Trees



# Binary Tree Representation

- Array Representation
- Linked List Representation

<https://www.prepbytes.com/blog/tree/array-representation-of-binary-tree/> (Array Representation: good theory explanation)

Codes:

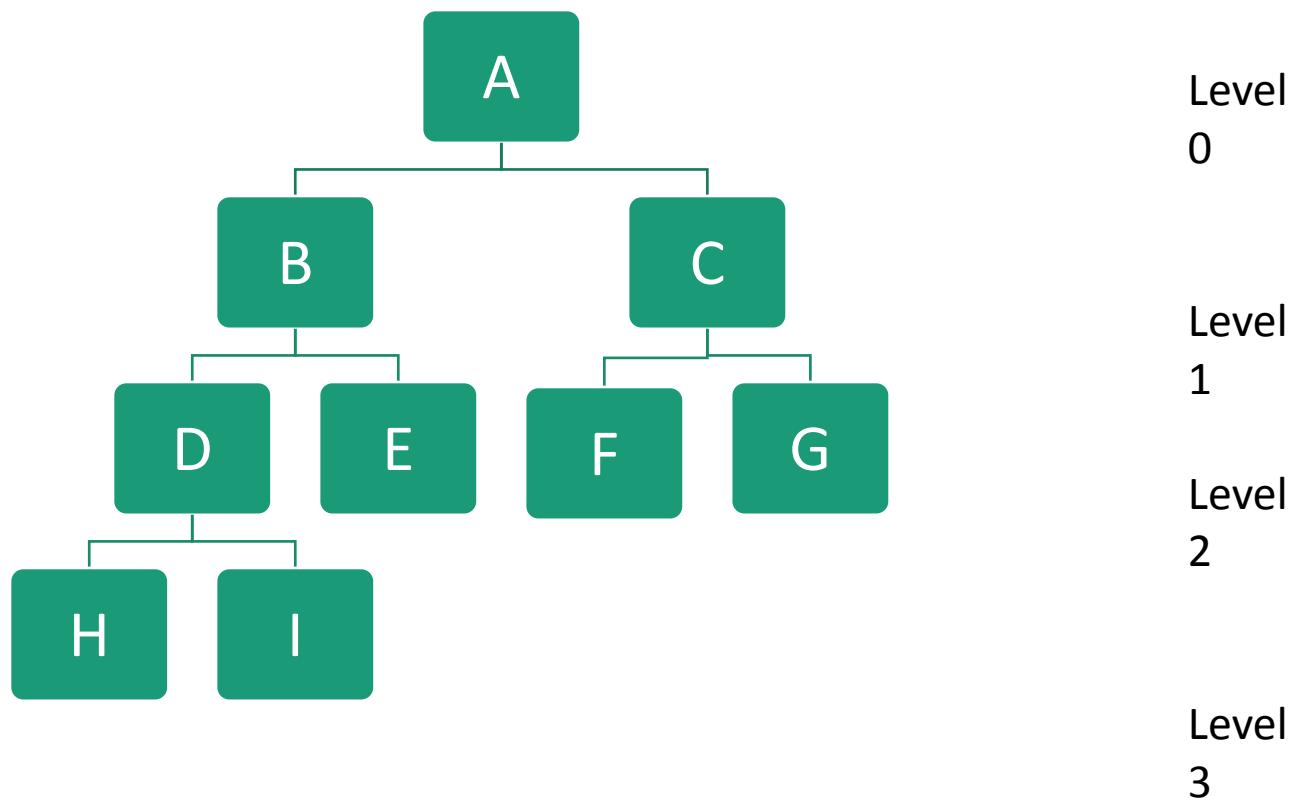
[https://docs.google.com/document/d/11SZBG-ZefpKbYaEW08SIX6cLv3xbxDeZGMCSoln\\_GaU/edit?usp=sharing](https://docs.google.com/document/d/11SZBG-ZefpKbYaEW08SIX6cLv3xbxDeZGMCSoln_GaU/edit?usp=sharing)

<https://www.scaler.com/topics/binary-tree-in-c/> (step by step code explanation)

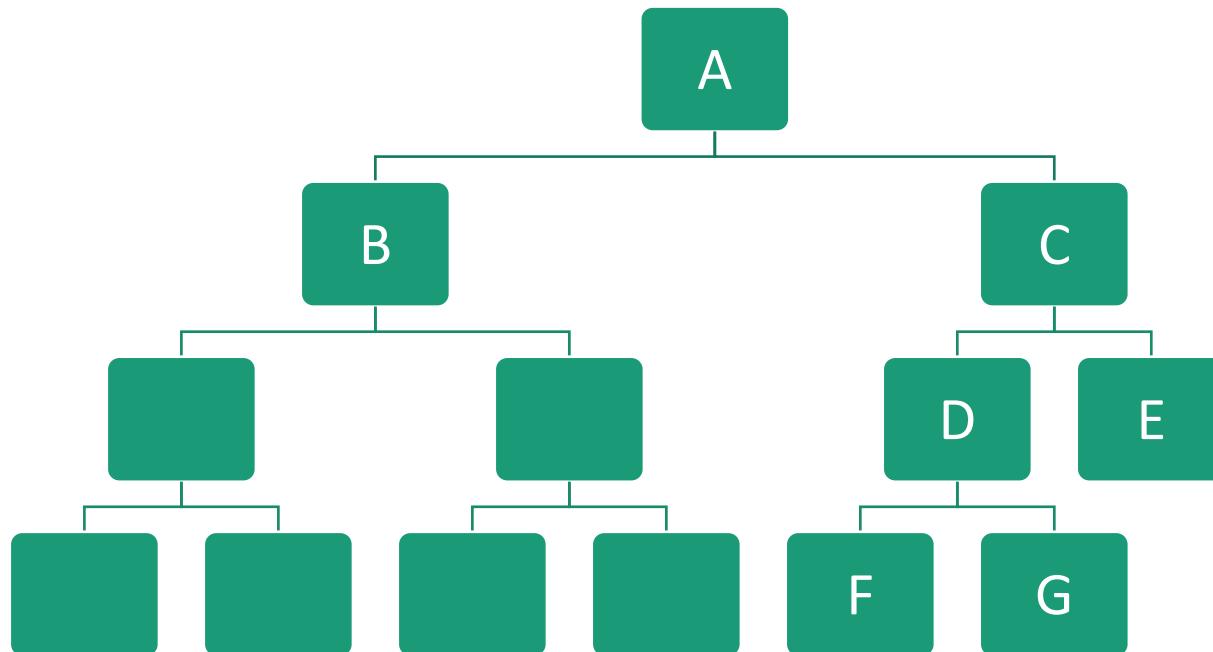
# Array Representation of a Binary Tree

- Using 1-D Array
- **Nodes are numbered sequentially level by level left to right.**
- **Even empty nodes are numbered**
- When data of the tree is stored in an array then the number appearing against the node will work as indices of the node in the array

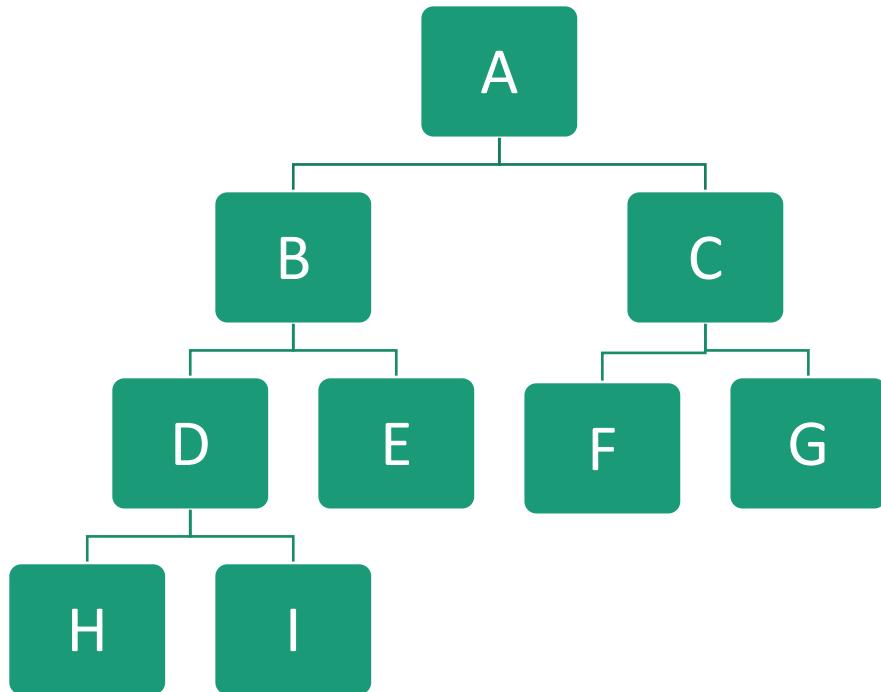
# Array Representation of a Binary Tree



# Array Representation of a Binary Tree



# Array Representation of a Binary Tree



Node in position p

Left child=2p+1

Right child=2p+2

For C= 2,

LC =  $2*2+1 = 5$

RC =  $2*2+2 = 6$

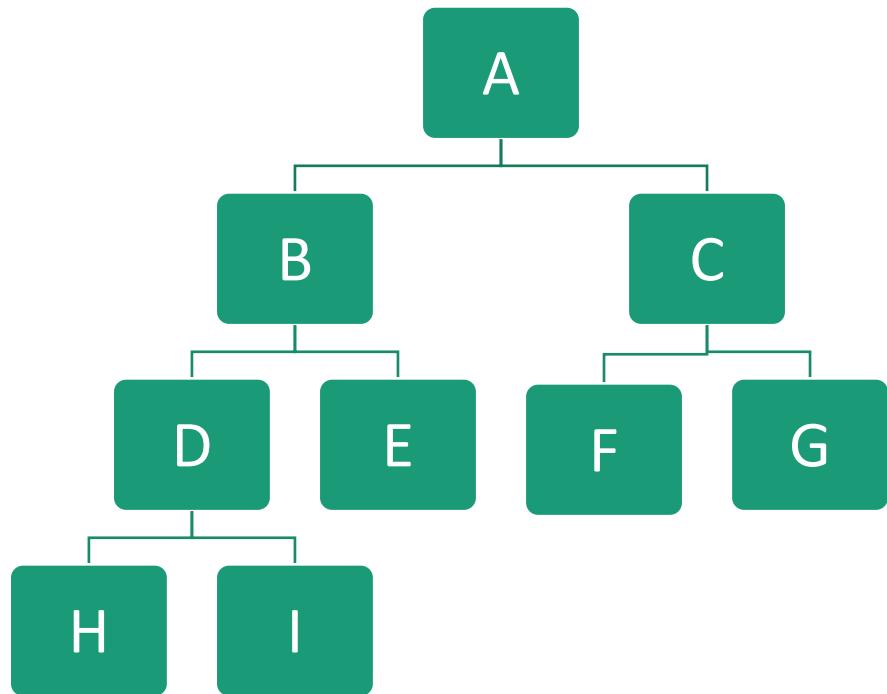
For D= 3,

LC =  $2*3+1 = 7$

RC =  $2*3+2 = 8$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
A	B	C	D	E	F	G	H	I

# Array Representation of a Binary Tree



Node in position  $p$  = sibling  
Given a Left child at position  $p$   
then

**Right Sibling =  $p+1$**

Given a Right child at position  $p$   
then

**Left Sibling =  $p-1$**

For  $C=I$ ,  
 $LS = 8-1 = 7$  i.e.  $H$

For  $D=$ ,  $P=3$ ,  
 $RS = 3+1 = 4$  i.e.  $E$

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
A	B	C	D	E	F	G	H	I

# Binary

## Tree

- **Representation of Binary Tree using Array:**

- Array index is a value in tree nodes and array value gives to the parent node of that particular index or node.
- Value of the root node index is always -1 as there is no parent for root.
- When the data item of the tree is sorted in an array, the number appearing against the node will work as indexes of the node in an array.

0	1	2	3	4	5	6	7
7	A	B	C	D	E	F	G

# Binary

## Tree

- **Representation of Binary Tree using Array:**

- Location number of an array is used to store the size of the tree.
- The first index of an array that is '0', stores the total number of nodes.
- All nodes are numbered from left to right level by level from top to bottom.
- In a tree, each node having an index  $i$  is put into the array as its  $i$  th element.

0	1	2	3	4	5	6	7
7	A	B	C	D	E	F	G

# Binary Tree

- **Representation of Binary Tree using Array:**

- The above figure shows how a binary tree is represented as an array.
- Value '7' is the total number of nodes. If any node does not have any of its child, null value is stored at the corresponding index of the array..

0	1	2	3	4	5	6	7
7	A	B	C	D	E	F	G

# Linked representation of Binary Tree

Linked representation of binary trees In the linked representation of a binary tree, every node will have three parts: the data element, a pointer to the left node, and a pointer to the right node.

So in C, the binary tree is built with a node type given below.

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

# Linked representation of Binary Tree

- Every binary tree has a pointer ROOT, which points to the root element (topmost element) of the tree. If ROOT = NULL, then the tree is empty.
- The left position is used to point to the left child of the node or to store the address of the left child of the node.
- The middle position is used to store the data.
- Finally, the right position is used to point to the right child of the node or to store the address of the right child of the node.
- Empty sub-trees are represented using X (meaning NULL).

# Linked representation of Binary Tree

- A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.

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

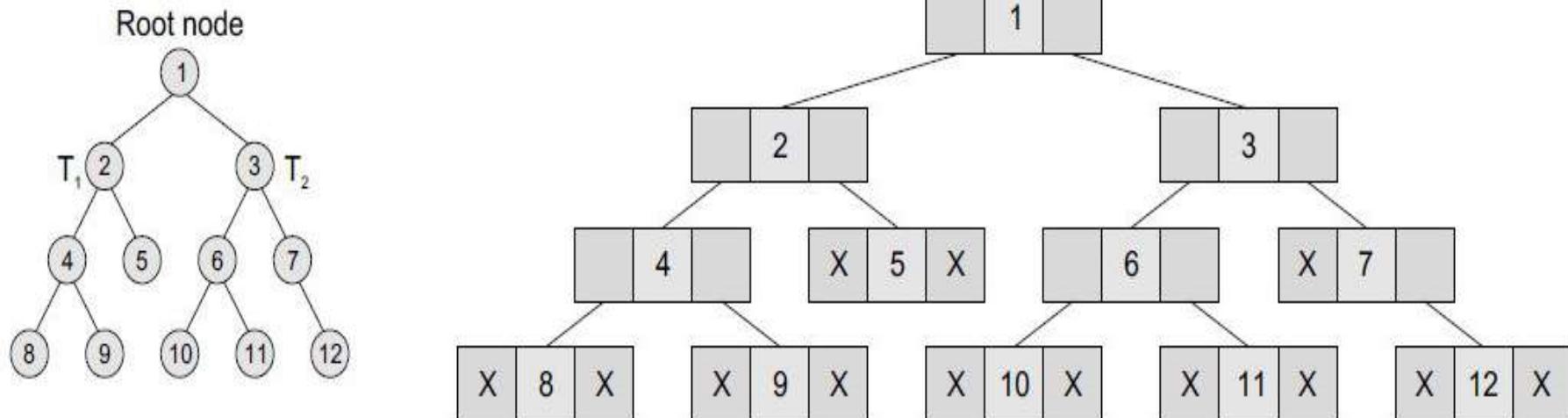
Tree has 3 fields in a node-  
Data field  
Address of left child  
Address of right child

# Linked representation of Binary Tree

- Every binary tree has a pointer ROOT, which points to the root element (topmost element) of the tree.
- If ROOT = NULL, then the tree is empty.

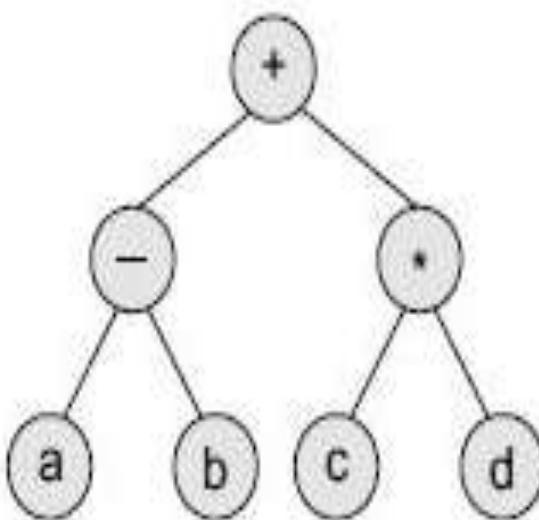
# Linked representation of Binary Tree

- Consider the binary tree given and the schematic diagram of the linked representation of the binary tree shown in Fig.



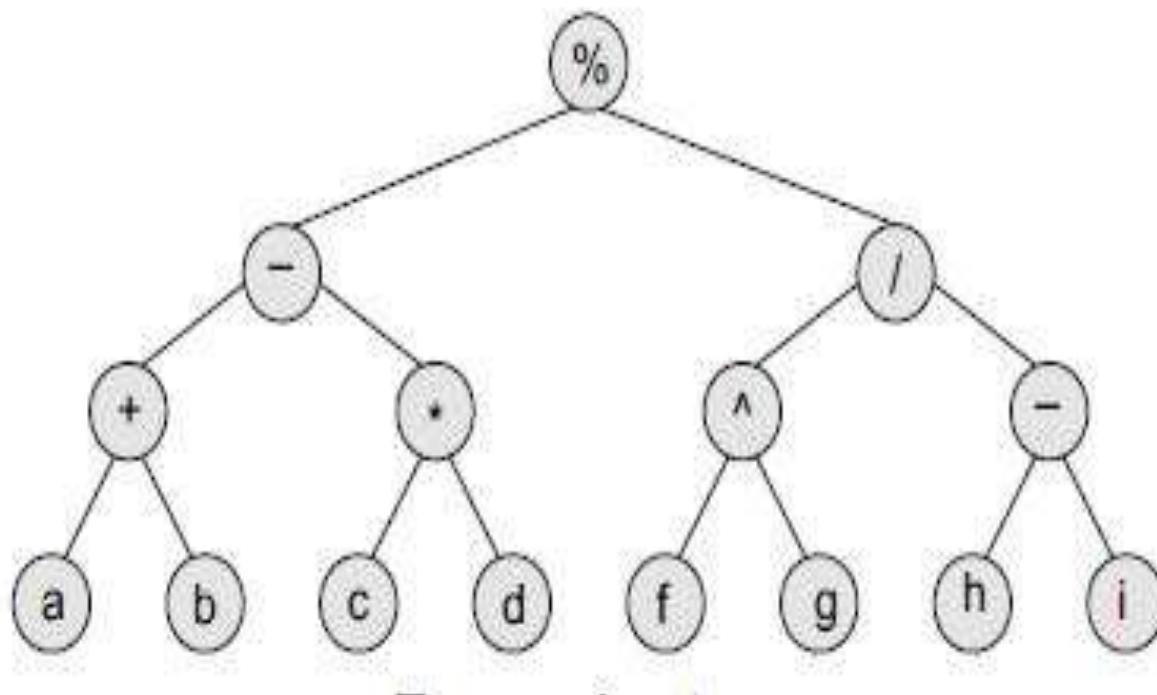
# Binary Tree

- **Expression Trees**
- Binary trees are widely used to store algebraic expressions.
- For example, consider the algebraic expression given as:
- $\text{Exp} = (a - b) + (c * d)$
- This expression can be represented using a binary tree as shown in Fig



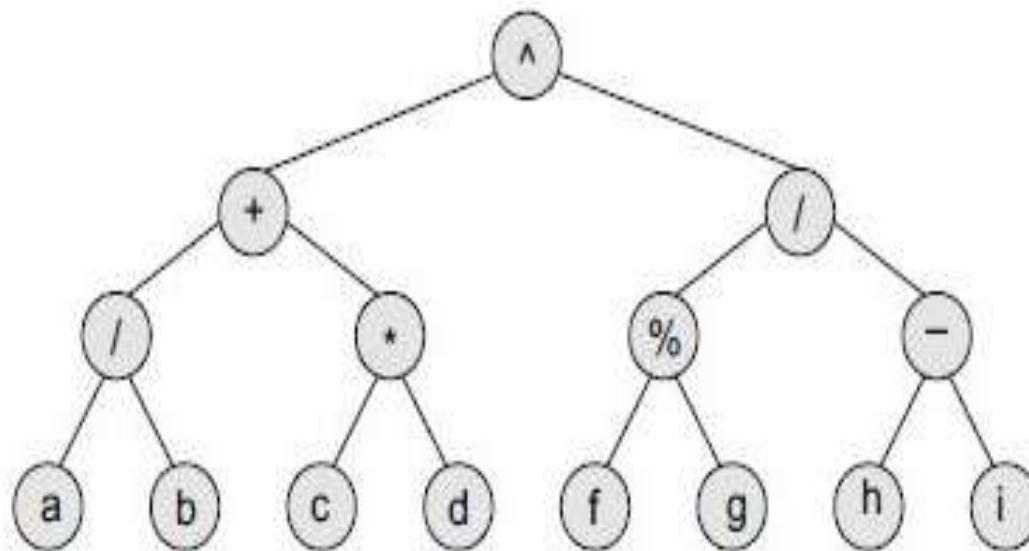
# Binary Tree

- Given an expression,  $Exp = ((a + b) - (c * d)) \% ((e ^ f) / (g - h))$ , construct the corresponding binary tree.



# Binary Tree

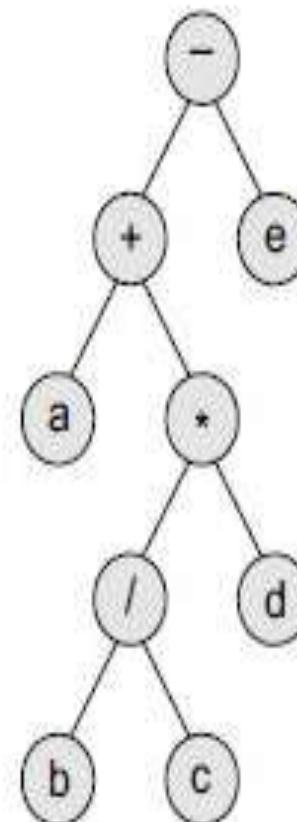
- Given the binary tree, write down the expression that it represents.



Expression for the above binary tree is  
$$[(a/b) + (c*d)] ^ [(f \% g)/(h - i)]$$

# Binary Tree

- Given the expression,  $Exp = a + b / c * d - e$ , construct the corresponding binary tree.
- Use the operator precedence to find the sequence in which operations will be performed. The given expression can be written as
- $Exp = ((a + ((b/c) * d)) - e)$



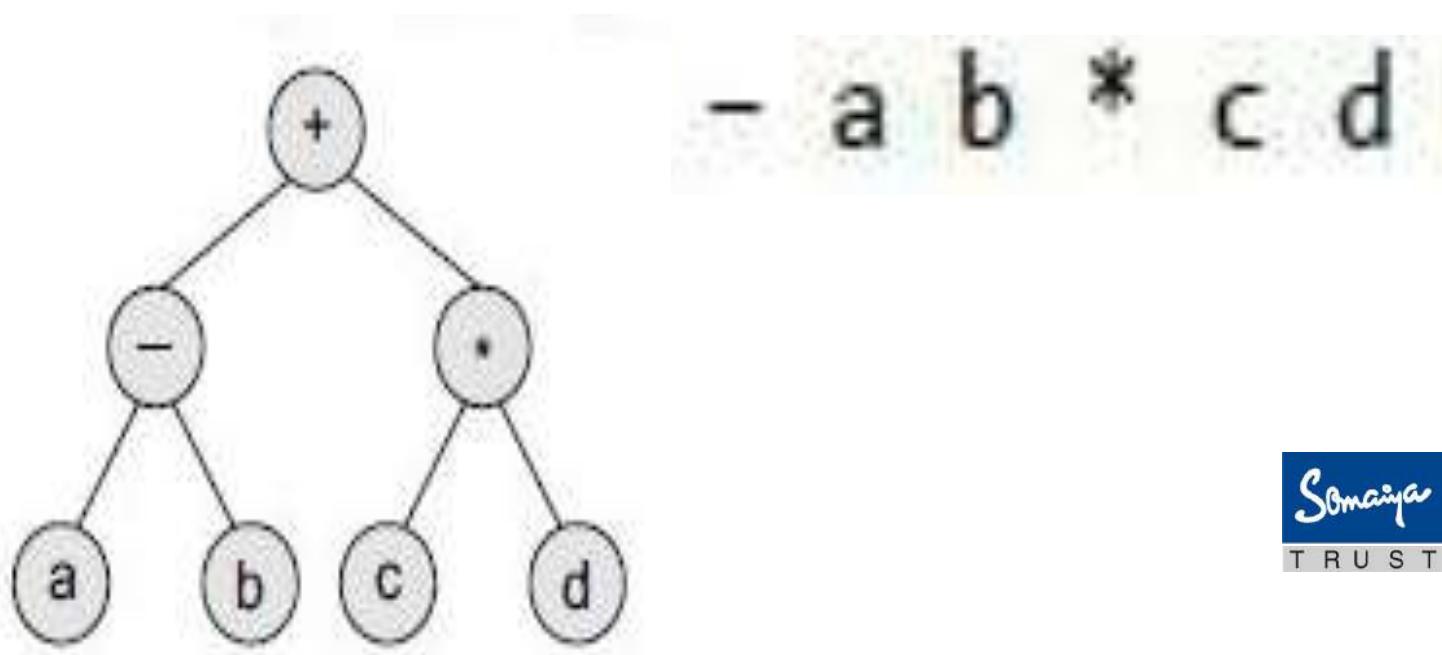
# Preorder Traversal

Pre-order traversal algorithms are used to extract a prefix notation from an expression tree.

For example, consider the expression given  $\text{Exp} = (a - b) + (c * d)$

This expression can be represented using a binary tree as

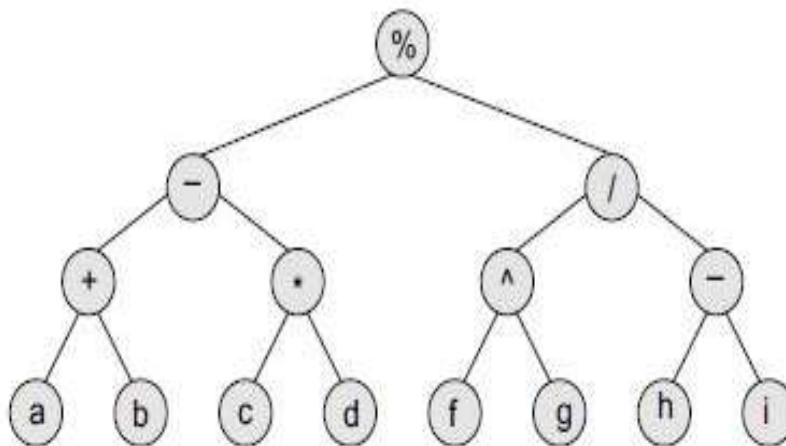
When we traverse the elements of a tree using the pre-order traversal algorithm, the expression that we get is a prefix expression.



# Preorder Traversal

Given an expression,  $\text{Exp} = ((a + b) - (c * d)) \% ((f ^ g) / (h - i))$ , construct the corresponding binary tree.

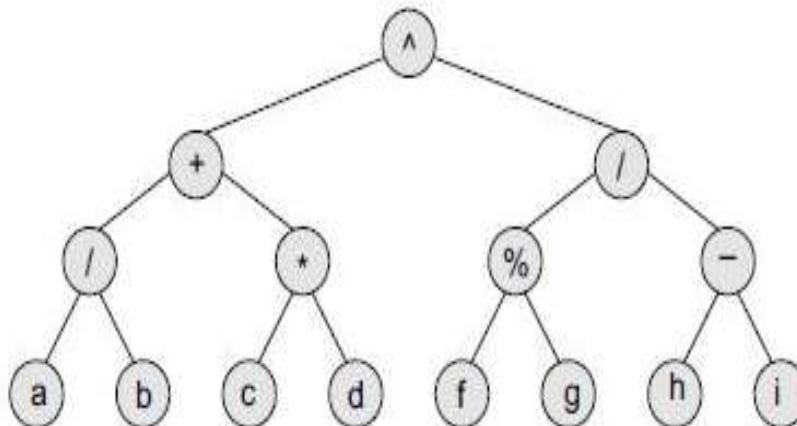
This expression can be represented using a binary tree as



When we traverse the elements of a tree using the pre-order traversal algorithm, the expression that we get is a prefix expression i.e.,  $\% - + a b * c d / ^ f g - h i$

# Preorder Traversal

Given the binary tree, write down the expression that it represents.



Expression for the above binary tree is  $[(a/b) + (c*d)] ^ [(f \% g)/(h - i)]$

When we traverse the elements of a tree using the pre-order traversal algorithm, the expression that we get is a prefix expression  $^ + / a b * c d \% f g - h i$

# Applications

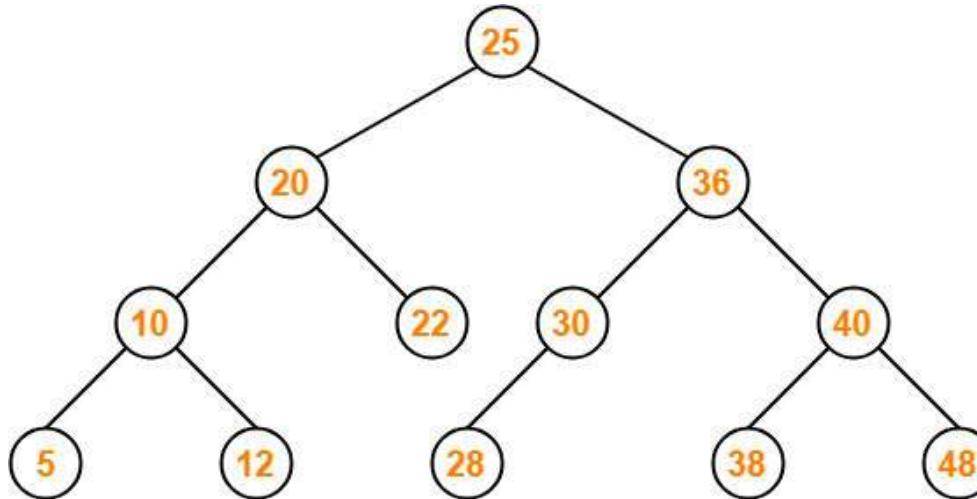
- Preorder traversal is used to get prefix expression of an expression tree.
- Preorder traversal is used to create a copy of the tree.
- Inorder traversal is used to get infix expression of an expression tree.
- Postorder traversal is used to get postfix expression of an expression tree.
- Postorder traversal is used to delete the tree. This is because it deletes the children first and then it deletes the parent.

# Binary Search Trees (BST)

- **What is a Binary search tree?**
- **Why Binary search trees?**
- **Binary search tree implementation**
- **Insertion in a BST**
- **Deletion from a BST**

# Binary Search Tree(BST)

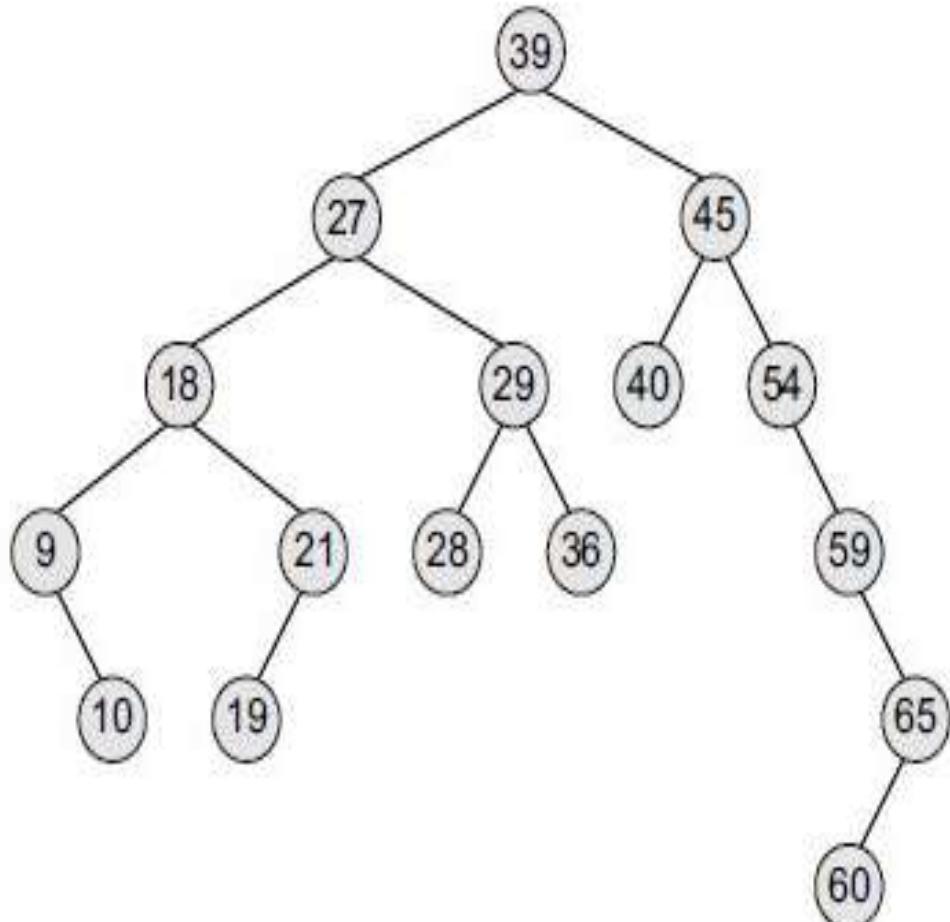
- Binary Search Tree is a special kind of binary tree in which nodes are arranged in a specific order.
- In a binary search tree (BST), each node contains-
- Only smaller values in its left sub tree
- Only larger values in its right sub tree



Binary Search Tree

# Binary Search Tree(BST)

- For example, in the given tree, if we have to search for 29.
- Then we know that we have to scan only the left sub-tree. If the value is present in the tree, it will only be in the left sub-tree, as 29 is smaller than 39 (the root node's value).
- The left sub-tree has a root node with the value 27. Since 29 is greater than 27, we will move to the right sub-tree, where we will find the element.



# Binary Search Trees

- In a BST, each node stores some information including a unique **key value**, and perhaps some associated data. A binary tree is a BST iff, for every node  $n$  in the tree:
  - All keys in  $n$ 's left subtree are less than the key in  $n$ , and
  - All keys in  $n$ 's right subtree are greater than the key in  $n$ .
- In other words, binary search trees are binary trees in which all values in the node's left subtree are less than node value all values in the node's right subtree are greater than node value.

# Properties and Operations

A BST is a binary tree of nodes ordered in the following way:

- i. Each node contains one key (also unique)
- ii. The keys in the left subtree are  $<$  (less) than the key in its parent node
- iii. The keys in the right subtree  $>$  (greater) than the key in its parent node
- iv. Duplicate node keys are not allowed.

# Inserting a node

- A naïve algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root.
- We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node. We note that a new node is always inserted as a leaf node.

# Inserting a node

- A recursive algorithm for inserting a node into a BST is as follows. Assume we insert a node  $N$  to tree  $T$ . if the tree is empty, the we return new node  $N$  as the tree. Otherwise, the problem of inserting is reduced to inserting the node  $N$  to left of right sub trees of  $T$ , depending on  $N$  is less or greater than  $T$ . A definition is as follows.

$\text{Insert}(N, T) = N \text{ if } T \text{ is empty}$   
 $= \text{insert}(N, T.\text{left}) \text{ if } N < T$   
 $= \text{insert}(N, T.\text{right}) \text{ if } N > T$

# Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

## Algorithm

1. START
2. Check whether the tree is empty or not
3. If the tree is empty, search is not possible
4. Otherwise, first search the root of the tree.
5. If the key does not match with the value in the root, search its sub
6. If the value of the key is less than the root value, search the left
7. If the value of the key is greater than the root value, search the r
8. If the key is not found in the tree, return unsuccessful search.
9. END

# Searching for a node

- Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node). A recursive definition of search is as follows. If the node is equal to root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for  $T.\text{left}$  or  $T.\text{right}$ , depending on  $N < T$  or  $N > T$ . A recursive definition is as follows.
- Search should return a true or false, depending on the node is found or not.

# Searching for a node

- $\text{Search}(N, T) = \text{false}$  if  $T$  is empty. Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node).
- A recursive definition of search is as follows. If the node is equal to root, then we return true. If the root is null, then we return false. Otherwise we recursively solve the problem for  $T.\text{left}$  or  $T.\text{right}$ , depending on  $N < T$  or  $N > T$ . A recursive definition is as follows.
- Search should return a true or false, depending on the node is found or not.

# Insert Operation

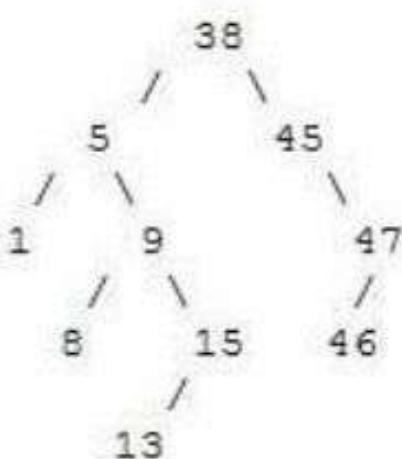
Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

## Algorithm

```
1 - START
2 - If the tree is empty, insert the first element as the root node of
3 - If an element is less than the root value, it is added into the left
4 - If an element is greater than the root value, it is added into the right
5 - The final leaf nodes of the tree point to NULL values as their children
6 - END
```

# Deleting a node

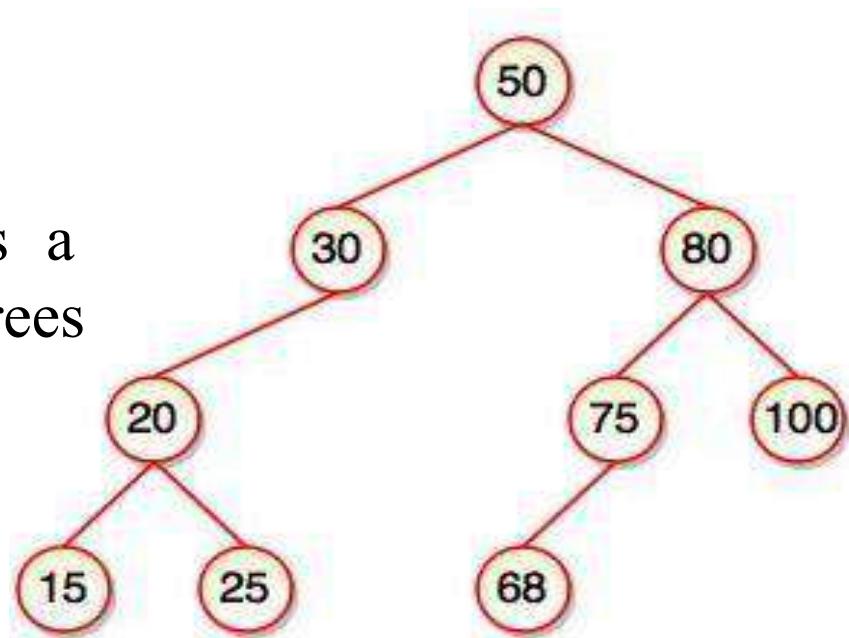
- A BST is a connected structure. That is, all nodes in a tree are connected to some other node. For example, each node has a parent, unless node is the root. Therefore deleting a node could affect all sub trees of that node. For example, deleting node 5 from the tree could result in losing sub trees that are rooted at 1 and 9.



# Binary Search Tree

- "Binary Search Tree is a binary tree where each node contains only smaller values in its left subtree and only larger values in its right subtree."

**Note:** Every binary search tree is a binary tree, but all the binary trees need not to be binary search trees.



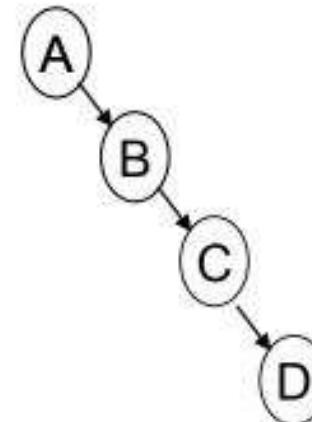
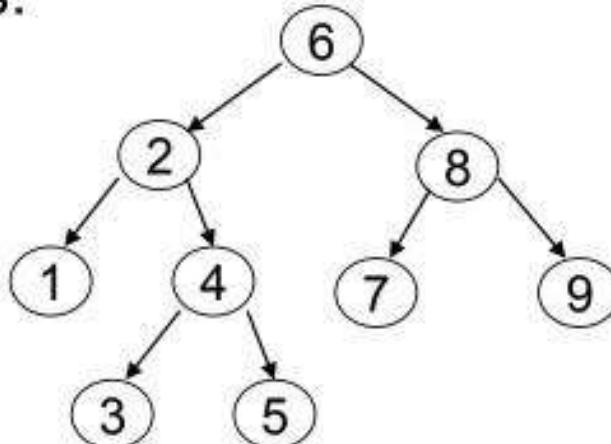
# Binary Search Trees (Definition)

A binary search tree (BST) is a binary tree that is empty or that satisfies the BST ordering property:  $LC < P < RC$

1. The key of each node is greater than each key in the left subtree, if any, of the node.
2. The key of each node is less than each key in the right subtree, if any, of the node.

Thus, *each key in a BST is unique*.

Examples:



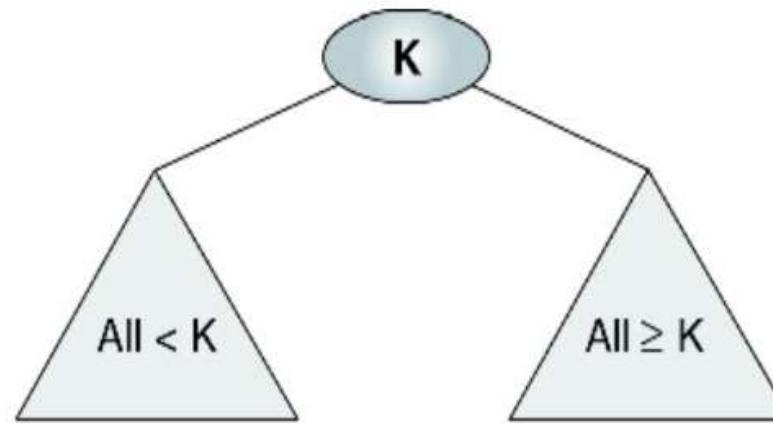
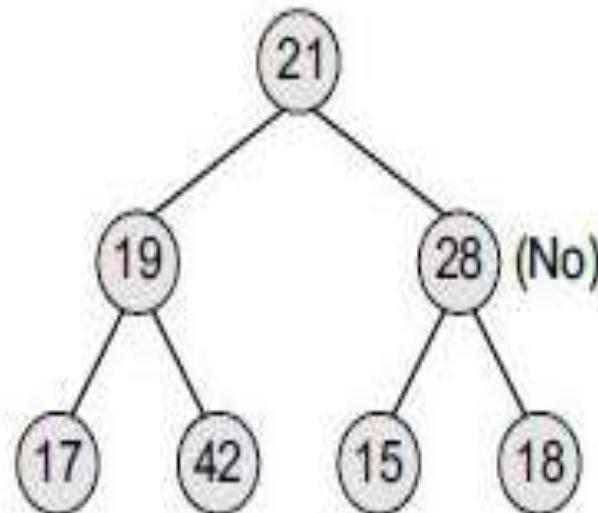
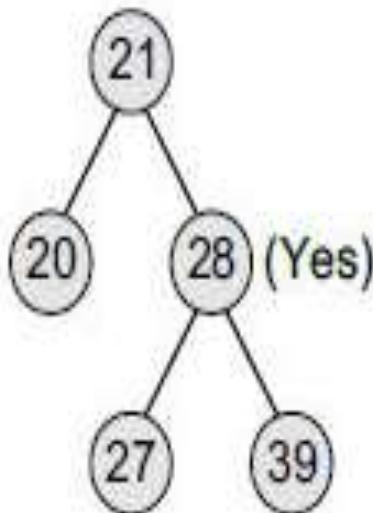
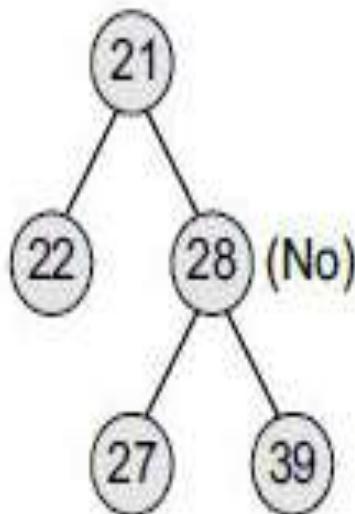


FIGURE 7-1 Binary Search Tree

---

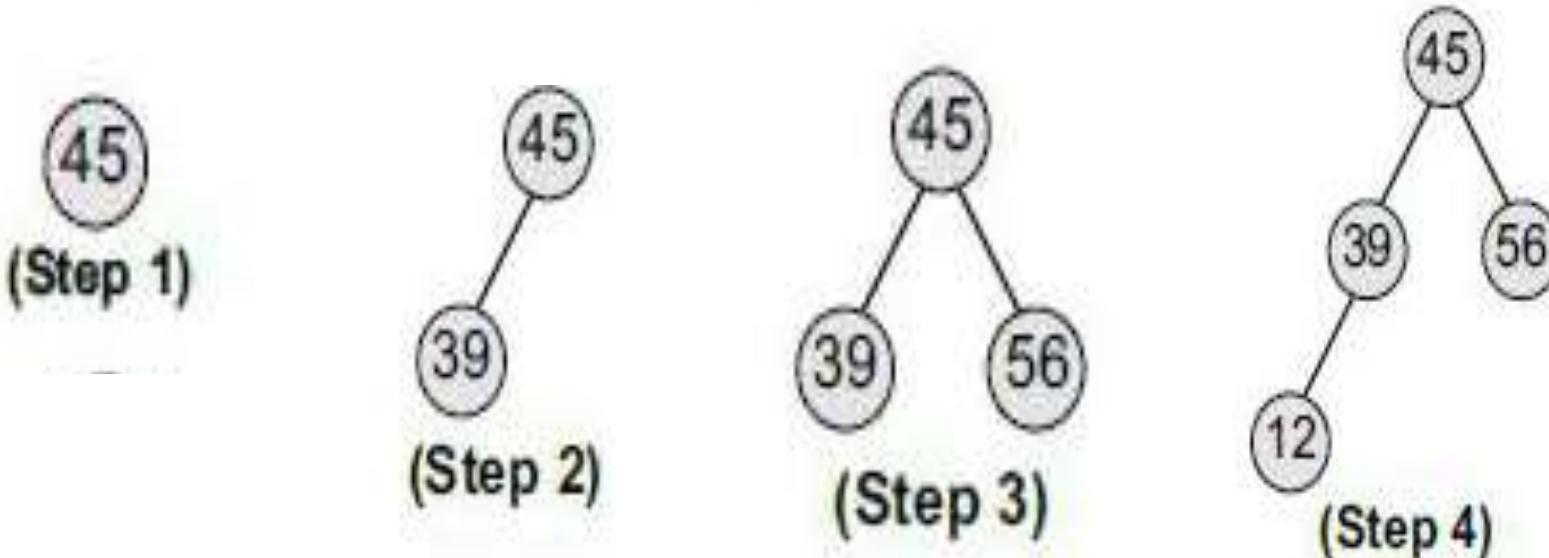
# Binary Search Tree(BST)

State whether the binary trees in Fig. are binary search trees or not.



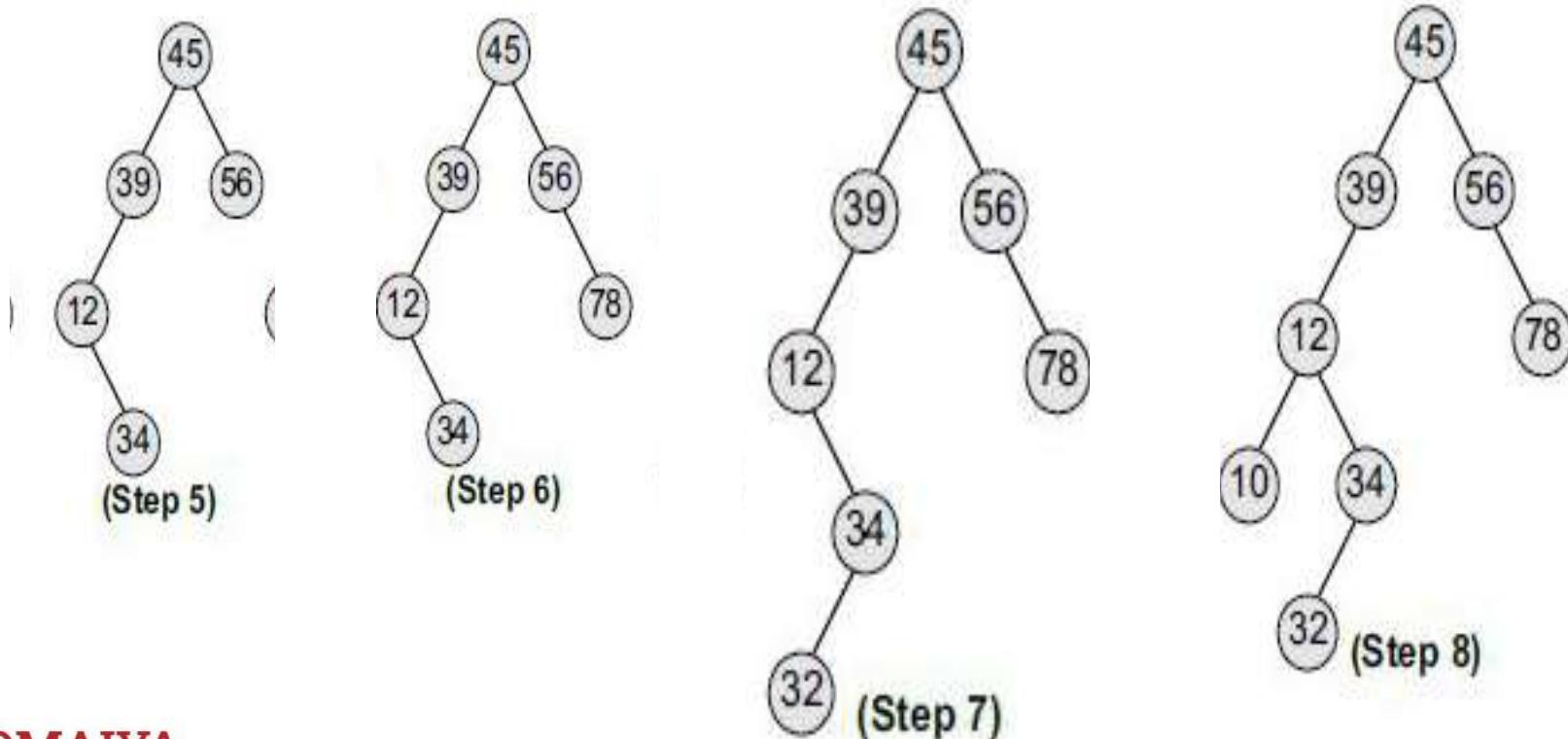
# Binary Search Tree

- Create a binary search tree using the following data elements:  
45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



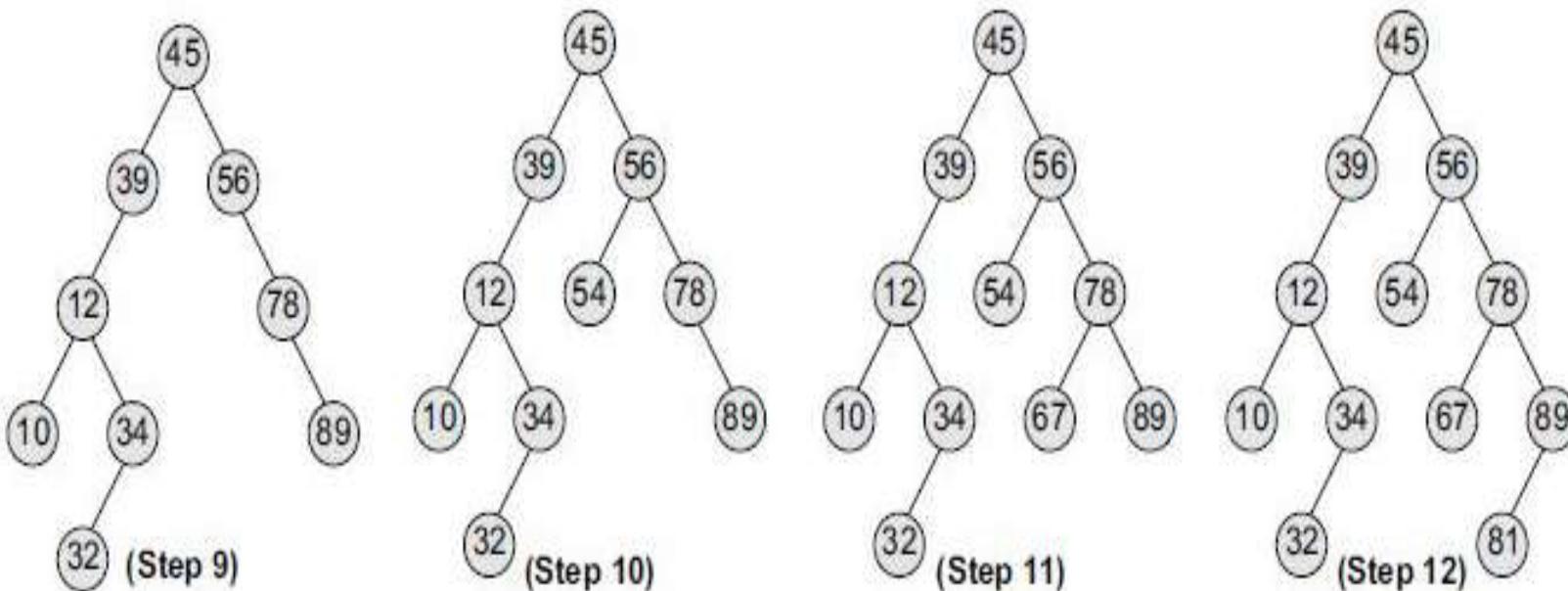
# Binary Search Tree

- Create a binary search tree using the following data elements:  
45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



# Binary Search Tree

- Create a binary search tree using the following data elements:  
45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



# Binary Search implementation

Struct tree{

    int data;

    struct tree \*left;

    struct tree \*right;

}

Struct tree \*t;

# Operations on BST

## Modifier Methods

1. Insert
2. Delete
  1. Leaf
  2. Non-Leaf
    1. With-one-child
    2. With-both-children

## Support Methods

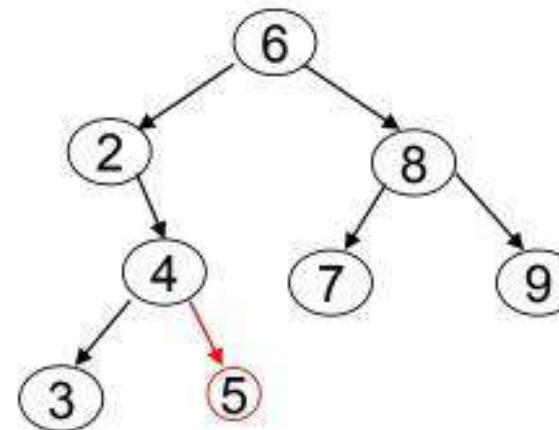
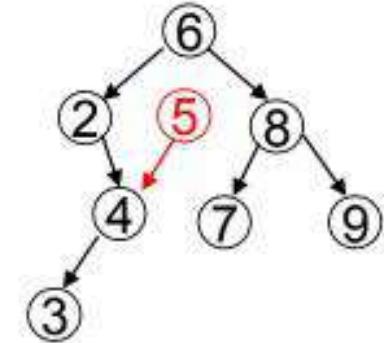
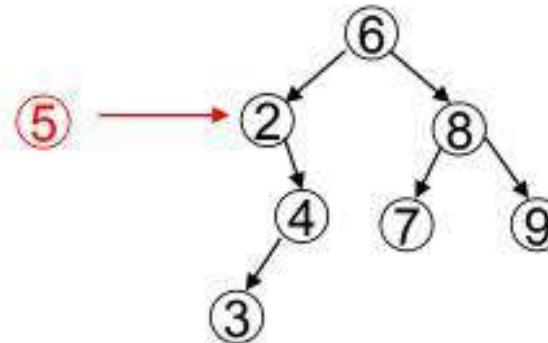
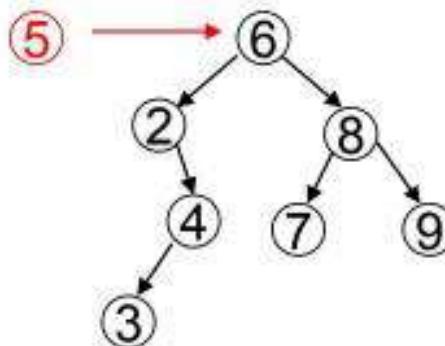
1. isempty
2. Search
  1. Element
  2. GetParent
  3. GetChildren
4. getMax
5. getMin
3. Traverse
  1. Inorder
  2. Preorder
  3. Postorder



**SOMAIYA**  
VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

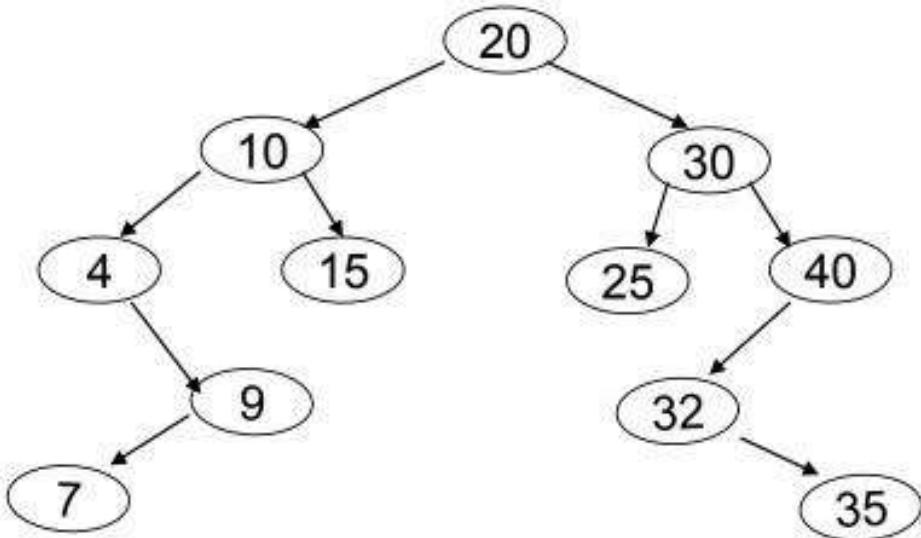
# Insertion in a BST



# Insertion in a BST

Insert following numbers on an empty BST

20, 30, 10, 4, 40, 9, 7, 15, 25, 32, 35



**SOMAIYA**  
VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

Somaiya  
20  
TRUST

# Insertion in a BST - Pseudocode

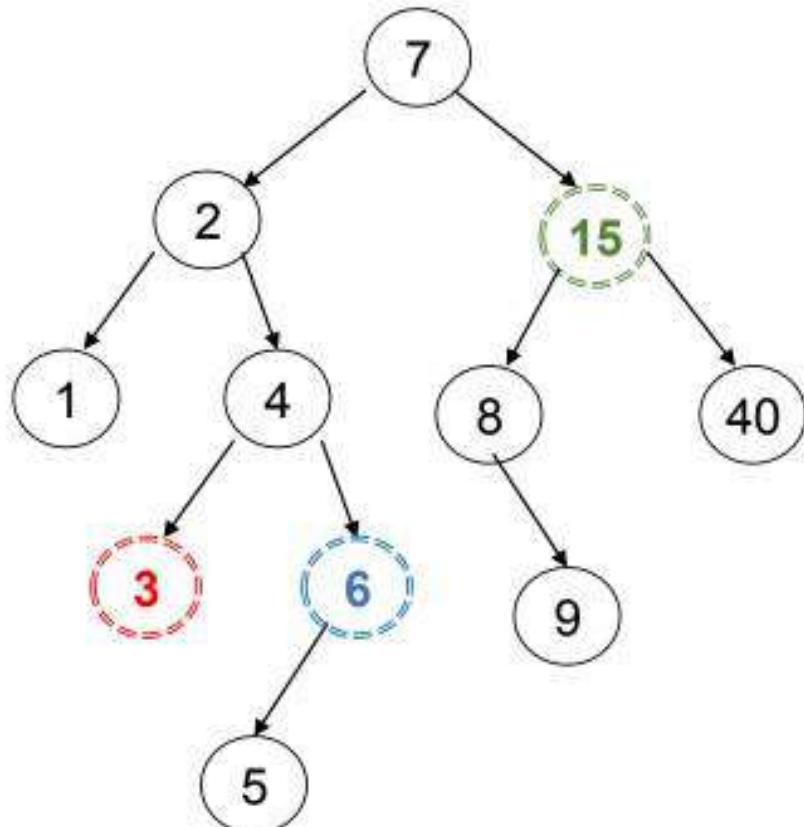
```
If(!isempty())
    ref = root
    dup=0
    while(ref !=NULL)
        if(ref->data > x)
            pnode=ref
            ref = ref->left
        else if(ref->data < x)
            pnode=ref
            ref = ref->right
        else
            dup=1
            break
    endwhile

    If(dup==0)
        if(pnode->data > x)
            pnode->left = xnode
        else
            pnode->right = xnode
        else
            print "duplicate data....not
                  allowed"
    else
        root = xnode
```

# Deletion in a BST

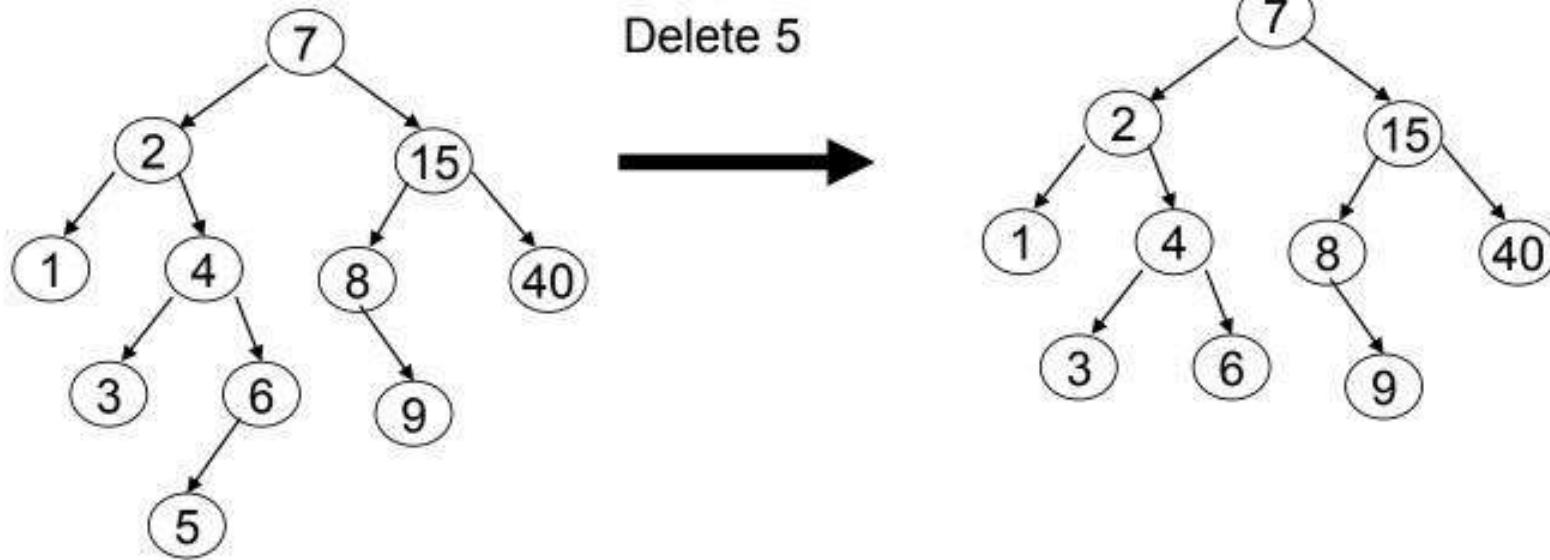
There are three cases:

1. The node to be deleted is a leaf node.
2. The node to be deleted has one non-empty child.
3. The node to be deleted has two non-empty children.



# Case 1 : Deleting a leaf node

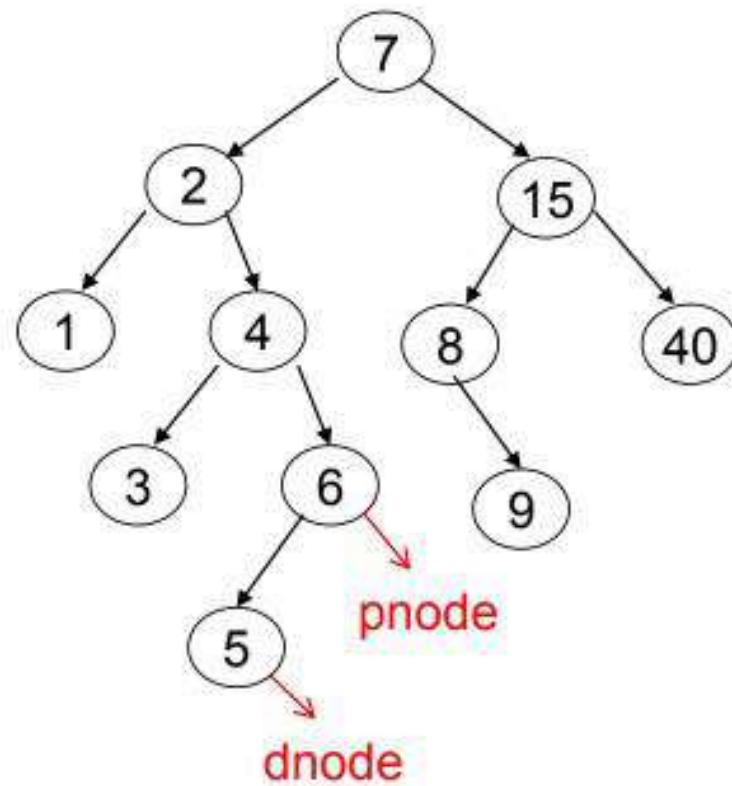
Example: Delete 5 in the tree below:



# Deleting a node (common for all cases)

## *Step 1 – setting up pointers*

```
Node *pnode, *dnode
If(!isempty())
dnode = root
while(dnode !=NULL)
    if(dnode->data > x)
        pnode = dnode
        dnode = dnode->left
    else if (dnode->data < x)
        pnode = dnode
        dnode = dnode->right
    else
        break
End while
```



# Case 1 : Deleting a leaf node (cont'd)

## Step 2 – checking leaf and deleting leaf

```
If(dnode -> left ==NULL && dnode -> right ==NULL)
```

```
    if(pnode->left == dnode)
```

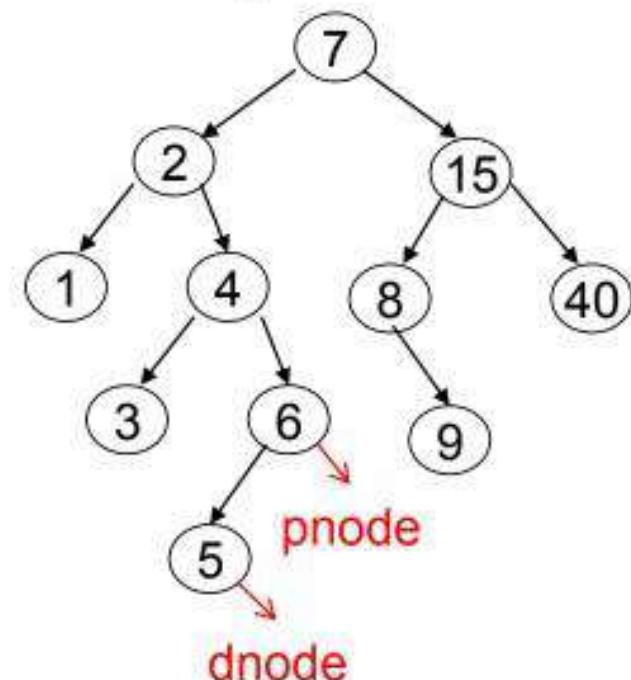
```
        pnode->left = NULL
```

```
    else if(pnode->right == dnode)
```

```
        pnode->right = NULL
```

```
free(dnode)
```

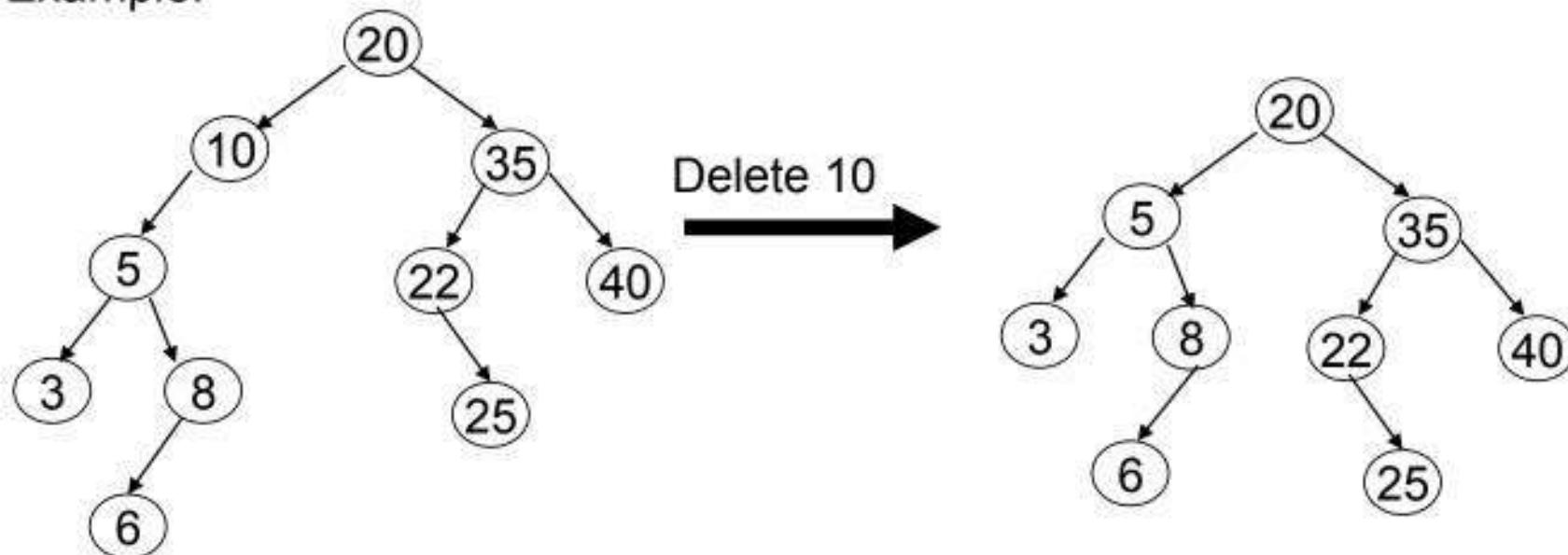
```
return(x)
```



# Case 2: Deleting a one-child node

(a) The right subtree of the node x to be deleted is empty.

Example:



# Case 2: Deleting a one-child node

(a) The right subtree of the node x to be deleted is empty.

Step 1 – setting up pointers

**Step 2 – checking right subtree is empty and  
relinking left subtree**

```
If(dnode -> left !=NULL && dnode -> right ==NULL)
```

```
    if(pnode->left == dnode)
```

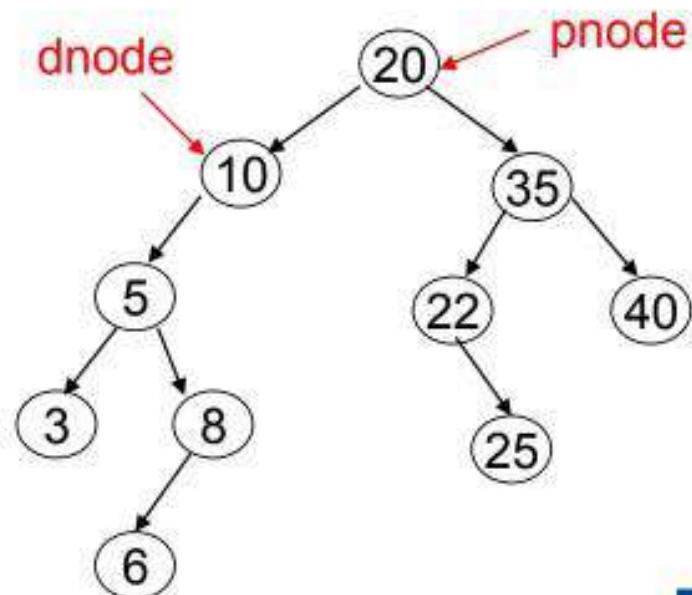
```
        pnode->left = dnode -> left
```

```
    else if(pnode->right == dnode)
```

```
        pnode->right = dnode -> left
```

```
free(dnode)
```

```
return(x)
```



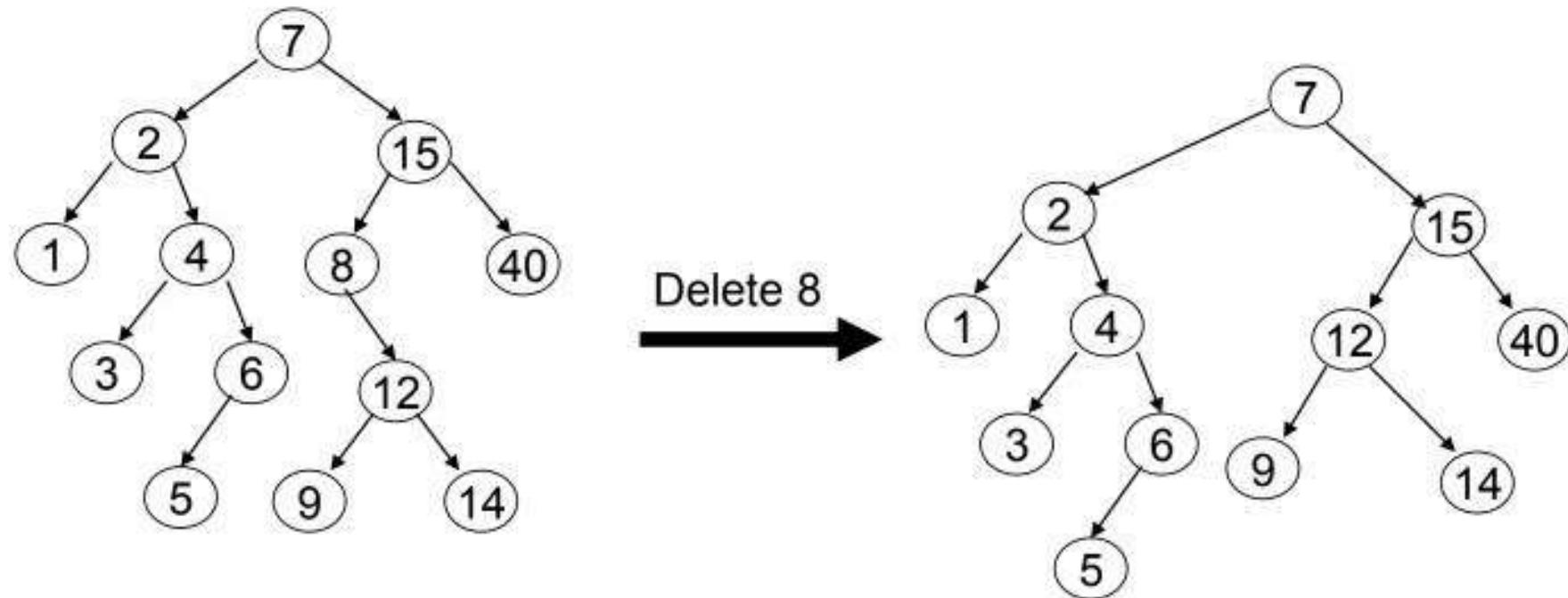
**SOMAIYA**  
VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

Somaiya  
SS  
TRUST

# Case 2: Deleting a one-child node

(b) The left subtree of the node x to be deleted is empty.



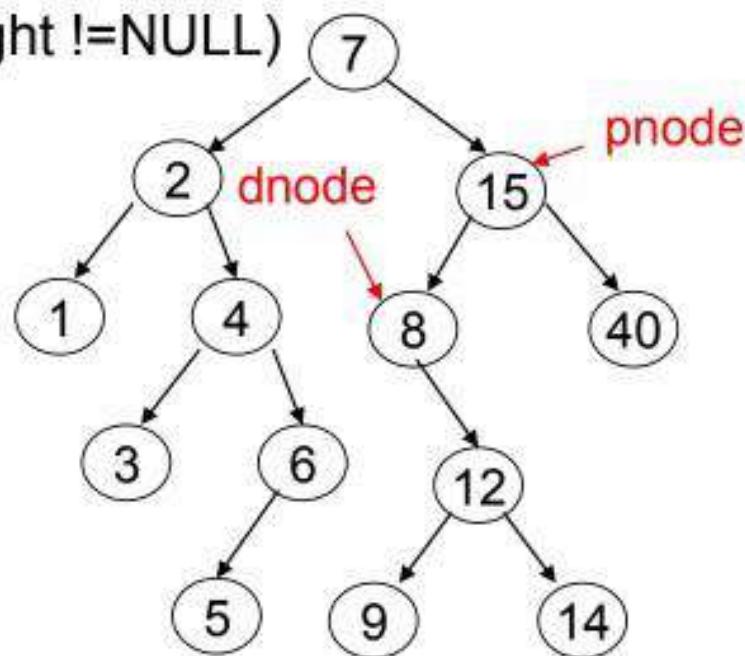
# Case 2: Deleting a one-child node

(b) The left subtree of the node x to be deleted is empty.

Step 1 – setting up pointers

**Step 2 – checking left subtree is empty and  
relinking right subtree**

```
If(dnode -> left ==NULL && dnode -> right !=NULL)
    if(pnode->left == dnode)
        pnode->left = dnode -> right
    else if(pnode->right == dnode)
        pnode->right = dnode -> right
    free(dnode)
    return(x)
```

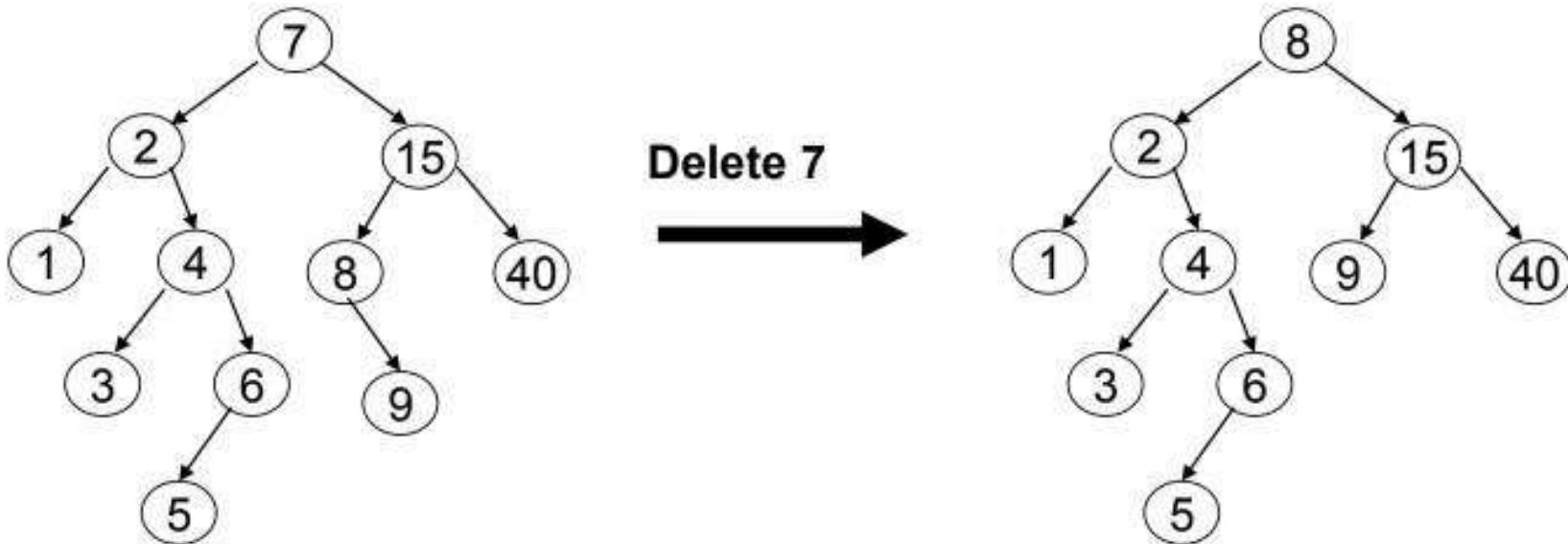


## Case 3: Deleting a node that has two non-empty children

## DELETION BY COPYING: METHOD#1

Copy the **minimum** key in the **right** subtree of  $x$  to the node  $x$ , then apply case 2 - delete one-child or case 1 – delete leaf to this minimum key node.

### Example:



# Case 3: Deleting a node that has two non-empty children

## DELETION BY COPYING: METHOD#1

Step 1 – setting up pointers

Step 2 – find min of right subtree and relinking right subtree

```
If(dnode -> left !=NULL && dnode -> right !=NULL)
```

```
    min=dnode->right
```

```
    while(min->left != NULL)
```

```
        rnode = min
```

```
        min = min->left
```

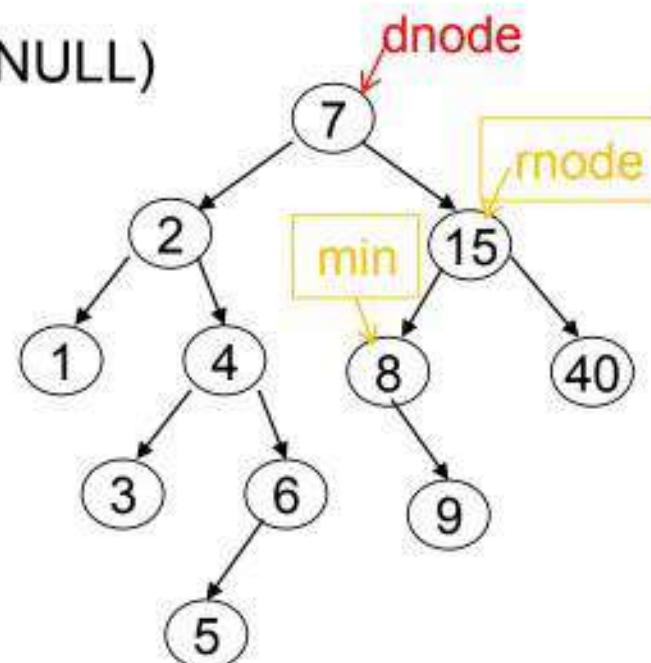
```
    dnode->data = min->data;
```

```
    if(min->right != NULL)
```

```
        rnode->left = min -> right
```

```
    free(min)
```

```
    return(x)
```

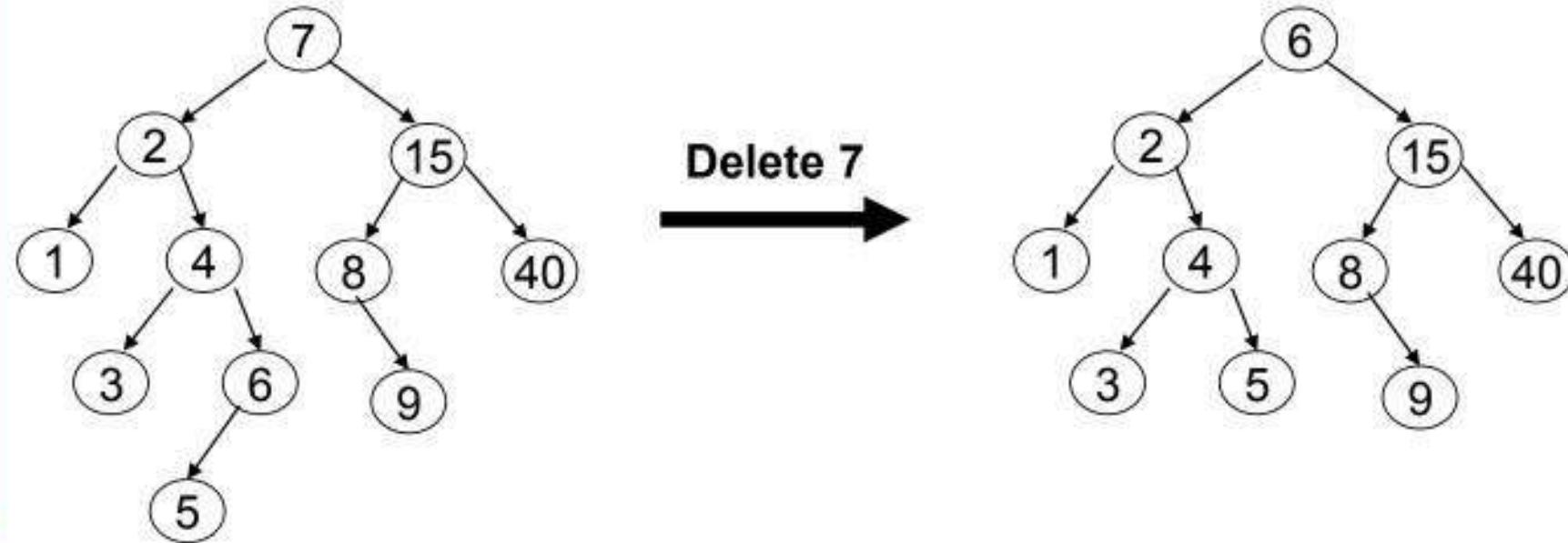


# Case 3: Deleting a node that has two non-empty children

## DELETION BY COPYING: METHOD#2

Copy the **maximum** key in the **left** subtree of x to the node x, then apply case 2 - delete one-child or case 1 – delete leaf to this minimum key node.

Example:



# Operations on BST : Searching for a Node

- The search function is used to find whether a **given value** is present in the tree or not.
- The searching process **begins** at the root node.
- The function first checks if the binary search tree is **empty**. If it is empty, then the value we are searching for is not present in the tree.
- So, the search algorithm **terminates** by displaying an appropriate message.
- However, if there are nodes in the tree, then the search function checks to see if the **key value of the current node is equal to the value to be searched**.
- If not, it checks if the value to be searched for **is less than the value of the current node**, in which case it should be recursively called on the left child node.
- In case the value **is greater than the value** of the current node, it should be recursively called on the **right child node**.

# Operations on Binary Search Trees

- **Searching for a Node in a Binary Search Tree**

```
searchElement (TREE, VAL)
```

```
Step 1: IF TREE->DATA = VAL OR TREE = NULL
```

```
    Return TREE
```

```
ELSE
```

```
    IF VAL < TREE->DATA
```

```
        Return searchElement(TREE->LEFT, VAL)
```

```
    ELSE
```

```
        Return searchElement(TREE->RIGHT, VAL)
```

```
    [END OF IF]
```

```
    [END OF IF]
```

```
Step 2: END
```

# Operations on Binary Search Trees

- **Inserting a New Node in a Binary Search Tree**
- The insert function is used to add a new node with a given **value at the correct position** in the binary search tree.
- Adding the node at the correct position means that the new node should not violate the properties of the binary search tree.
- The initial code for the insert function is similar to the search function. This is because we first find the correct position where the insertion has to be done and then add the node at that position.
- The insertion function changes the structure of the tree. Therefore, when the insert function is called recursively, the function should return the new tree pointer.

# Operations on Binary Search Trees

- **Inserting a New Node in a Binary Search Tree**

```
Insert (TREE, VAL)
```

```
Step 1: IF TREE = NULL
```

```
    Allocate memory for TREE
```

```
    SET TREE->DATA = VAL
```

```
    SET TREE->LEFT = TREE->RIGHT = NULL
```

```
ELSE
```

```
    IF VAL < TREE->DATA
```

```
        Insert(TREE->LEFT, VAL)
```

```
    ELSE
```

```
        Insert(TREE->RIGHT, VAL)
```

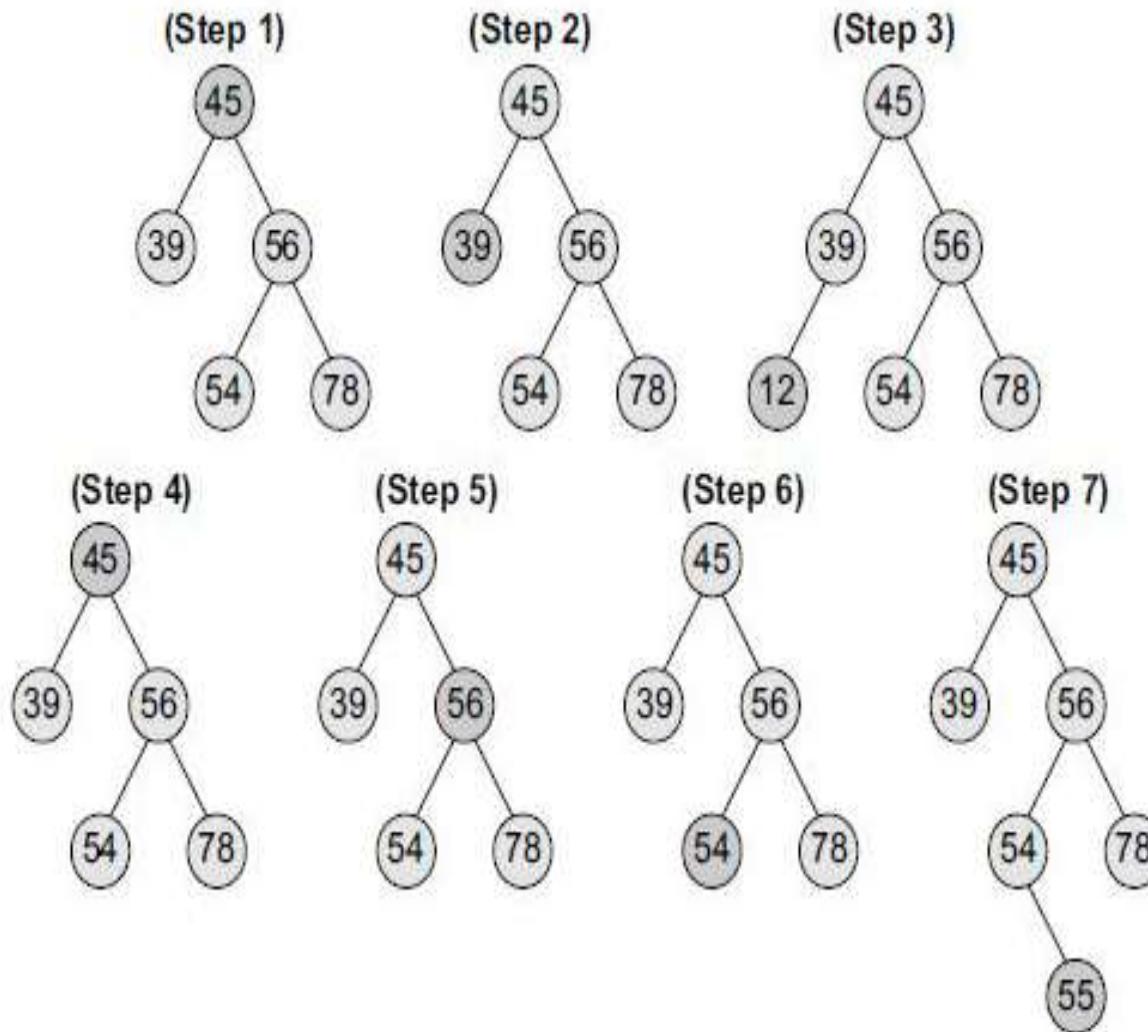
```
    [END OF IF]
```

```
    [END OF IF]
```

```
Step 2: END
```

# Operations on Binary Search Trees

- We will take up the case of inserting 12 and 55.

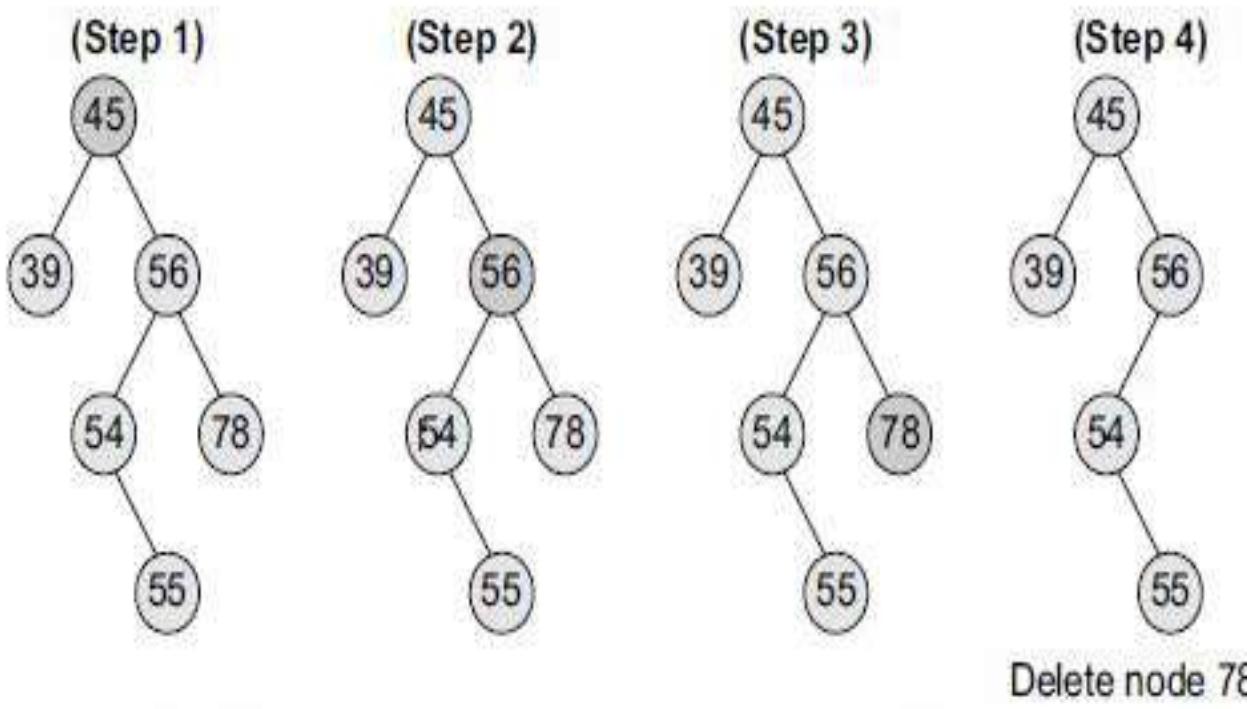


# Operations on Binary Search Trees

- **Deleting a Node from a Binary Search Tree**
- The delete function deletes a node from the binary search tree. However, utmost care should be taken that the properties of the binary search tree are not violated and nodes are not lost in the process.
- We will take up three cases and discuss how a node is deleted from a binary search tree.
- ***Case 1: Deleting a Node that has No Children***
- ***Case 2: Deleting a Node with One Child***
- ***Case 3: Deleting a Node with Two Children***

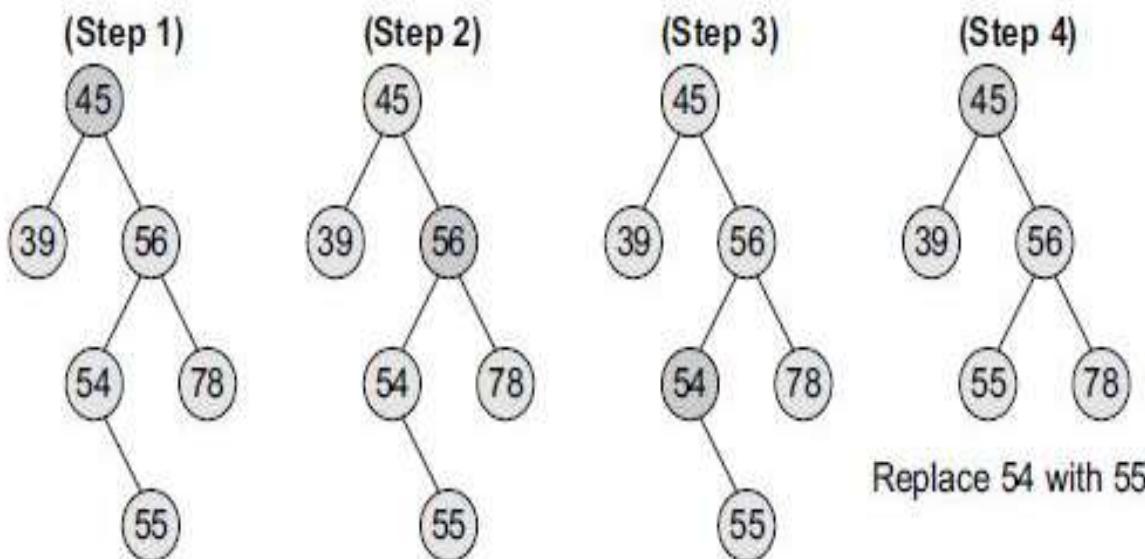
# Operations on Binary Search Trees

- ***Case 1: Deleting a Node that has No Children***
- Look at the binary search tree given. If we have to delete node 78, we can simply remove this node without any issue. This is the simplest case of deletion.



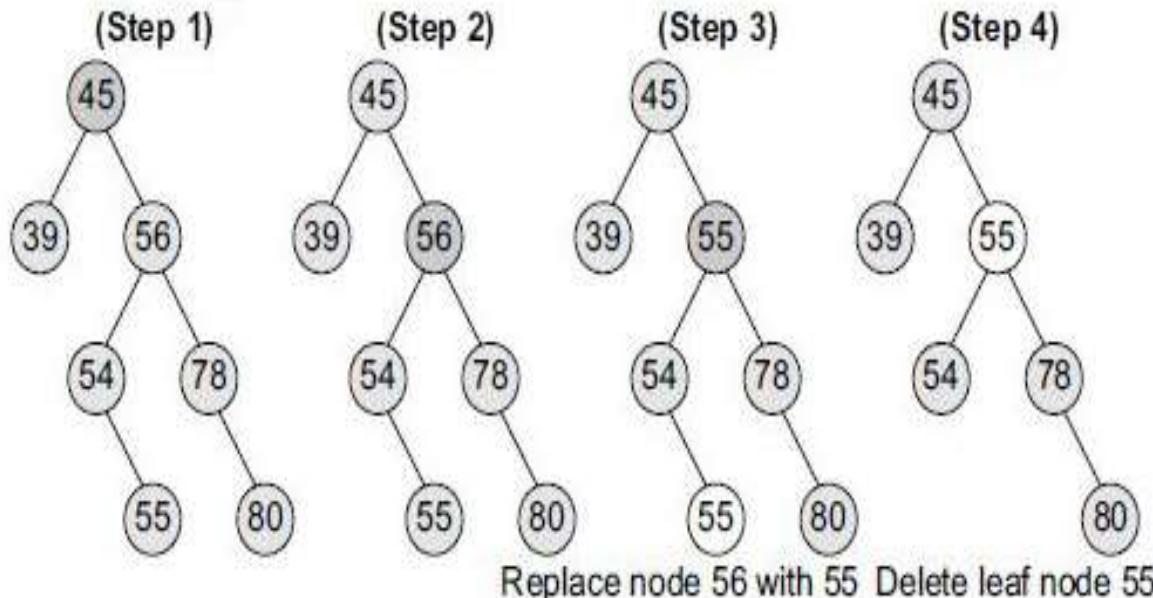
# Operations on Binary Search Trees

- **Case 2: Deleting a Node with One Child**
- The node's child is set as the child of the node's parent. In other words, **replace the node with its child**. Now, if the node is the left child of its parent, the node's child becomes the left child of the node's parent. Correspondingly, if the node is the right child of its parent, the node's child becomes the right child of the node's parent.



# Operations on Binary Search Trees

- **Case 3: Deleting a Node with Two Children**
- Replace the node's value with its *in-order predecessor* (largest value in the left sub-tree) or *in-order successor* (smallest value in the right sub-tree). The in-order predecessor or the successor can then be deleted using any of the above cases. Let us see how deletion of node with value 56 is handled.



# Operations on Binary Search Trees

- ***Case 3: Deleting a Node with Two Children***
- This deletion could also be handled by replacing node 56 with its in-order successor

# Operations on Binary Search Trees

- In Step 1 of the algorithm, we first check if TREE=NULL, because if it is true, then the node to be deleted is not present in the tree.
- However, if that is not the case, then we check if the value to be deleted is less than the current node's data. In case the value is less, we call the algorithm recursively on the node's left sub-tree, otherwise the algorithm is called recursively on the node's right sub-tree.
- Note that if we have found the node whose value is equal to VAL, then we check which case of deletion it is.

# Operations on Binary Search Trees

- If the node to be deleted does not have any child, then we simply set the node to NULL.
- If the node to be deleted has either a left or a right child but not both, then the current node is replaced by its child node and the initial child node is deleted from the tree.
- If the node to be deleted has both left and right children, then we find the in-order predecessor of the node or in-order successor .

# Operations on Binary Search Trees

- If the node to be deleted has both left and right children, then we find the in-order predecessor of the node by calling
- `findLargestNode(TREE -> LEFT)` and
- replace the current node's value with that of its in-order predecessor.
- Then, we call `Delete(TREE -> LEFT, TEMP -> DATA)` to delete the initial node of the in-order predecessor.
- Thus, we reduce the case 3 of deletion into either case 1 or case 2 of deletion.

# Operations on Binary Search Trees

## Delete (TREE, VAL)

Step 1: IF TREE = NULL

    Write "VAL not found in the tree"

ELSE IF VAL < TREE->DATA

    Delete(TREE->LEFT, VAL)

ELSE IF VAL > TREE->DATA

    Delete(TREE->RIGHT, VAL)

ELSE IF TREE->LEFT AND TREE->RIGHT

    SET TEMP = findLargestNode(TREE->LEFT)

    SET TREE->DATA = TEMP->DATA

    Delete(TREE->LEFT, TEMP->DATA)

ELSE

    SET TEMP = TREE

    IF TREE->LEFT = NULL AND TREE->RIGHT = NULL

        SET TREE = NULL

    ELSE IF TREE->LEFT != NULL

        SET TREE = TREE->LEFT

    ELSE

        SET TREE = TREE->RIGHT

    [END OF IF]

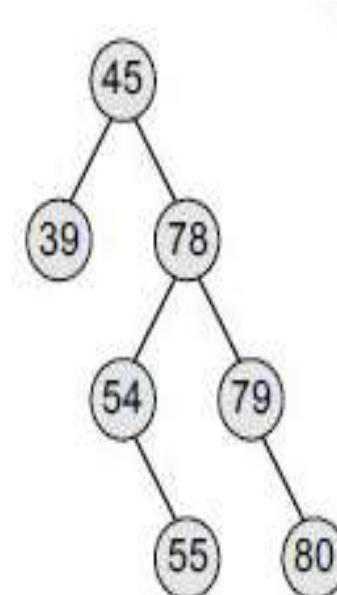
    FREE TEMP

    [END OF IF]

Step 2: END

# Determining the Number of Nodes

- To calculate the total number of elements/nodes the tree, we count the number of nodes in the left sub-tree and the right sub-tree.
- Number of nodes =  $\text{totalNodes}(\text{left sub-tree}) + \text{totalNodes}(\text{right sub-tree}) + 1$
- The total number of nodes in the tree can be calculated as:
  - Total nodes of left sub-tree = 1
  - Total nodes of left sub-tree = 5
  - Total nodes of tree =  $(1 + 5) + 1$
  - Total nodes of tree = 7



# Determining the Number of Nodes

**totalNodes (TREE)**

Step 1: IF TREE = NULL

    Return 0

ELSE

    Return totalNodes (TREE → LEFT)

        + totalNodes (TREE → RIGHT) + 1

    [END OF IF]

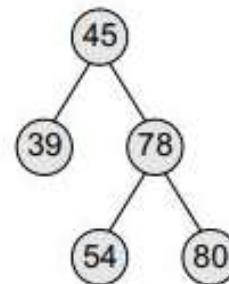
Step 2: END

# Determining the Height of BST

Calculate the height of the left sub-tree and the right sub-tree. Whichever height is greater, 1 is added to it.

Eg. if the height of the left sub-tree is greater than that of the right sub-tree, then 1 is added to the left sub-tree, else 1 is added to the right sub-tree.

Since the height of the right sub-tree is greater than the height of the left sub-tree, the height of the tree = height (right sub-tree) + 1 = 2 + 1 = 3.



**Figure 10.16** Binary search tree with height = 3

## **Algorithm:**

1. Check if the current node of the TREE = NULL. If the condition is true, then 0 is returned to the calling code.
2. Otherwise, for every node, we recursively call the algorithm to calculate the height of its left sub-tree as well as its right sub-tree.
3. The height of the tree at that node is given by adding 1 to the height of the left sub-tree or the height of right sub-tree, whichever is greater

# Determining the Smallest Node of BST

## ALGORITHM 7-1 Find Smallest Node in a BST

```
Algorithm findSmallestBST (root)
```

This algorithm finds the smallest node in a BST.

Pre      root is a pointer to a nonempty BST or subtree

Return address of smallest node

```
1 if (left subtree empty)
  1 return (root)
2 end if
3 return findSmallestBST (left subtree)
end findSmallestBST
```

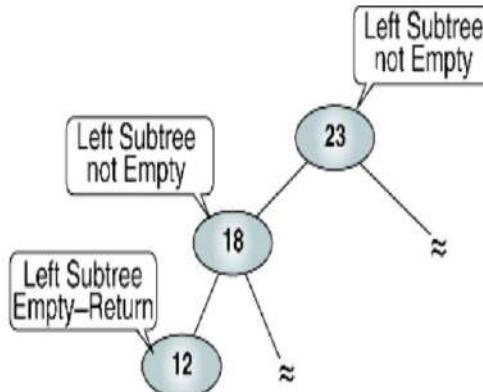


FIGURE 7-5 Find Smallest Node in a BST

# Determining the Largest Node of BST

## ALGORITHM 7-2 Find Largest Node in a BST

```
Algorithm findLargestBST (root)
```

This algorithm finds the largest node in a BST.

Pre      root is a pointer to a nonempty BST or subtree

Return address of largest node returned

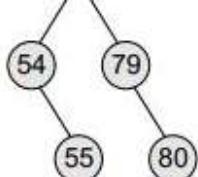
```
1 if (right subtree empty)
```

```
    1 return (root)
```

```
2 end if
```

```
3 return findLargestBST (right subtree)
```

```
end findLargestBST
```



**Figure 10.18** Binary search tree

### **Determining the Number of Internal Nodes**

To calculate the total number of internal nodes or non-leaf nodes, we count the number of internal nodes in the left sub-tree and the right sub-tree and add 1 to it (1 is added for the root node).

$$\begin{aligned} \text{Number of internal nodes} = \\ \text{totalInternalNodes(left sub-tree)} + \\ \text{totalInternalNodes(right sub-tree)} + 1 \end{aligned}$$

Consider the tree given in Fig. 10.18. The total number of internal nodes in the tree can be calculated as:

$$\begin{aligned} \text{Total internal nodes of left sub-tree} &= 0 \\ \text{Total internal nodes of right sub-tree} &= 3 \\ \text{Total internal nodes of tree} &= (0 + 3) + 1 \\ &= 4 \end{aligned}$$

Figure 10.20 shows a recursive algorithm to calculate the total number of internal nodes in a binary search tree. For every node, we recursively call the algorithm on its left sub-tree as well as the right sub-tree. The total number of internal nodes at a given node is then returned by adding internal nodes in its left as well as right sub-tree. However, if the tree is empty, that is  $\text{TREE} = \text{NULL}$ , then the number of internal nodes will be zero. Also if there is only one node in the tree, then the number of internal nodes will be zero.

```

totalNodes(TREE)

Step 1: IF TREE = NULL
        Return 0
    ELSE
        Return totalNodes(TREE → LEFT)
            + totalNodes(TREE → RIGHT) + 1
    [END OF IF]
Step 2: END
  
```

**Figure 10.19** Algorithm to calculate the number of nodes in a binary search tree

```

totalInternalNodes(TREE)

Step 1: IF TREE = NULL
        Return 0
    [END OF IF]
    IF TREE → LEFT = NULL AND TREE → RIGHT = NULL
        Return 0
    ELSE
        Return totalInternalNodes(TREE → LEFT) +
            totalInternalNodes(TREE → RIGHT) + 1
    [END OF IF]
Step 2: END
  
```

**Figure 10.20** Algorithm to calculate the total number of internal nodes in a binary search tree

### **Determining the Number of External Nodes**

To calculate the total number of external nodes or leaf nodes, we add the number of

```
SET TREE -> RIGHT = TEMP
[END OF IF]
Step 2: END
```

**Figure 10.23** Algorithm to obtain the mirror image mirror image  $T'$  of a binary search tree

## 10.2.8 Finding the Smallest Node in a Binary Search Tree

The very basic property of the binary search tree states that the smaller value will occur in the left sub-tree. If

### 306 Data Structures Using C

the left sub-tree is `NULL`, then the value of the root node will be smallest as compared to the nodes in the right sub-tree. So, to find the node with the smallest value, we find the value of the leftmost node of the left sub-tree. The recursive algorithm to find the smallest node in a binary search tree is shown in Fig. 10.25.

```
deleteTree(TREE)

Step 1: IF TREE != NULL
        deleteTree (TREE -> LEFT)
        deleteTree (TREE -> RIGHT)
        Free (TREE)
    [END OF IF]
Step 2: END
```

**Figure 10.24** Algorithm to delete a binary search tree

```
findSmallestElement(TREE)

Step 1: IF TREE = NULL OR TREE -> LEFT = NULL
        Return TREE
    ELSE
        Return findSmallestElement(TREE -> LEFT)
    [END OF IF]
Step 2: END
```

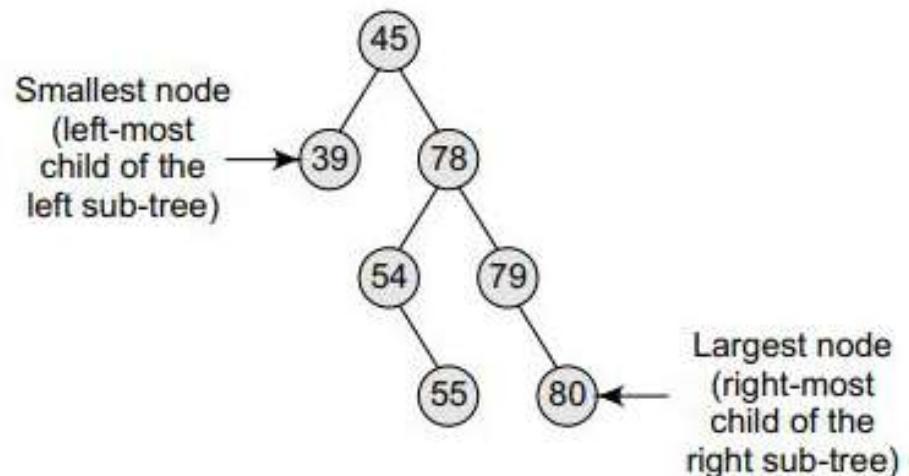
**Figure 10.25** Algorithm to find the smallest node in a binary search tree

### 10.2.9 Finding the Largest Node in a Binary Search Tree

To find the node with the largest value, we find the value of the rightmost node of the right sub-tree. However, if the right sub-tree is empty, then the root node will be the largest value in the tree. The recursive algorithm to find the largest node in a binary search tree is shown in Fig. 10.26.

```
findLargestElement(TREE)
```

```
Step 1: IF TREE = NULL OR TREE -> RIGHT = NULL
        Return TREE
    ELSE
        Return findLargestElement(TREE -> RIGHT)
    [END OF IF]
Step 2: END
```



**Figure 10.26** Algorithm to find the largest node in a binary search tree

**Figure 10.27** Binary search tree

**Code (all operations):**

[https://docs.google.com/document/d/11SZBG-ZefpKbYaEW08SIX6cLv3xbxDeZGMCSoln\\_GaU/edit?usp=sharing](https://docs.google.com/document/d/11SZBG-ZefpKbYaEW08SIX6cLv3xbxDeZGMCSoln_GaU/edit?usp=sharing)

# BST Examples

Construct BST

1. 10, 12, 34, 56, 71, 87

2. 50, 43, 37, 23, 21, 11, 9, 4, 30

3. 56, 78, 34, 23, 0, 26, 57, 68, 9, 36, 44, 30

Find Parent of 34, 26, 12, 9, 30

Delete 34, 26, 12, 9, 30

1.

10

\

12

\

34

\

56

\

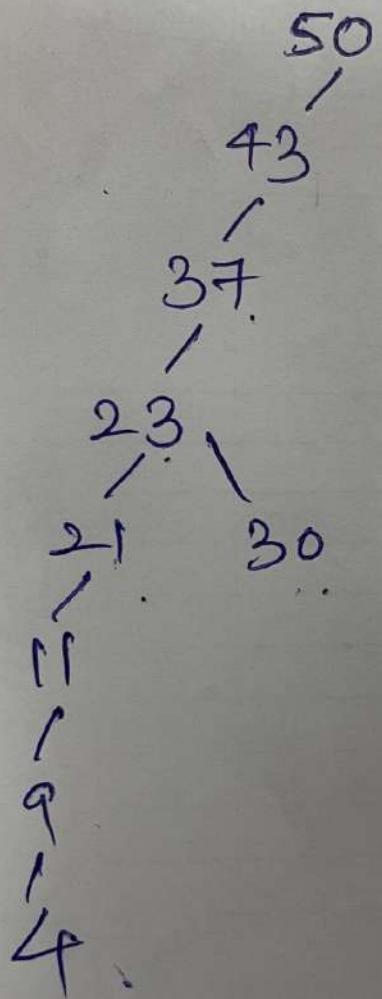
71

\

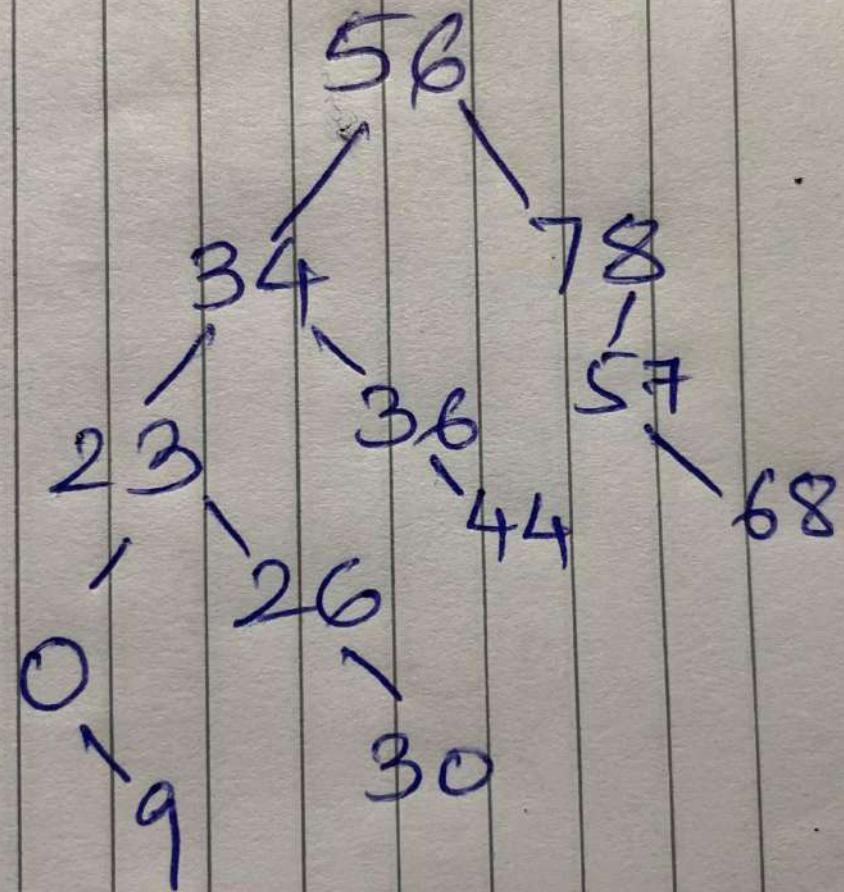
87

2.

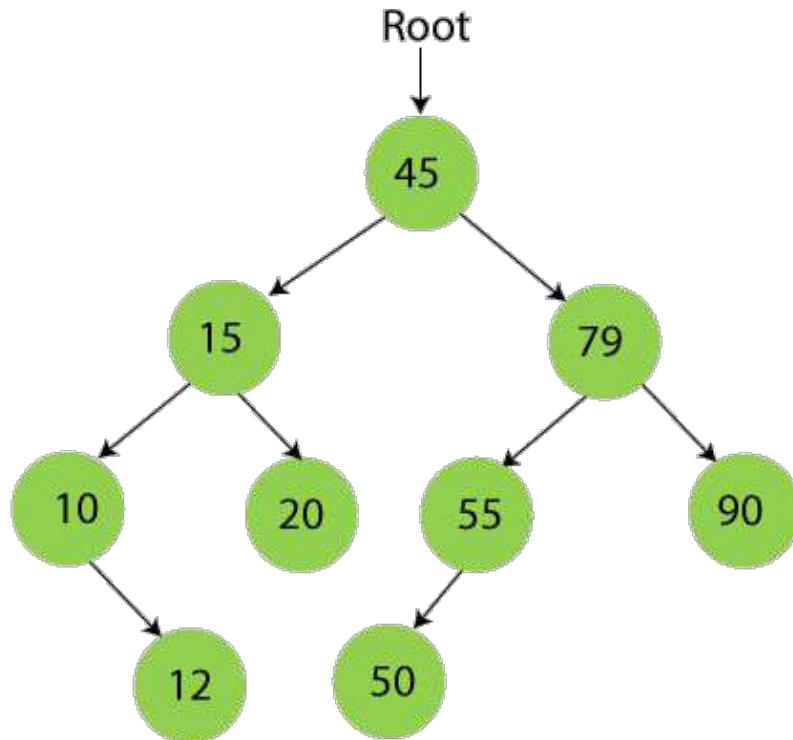
②



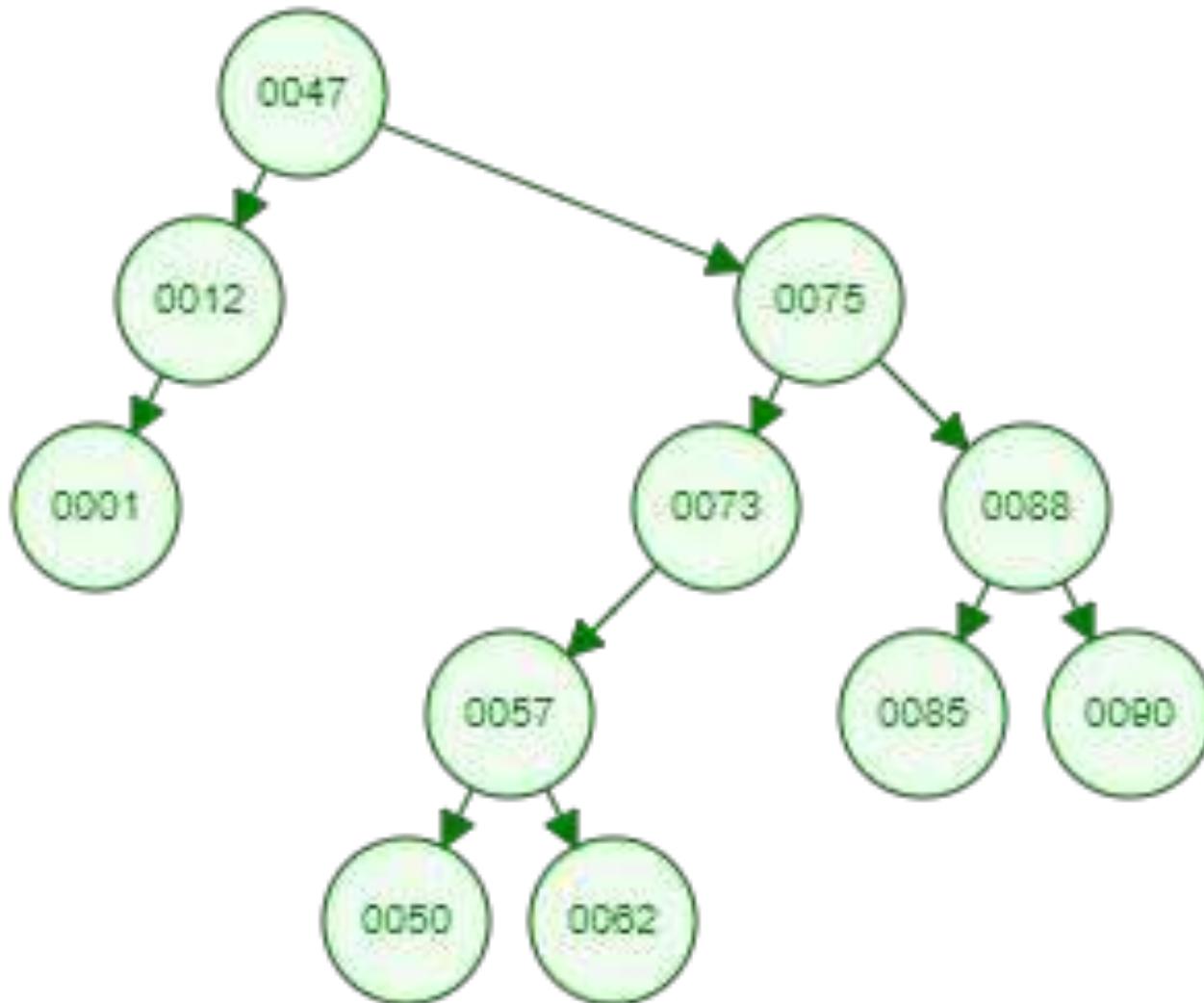
3.



Create a BST with following data elements - 45, 15, 79, 90, 10, 55, 12, 20, 50



Construct Binary search tree for the following elements 47,12,75,88,90,73,57,1,85,50,62

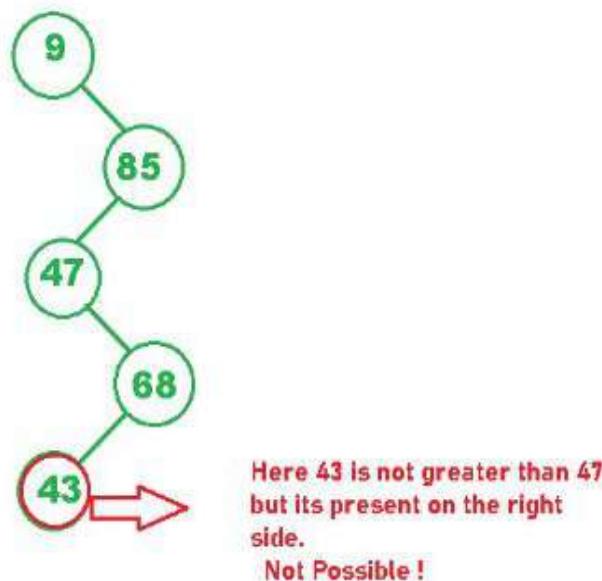


Suppose that we have numbers between 1 and 100 in a binary search tree and want to search for the number 55. Which of the following sequences CANNOT be the sequence of nodes examined?

- (A) {10, 75, 64, 43, 60, 57, 55}
  - (B) {90, 12, 68, 34, 62, 45, 55}
  - (C) {9, 85, 47, 68, 43, 57, 55}
  - (D) {79, 14, 72, 56, 16, 53, 55}

Answer: (C)

**Explanation:** In BST, on right child of parent should be greater than parent and left child should be smaller than the parent, but in C after 47, 68 goes on the right side because it greater then parent, now everything below this point should be greater then 47 but 43 appears that does not satisfy the BST property.



**Direction: A binary search tree is constructed by inserting the following numbers in order.**

**60, 25, 72, 15, 30, 68, 101, 13, 18, 47, 70, 34**

**The number of nodes in the left subtree is:**

**Answer:**

**Left subtree = 7**

**Right subtree = 4**

# Tree Traversals

- A binary tree is defined recursively: it consists of a root, a left subtree, and a right subtree.
- To traverse (or walk) the binary tree is to visit each node in the binary tree exactly once.
- Tree traversals are naturally recursive.
- Standard traversal orderings:
  - **preorder**
  - **inorder**
  - **postorder**
  - **level-order**

# Preorder, Inorder, Postorder

- In Preorder, the root is visited before (pre) the subtrees traversals.
- In Inorder, the root is visited in-between left and right subtree traversal.
- In Preorder, the root is visited after (pre) the subtrees traversals.

## Preorder Traversal:

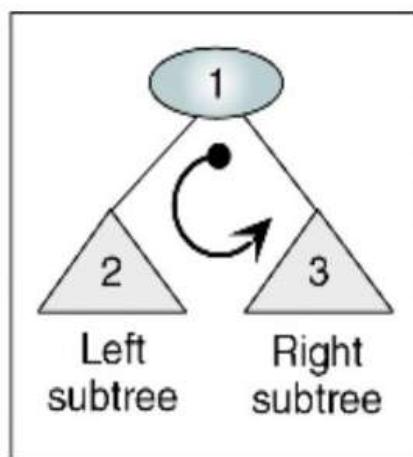
1. Visit the root
2. Traverse left subtree
3. Traverse right subtree

## Inorder Traversal:

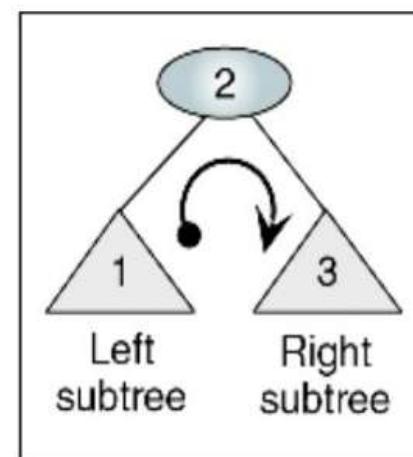
1. Traverse left subtree
2. Visit the root
3. Traverse right subtree

## Postorder Traversal:

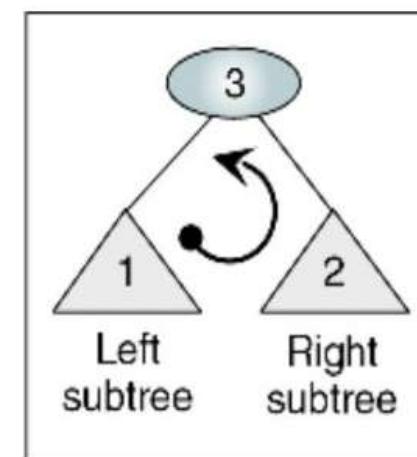
1. Traverse left subtree
2. Traverse right subtree
3. Visit the root



(a) Preorder traversal

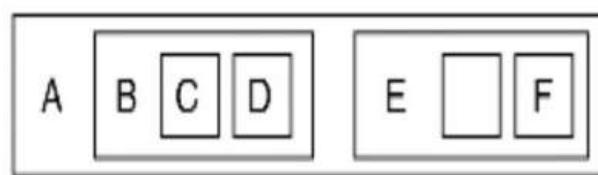
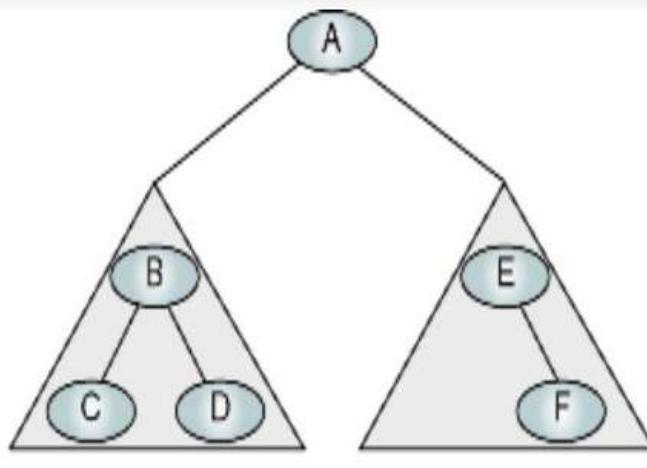


(b) Inorder traversal

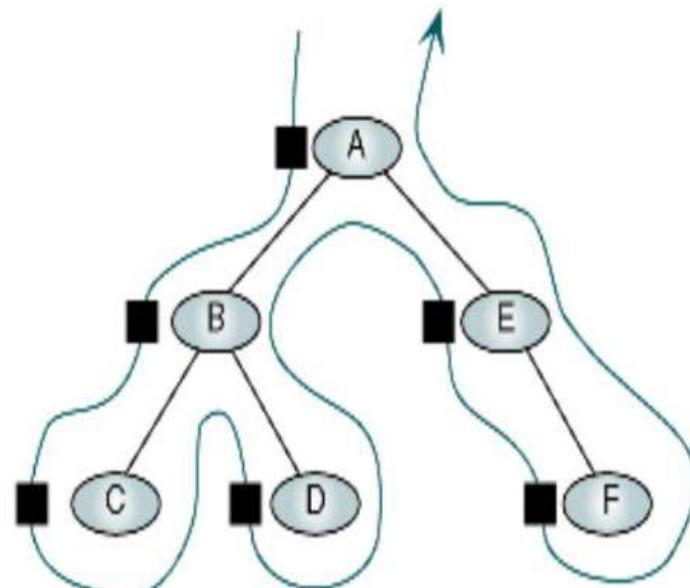


(c) Postorder traversal

FIGURE 6-8 Binary Tree Traversals

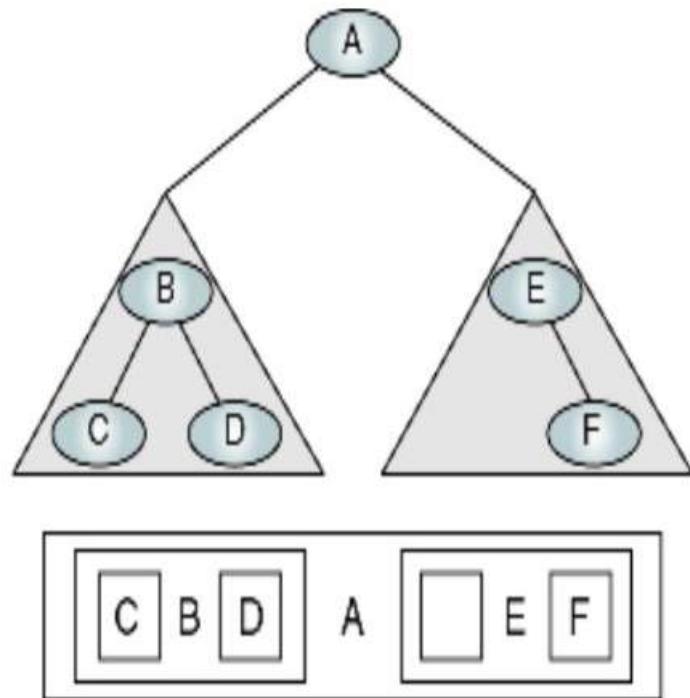


(a) Processing order

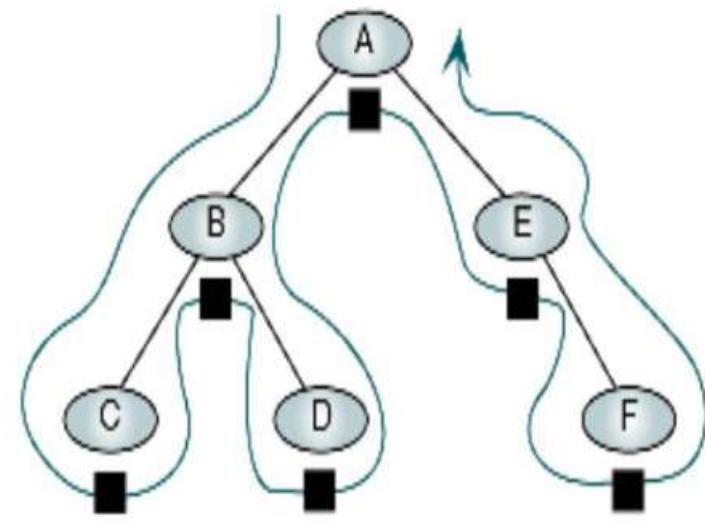


(b) "Walking" order

FIGURE 6-10 Preorder Traversal—A B C D E F

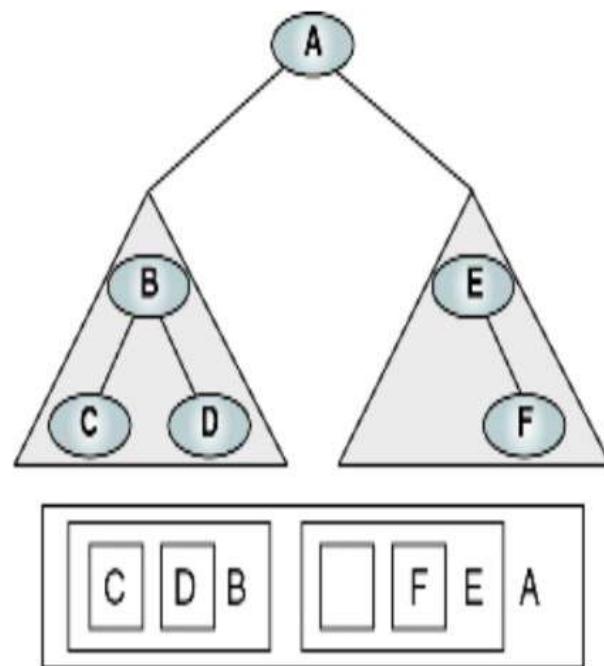


(a) Processing order

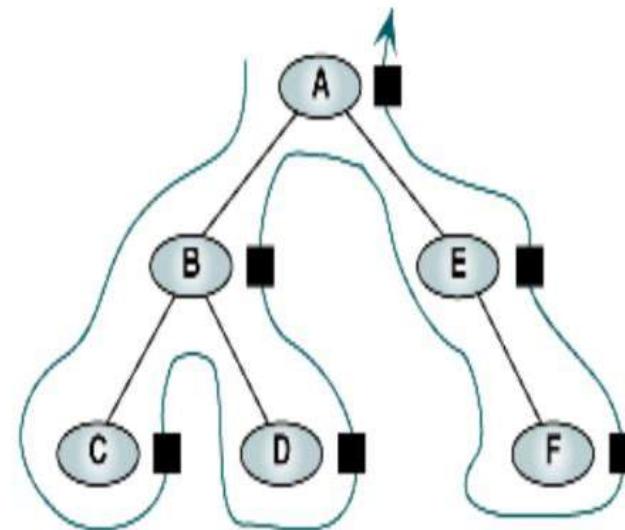


(b) "Walking" order

FIGURE 6-12 Inorder Traversal—C B D A E F



(a) Processing order

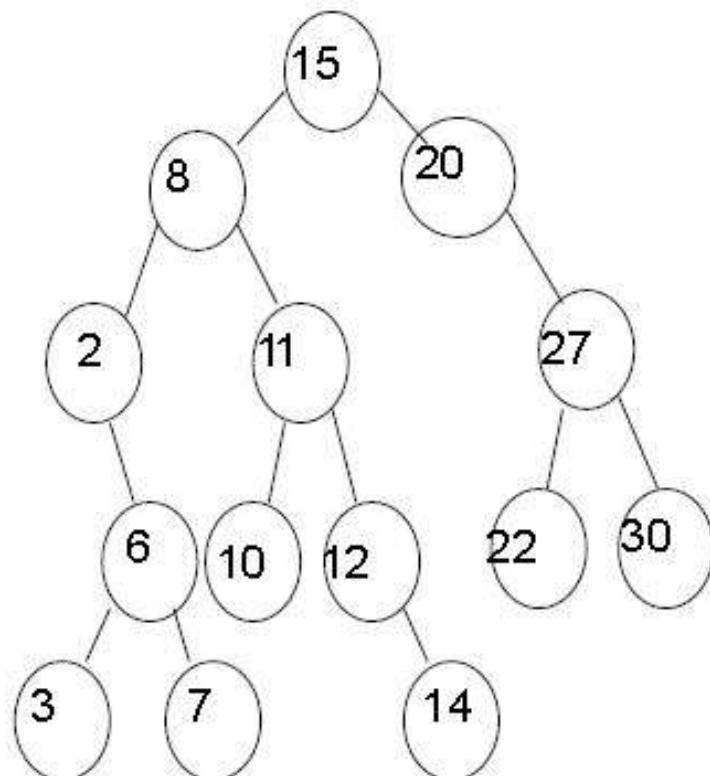


(b) "Walking" order

FIGURE 6-13 Postorder Traversal—C D B F E A

# Example of Tree Traversal

- Assume: visiting a node is printing its data
- Preorder: 15 8 2 6 3 7 11 10 12 14 20 27 22 30
- Inorder: 2 3 6 7 8 10 11 12 14 15 20 22 27 30
- Postorder: 3 7 6 2 10 14 12 11 8 22 30 27 20 15

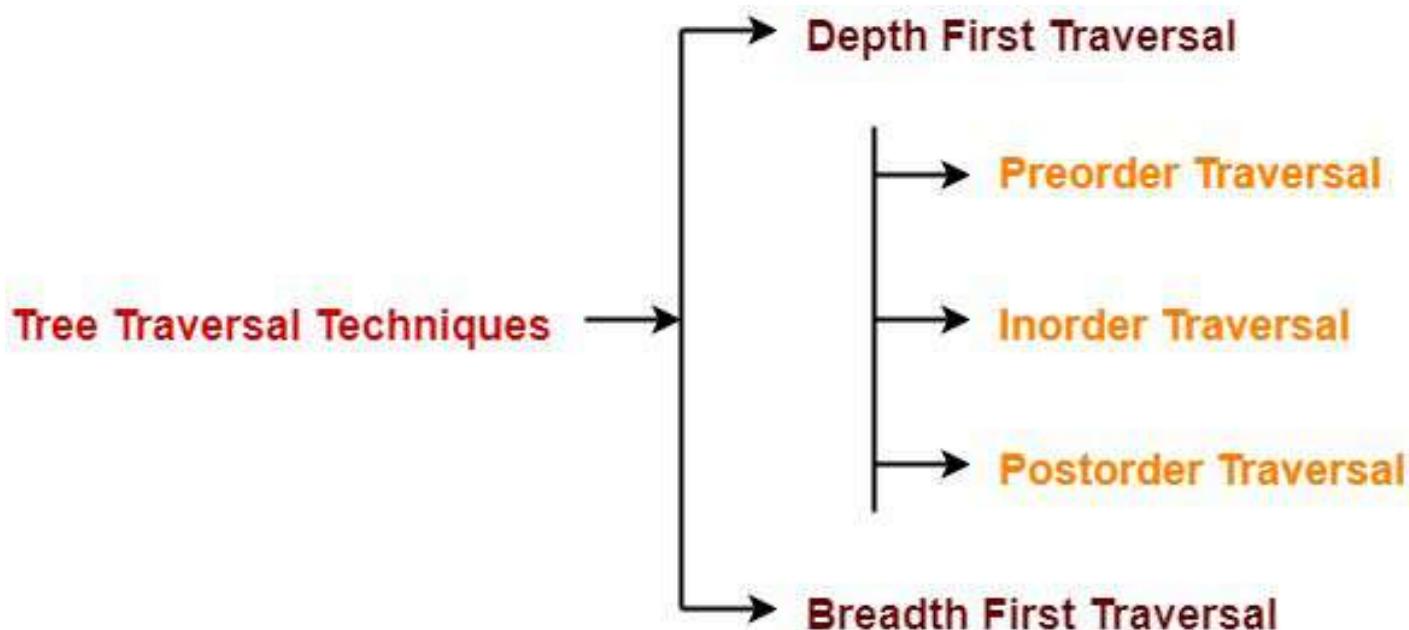


# Traversal Techniques

```
void preorder(tree *tree) {  
    if (tree->isEmpty())    return;  
    visit(tree->getRoot());  
    preOrder(tree->getLeftSubtree());  
    preOrder(tree->getRightSubtree());  
}  
  
void inOrder(Tree *tree){  
    if (tree->isEmpty())    return;  
    inOrder(tree->getLeftSubtree());  
    visit(tree->getRoot());  
    inOrder(tree->getRightSubtree());  
}  
  
void postOrder(Tree *tree){  
    if (tree->isEmpty())    return;  
    postOrder(tree->getLeftSubtree());  
    postOrder(tree->getRightSubtree());  
    visit(tree->getRoot());  
}
```

# Binary Tree Traversal

- Tree Traversal refers to the process of visiting each node in a tree data structure exactly once.



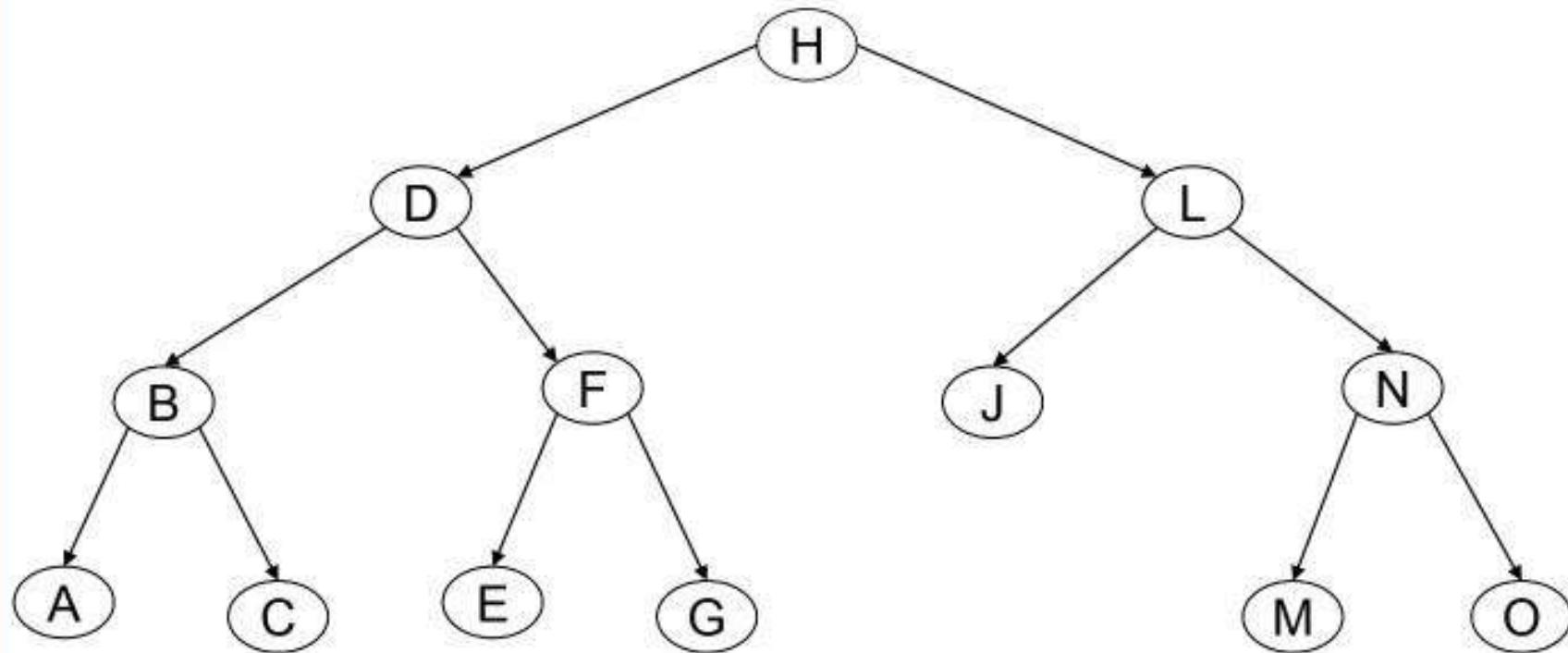
# Tree Traversal (Definition)

The process of systematically visiting all the nodes in a tree and performing some computation at each node in the tree is called a tree traversal.

There are two methods in which to traverse a tree:

1. Breadth-First Traversal.
2. Depth-First Traversal:
  - i. Preorder traversal
  - ii. Inorder traversal (for binary trees only)
  - iii. Postorder traversal

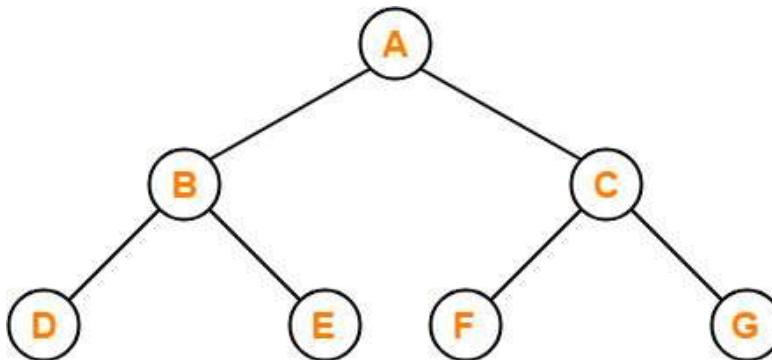
# Breadth-First Traversal



H	D	L	B	F	J	N	A	C	E	G	M	O
---	---	---	---	---	---	---	---	---	---	---	---	---

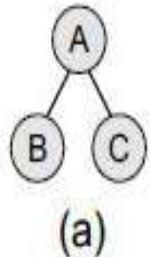
# Breadth First Traversal

- Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- Breadth First Traversal is also called as **Level Order Traversal**.
- All the nodes at a level are accessed before going to the next level.

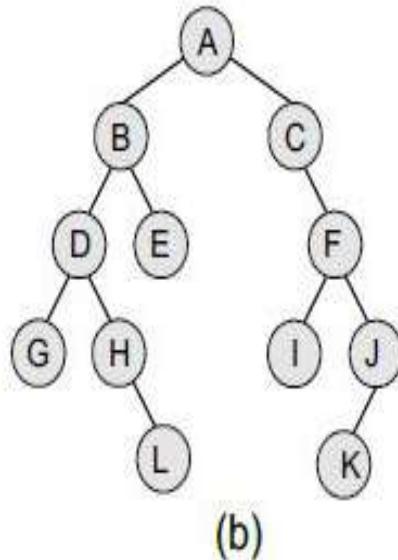


Level Order Traversal : A , B , C , D , E , F , G

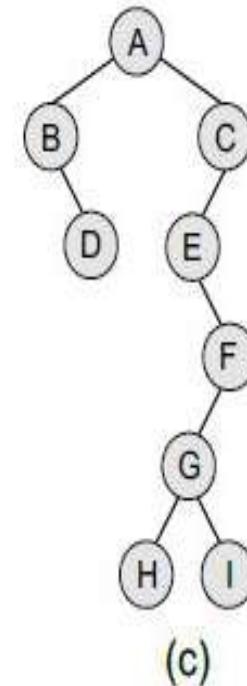
# Breadth First Traversal



(a)



(b)



(c)

TRAVERSAL ORDER:  
A, B, and C

TRAVERSAL ORDER:  
A, B, C, D, E, F, G, H, I, J, L, and K

TRAVERSAL ORDER:  
A, B, C, D, E, F, G, H, and I

# Depth First Traversal

Starting from top, Left to right

1 -> 12 -> 5 -> 6 -> 9

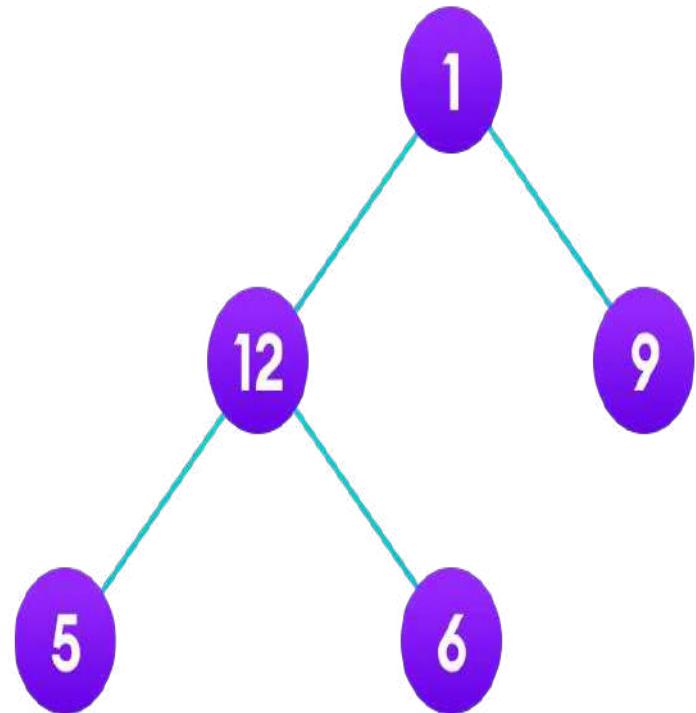
Starting from bottom, Left to right

5 -> 6 -> 12 -> 9 -> 1

Although this process is somewhat easy, it doesn't respect the hierarchy of the tree, only the depth of the nodes.

Instead, we use traversal methods that take into account the basic structure of a tree i.e.

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



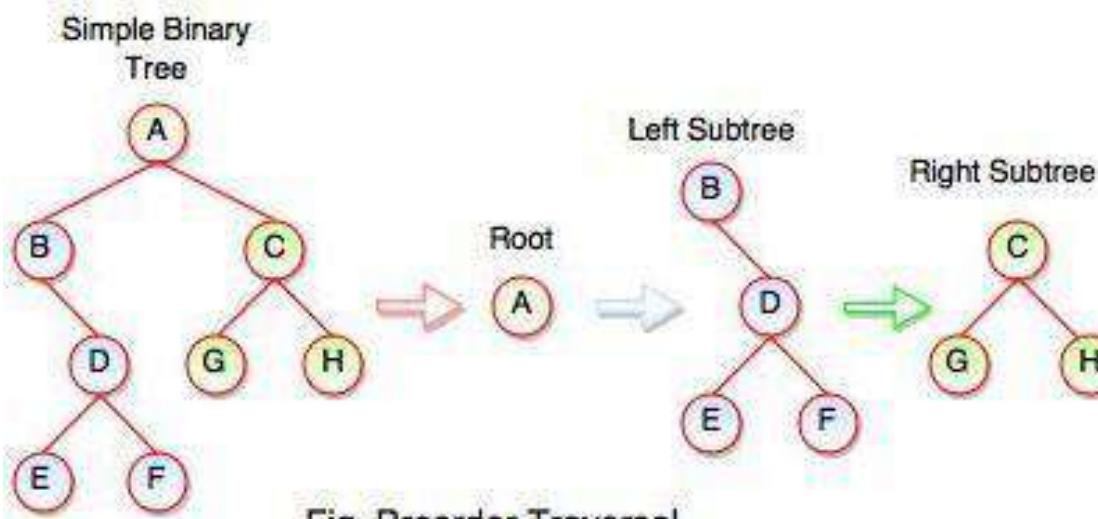
# Depth First Traversal

Following three traversal techniques fall under Depth First Traversal-

- Preorder Traversal
- Inorder Traversal
- Postorder Traversal

# Binary Search Tree

- Preorder Traversal:
- **Algorithm for preorder traversal**
  - Step 1 : Start from the Root.
  - Step 2 : Then, go to the Left Subtree.
  - Step 3 : Then, go to the Right Subtree.



$A + B + D + E + F + C + G + H$

# Binary Search

## Tree

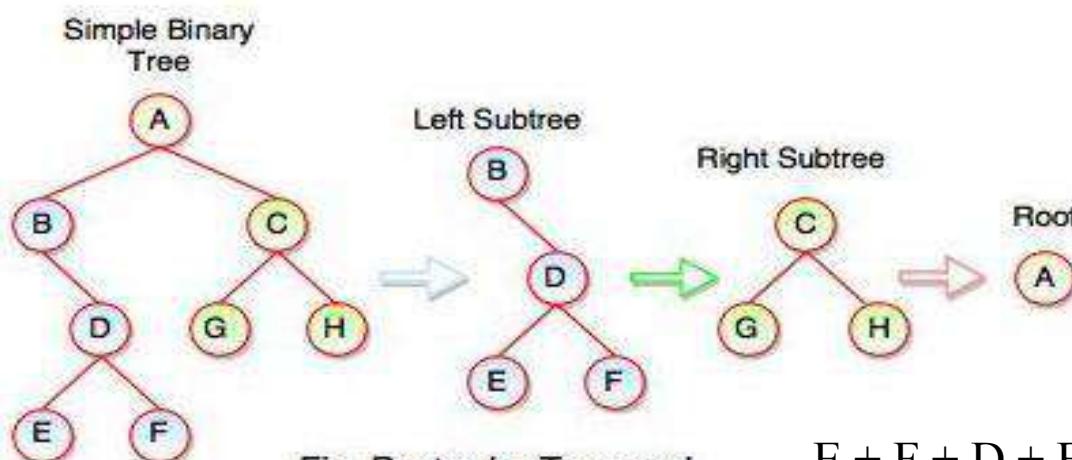
- Postorder Traversal

- Algorithm for postorder traversal

**Step 1 :** Start from the Left Subtree (Last Leaf).

**Step 2 :** Then, go to the Right Subtree.

**Step 3 :** Then, go to the Root.



$E + F + D + B + G + H + C + A$

# Binary Search

## Tree

- Inorder Traversal:

- **Algorithm for inorder traversal**

**Step 1 :** Start from the Left Subtree.

**Step 2 :** Then, visit the Root.

**Step 3 :** Then, go to the Right Subtree.

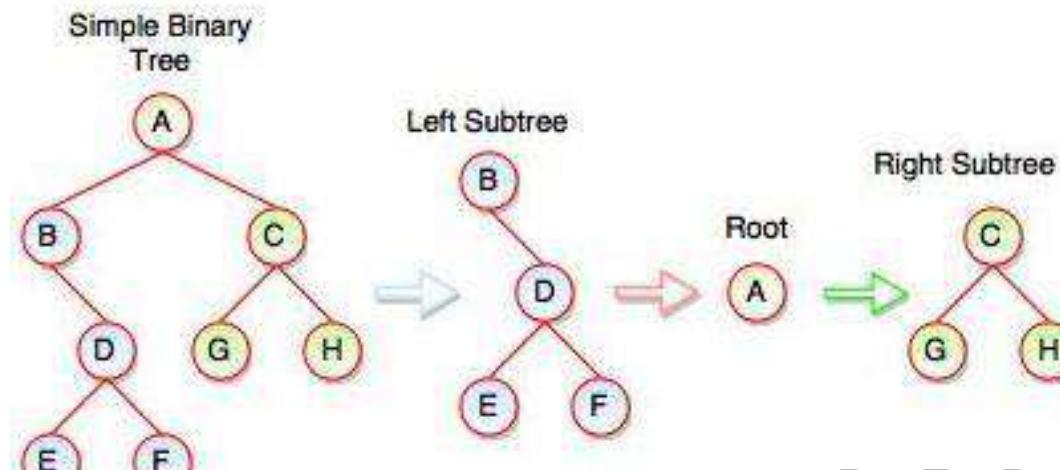


Fig. Inorder Traversal

B + E + D + F + A + G + C + H

# Depth-First Traversals

## Preorder (N-L-R)

### Steps

1. Visit the node (N)
2. Visit the left subtree (L), if any.
3. Visit the right subtree (R), if any.

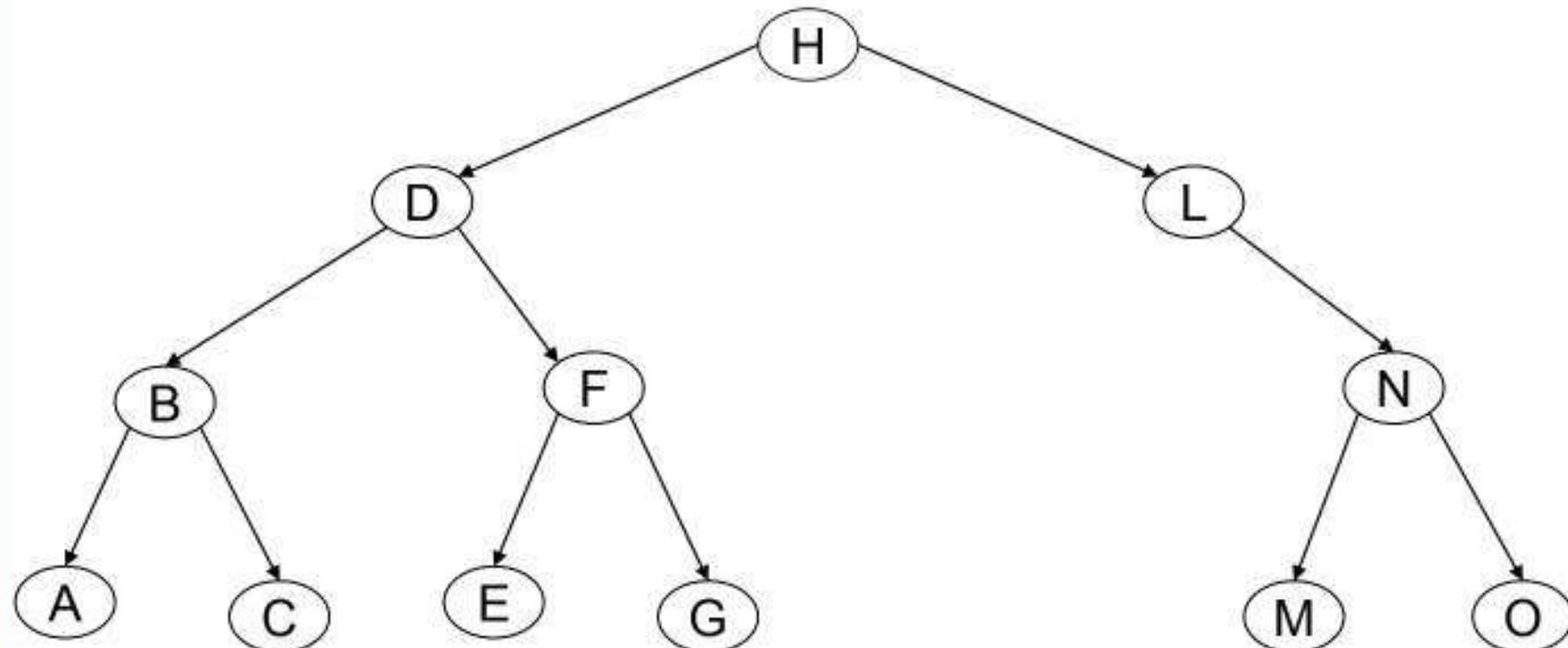
## Preorder (N-L-R)

### Recursive Pseudocode

```
Function preorder( root)
  if(root != NULL)
    print "root->data"
    preorder(root->left);
    preorder(root->right);
  endif
```

# Depth-first Traversal

Preorder N-L-R



H	D	B	A	C	F	E	G	L	N	M	O
---	---	---	---	---	---	---	---	---	---	---	---



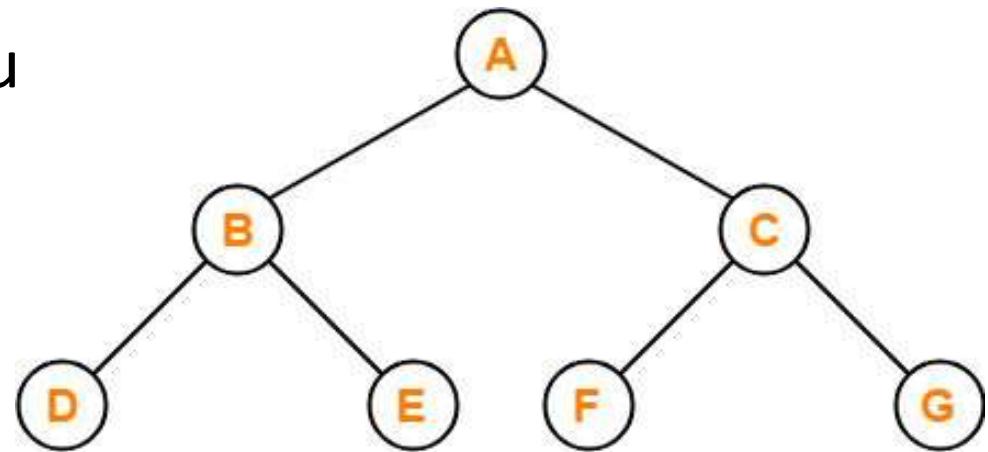
SOMAIYA  
VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

Somaiya  
TRUST

# Preorder Traversal

- Visit root node
- Visit all the nodes in the left subtree
- Visit all the nodes in the right subtree
- **Root → Left → Right**
- `display(root->data)`
- `preorder(root->left)`
- `preorder(root->right)`



Preorder Traversal : A , B , D , E , C , F , G

# Preorder Traversal

## Algorithm for pre-order traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:           Write TREE->DATA
Step 3:           PREORDER(TREE-> LEFT)
Step 4:           PREORDER(TREE-> RIGHT)
                  [END OF LOOP]
Step 5: END
```

Pre-order traversal is also called as *depth-first traversal*.

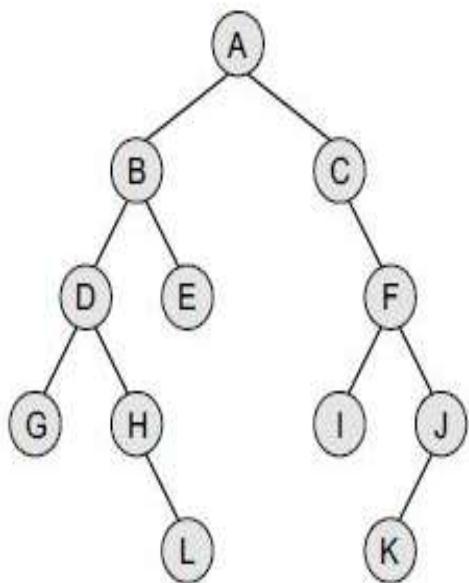
Pre-order algorithm is also known as the NLR traversal algorithm (Node-Left-Right).

# Preorder Traversal

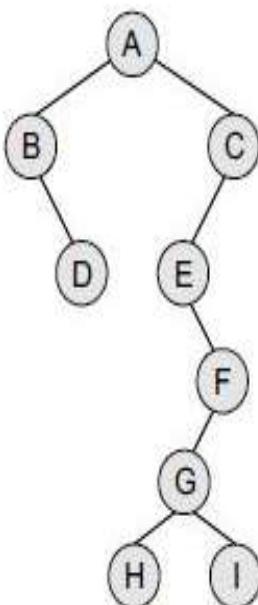
```
void preorder(struct node *tree)
{
    if(tree!=NULL)
    {
        printf("%d\n",tree->num);
        preorder(tree->left);
        preorder(tree->right);
    }
}
```

# Preorder Traversal

- In Figs (a) and (b), find the sequence of nodes that will be visited using pre-order traversal algorithm.



(a)



(b)

## ***Solution***

(a) TRAVERSAL ORDER:  
A, B, D, G, H, L, E, C, F, I, J,  
and K

(b) TRAVERSAL ORDER:  
A, B, D, C, E, F, G, H, and I

# Depth-First Traversals

## Inorder (L-N-R)

### Steps

1. Visit the left subtree (L), if any.
2. Visit the node (N)
3. Visit the right subtree (R), if any.

## Inorder (L-N-R)

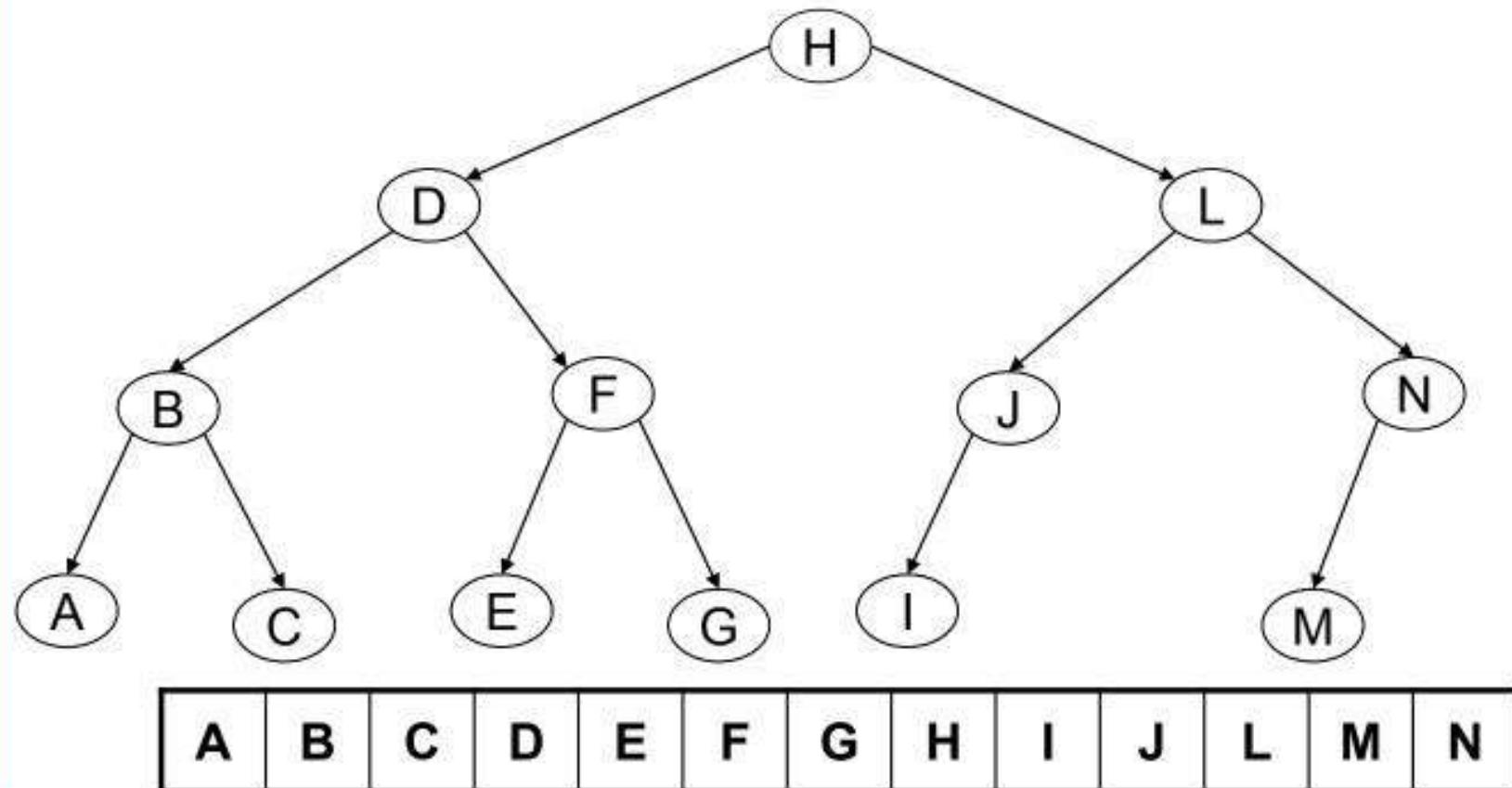
### Recursive Pseudocode

```
Function inorder( root)
  if(root != NULL)
    inorder(root->left);
    print "root->data"
    inorder(root->right);
  endif
```



# Depth-First Traversal

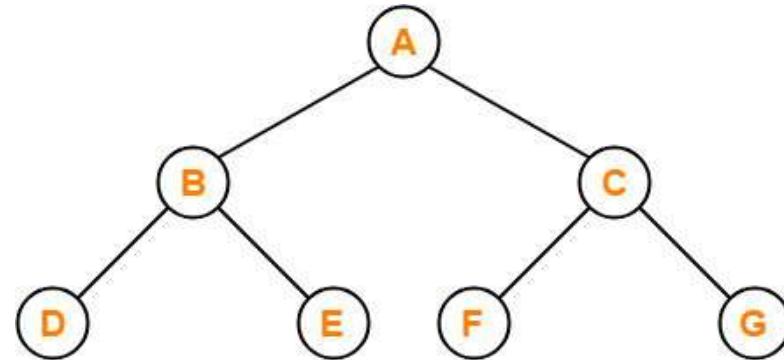
Inorder L-N-R



**Note: An inorder traversal of a BST visits the keys sorted in increasing order.**

# Inorder Traversal

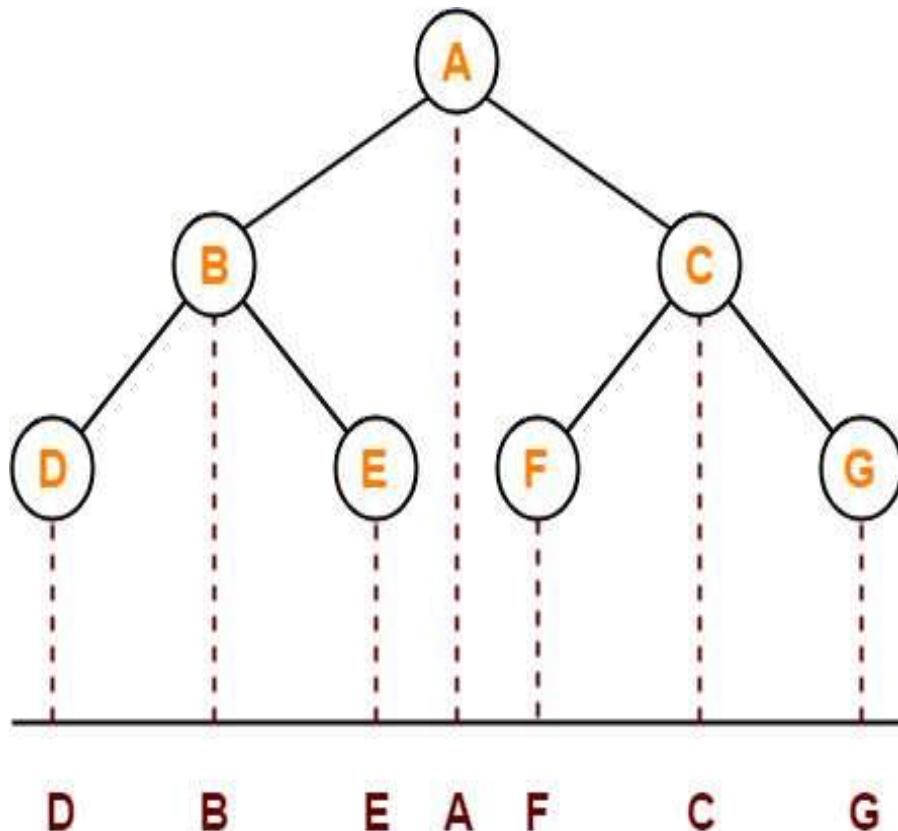
- **Inorder traversal**
- First, visit all the nodes in the left subtree
- Then the root node
- Visit all the nodes in the right subtree
- **Left → Root → Right**
- `inorder(root->left)`
- `display(root->data)`
- `inorder(root->right)`



Inorder Traversal : D , B , E , A , F , C , G

# Inorder Traversal

Keep a plane mirror horizontally at the bottom of the tree and take the projection of all the nodes.



Inorder Traversal : D , B , E , A , F , C , G

# Inorder Traversal

- Algorithm for Inorder Traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:          INORDER(TREE->LEFT)
Step 3:          Write TREE->DATA
Step 4:          INORDER(TREE->RIGHT)
               [END OF LOOP]
Step 5: END
```

In-order traversal is also called as *symmetric traversal*.

In-order algorithm is also known as the LNR traversal algorithm (Left-Node-Right).

# Inorder Traversal

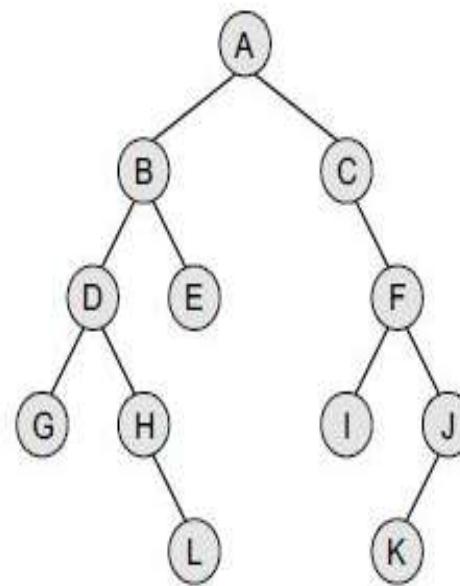
```
void inorder(struct node *tree)
{
    if(tree!=NULL)
    {
        inorder(tree->left);
        printf("%d\n",tree->num);
        inorder(tree->right);
    }
}
```

# Inorder Traversal

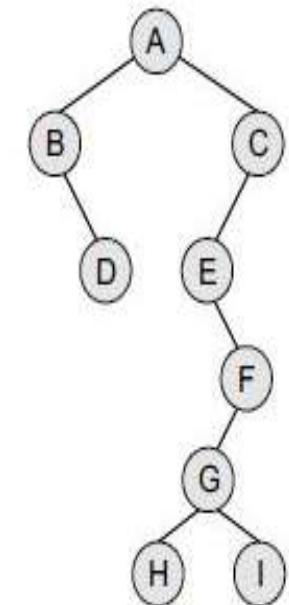
- In-order traversal algorithm is usually used to display the elements of a binary search tree.
- Here, all the elements with a value lower than a given value are accessed before the elements with a higher value.
- We will discuss binary search trees in detail

# Inorder Traversal

- For the trees given find the sequence of nodes that will be visited using in-order traversal alg



(a)

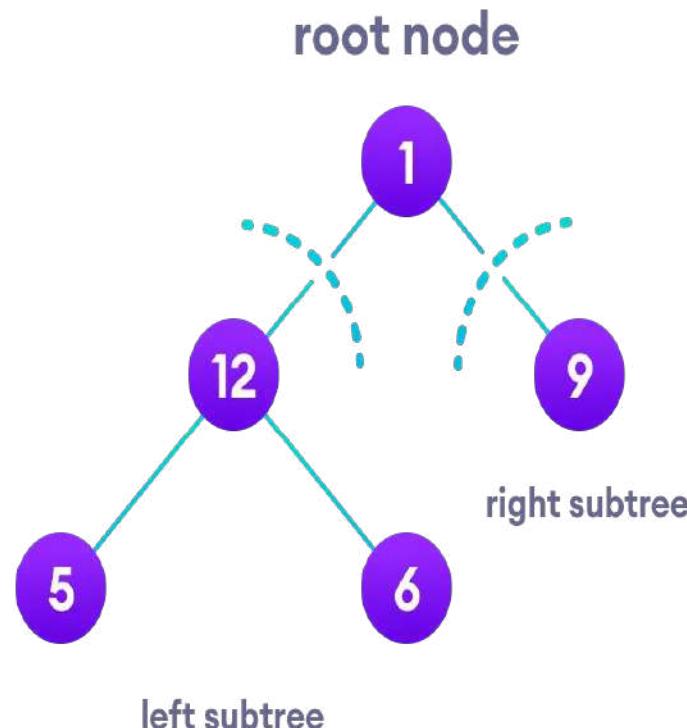


(b)

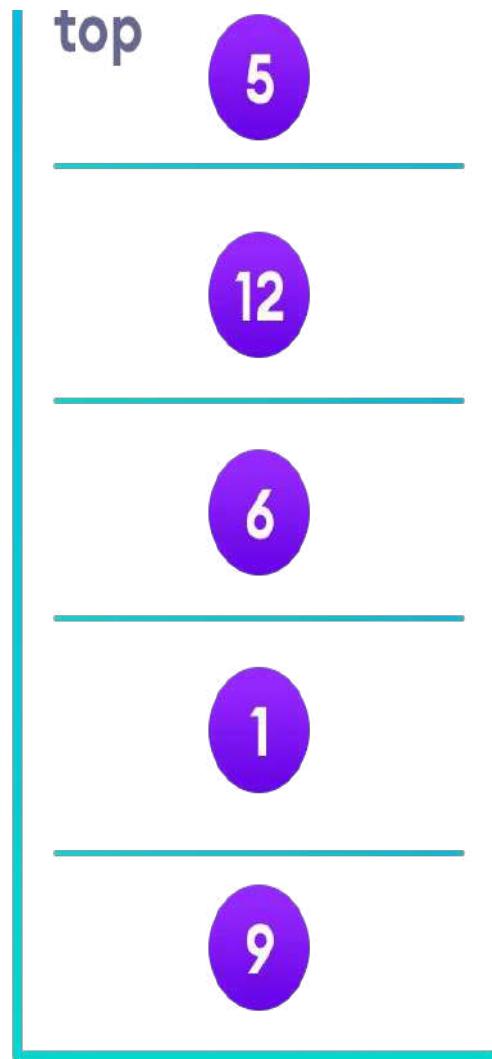
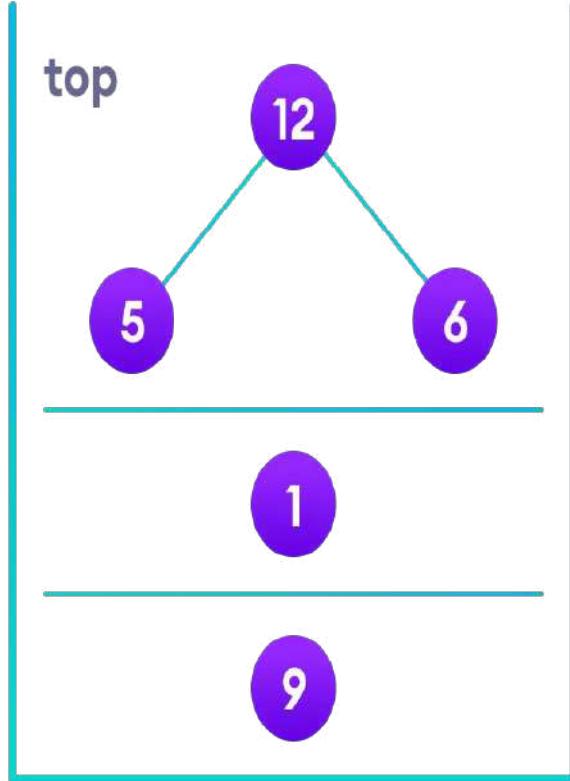
- TRAVERSAL ORDER: G, D, H, L, B, E, A, C, I, F, K, and J
- TRAVERSAL ORDER: B, D, A, E, H, G, I, F, and C

# Inorder Traversal

- Let's visualize in-order traversal. We start from the root node.
- We traverse the left subtree first. We also need to remember to visit the root node and the right subtree when this tree is done.



# Inorder Traversal



- Let's put all this in a stack so that we remember.
- We traverse left subtree.
  - Since the node "5" doesn't have any subtrees, we print it directly.
  - After that we print its parent "12" and then the right child "6".
- Root node print
- We traverse right subtree.
- We don't have to create the stack ourselves because recursion maintains the correct order for us.

# Depth-First Traversals

## Postorder (L-R-N)

### Steps

1. Visit the left subtree (L), if any.
2. Visit the right subtree (R), if any.
3. Visit the node (N)

## Postorder (L-R-N)

### Recursive Pseudocode

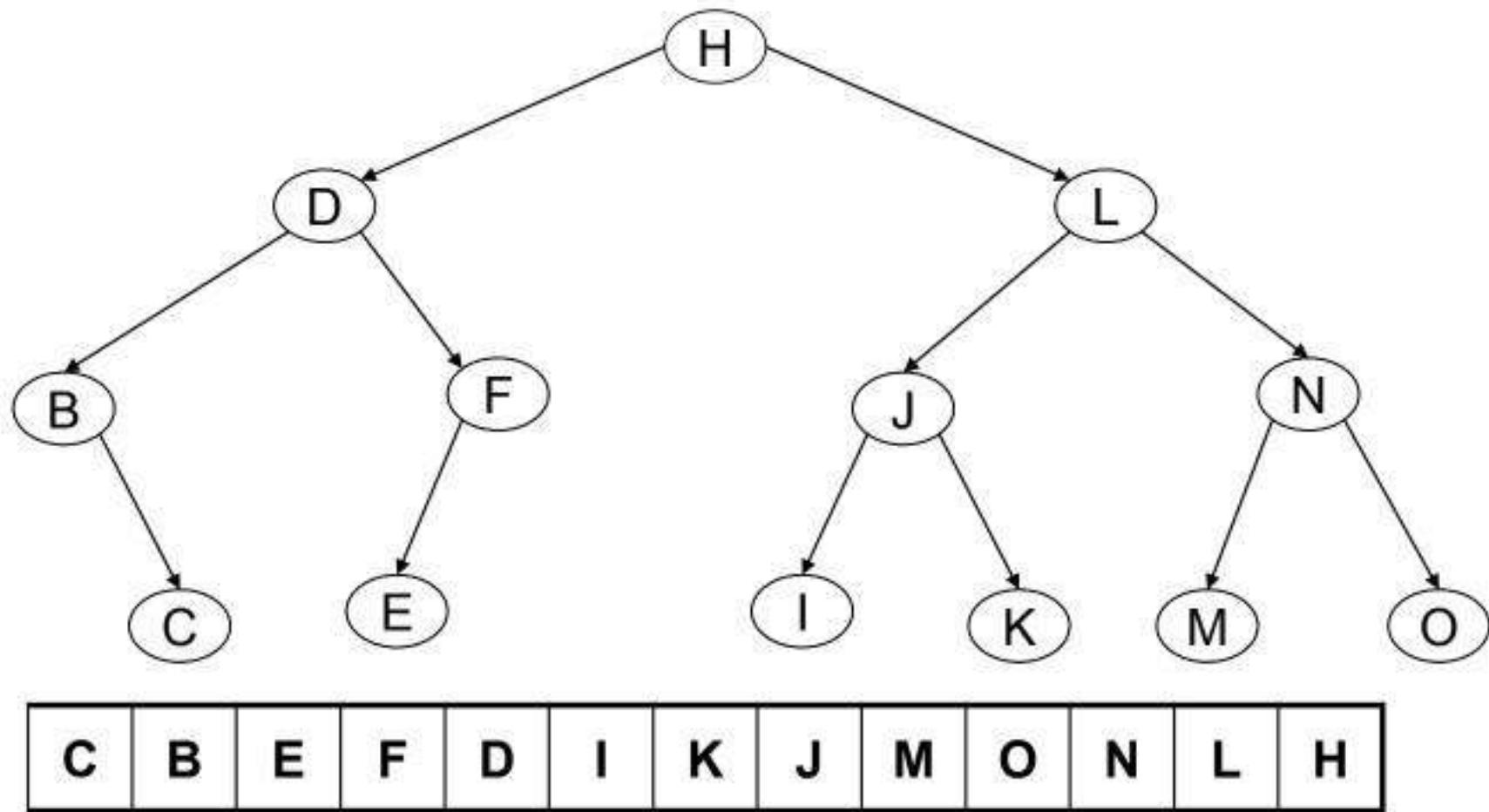
```
Function postorder( root)
    if(root != NULL)
        postorder(root->left);
        postorder(root->right);
        print "root->data"
    endif
```

### Code:

[https://docs.google.com/document/d/11SZBG-ZefpKbYaEW08SIX6cLv3xbxDeZGMCSoln\\_GaU/edit?usp=sharing](https://docs.google.com/document/d/11SZBG-ZefpKbYaEW08SIX6cLv3xbxDeZGMCSoln_GaU/edit?usp=sharing)

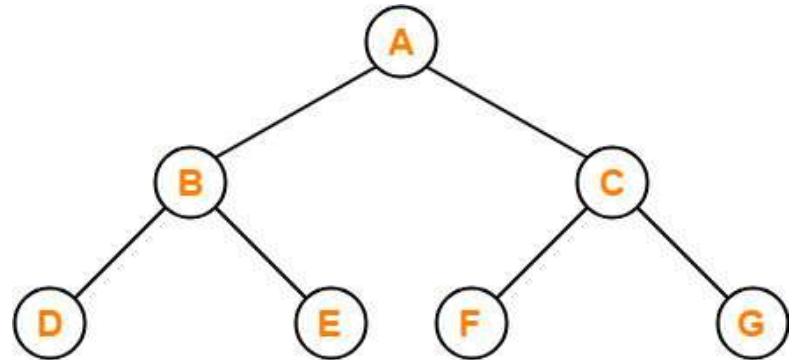
# Depth-First Traversal

Postorder L-R-N



# Postorder Traversal

- Visit all the nodes in the left subtree
- Visit all the nodes in the right subtree
- Visit the root node
- **Left → Right → Root**
- `postorder(root->left)`
- `postorder(root->right)`
- `display(root->data)`



Postorder Traversal : D , E , B , F , G , C , A

# Post-order Traversal

- Algorithm for post-order traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:           POSTORDER(TREE -> LEFT)
Step 3:           POSTORDER(TREE -> RIGHT)
Step 4:           Write TREE -> DATA
                  [END OF LOOP]
Step 5: END
```

Post-order algorithm is also known as the LRN traversal algorithm (Left-Right-Node).

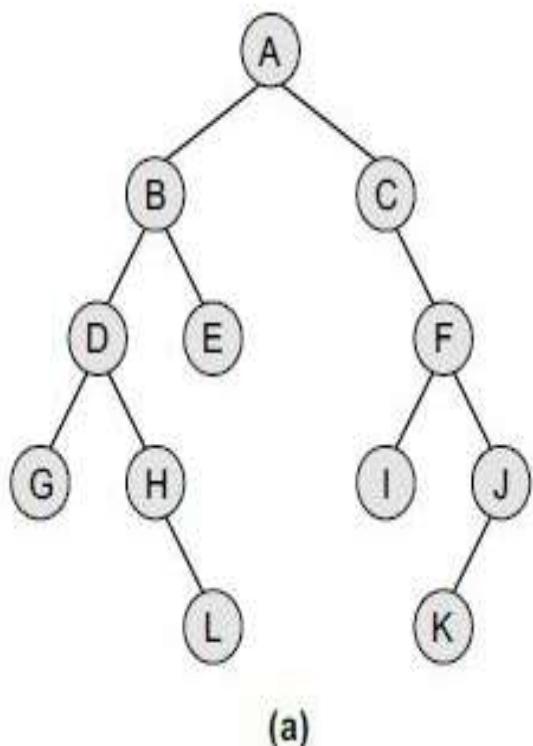
Post-order traversals are used to extract postfix notation from an expression tree.

# Post-order Traversal

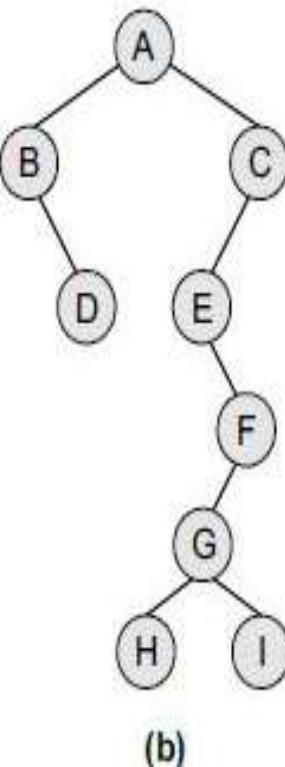
```
void postorder(struct node *tree)
{
    if(tree!=NULL)
    {
        postorder(tree->left);
        postorder(tree->right);
        printf("%d\n",tree->num);
    }
}
```

# Post-order Traversal

- For the trees given, write the sequence of nodes that will be visited using post-order traversal algorithm.



(a)



(b)

TRAVERSAL ORDER:  
G, L, H, D, E, B, I, K, J, F, C,  
and A

TRAVERSAL ORDER:  
D, B, H, I, G, F, E, C, and A

# Constructing a Binary Tree from Traversal Results

- We can construct a binary tree if we are given at least two traversal results.
- The first traversal must be the **in-order traversal** and the second can be **either pre-order or post-order traversal**.
- In-order traversal result can be used to determine the **left and the right child nodes**, & the pre-order/post-order can be used to determine **the root node**.

# Steps to Construct a Tree from Traversal Results

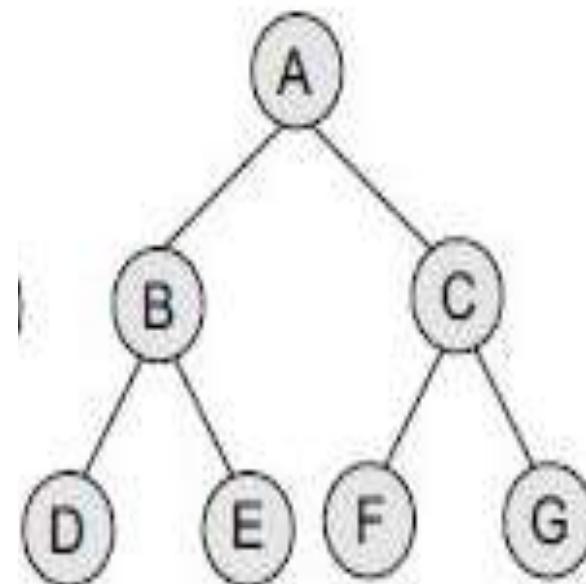
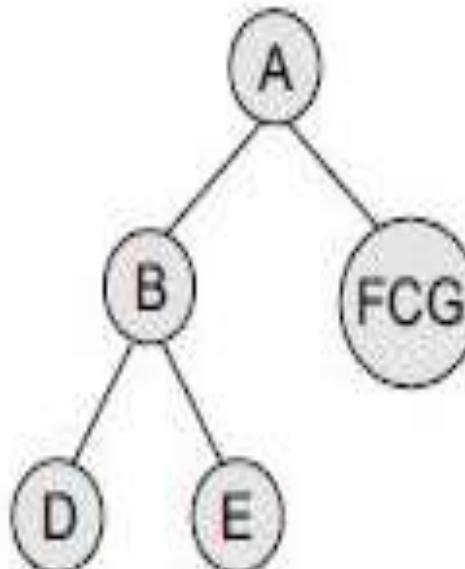
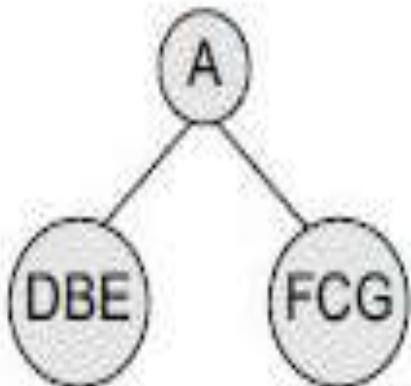
- **Step 1** Use the pre-order sequence to determine the root node of the tree. The first element would be the root node.
- **Step 2** Elements on the left side of the root node in the in-order traversal sequence form the left sub-tree of the root node. Similarly, elements on the right side of the root node in the in-order traversal sequence form the right sub-tree of the root node.
- **Step 3** Recursively select each element from pre-order traversal sequence and create its left and right sub-trees from the in-order traversal sequence.

# Constructing a Binary Tree from Traversal Results

Consider the traversal results given below:

In-order Traversal: D B E A F C G

Pre-order Traversal: A B D E C F G



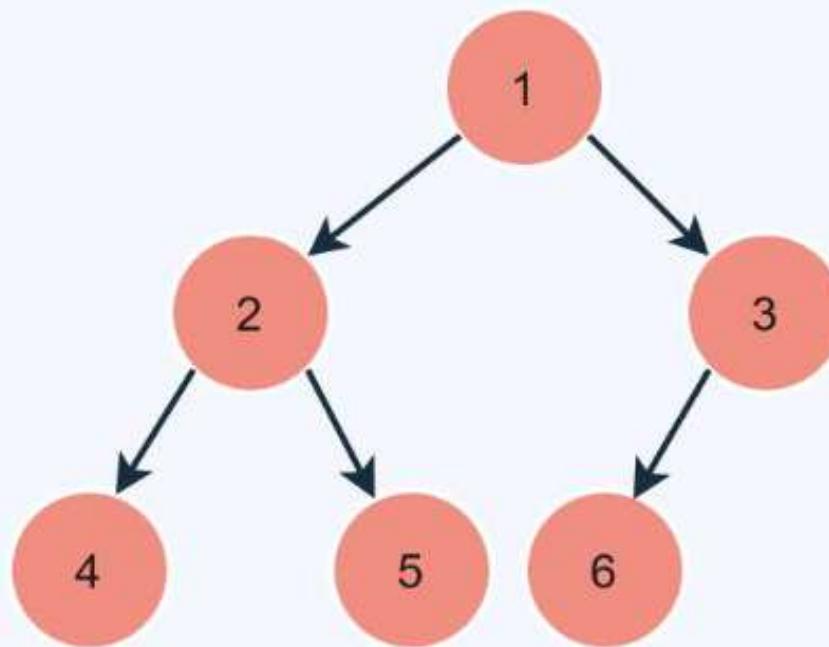
Input

Post-order Traversal: 4 5 2 6 3 1

In-order Traversal: 4 2 5 1 6 3

Output

Binary Tree:



# BST Examples

Write all traversal sequence for each of the following BST

1. 10, 12, 34, 56, 71, 87
2. 50, 43, 37, 23, 21, 11, 9, 4, 30
3. 56, 78, 34, 23, 0, 26, 57, 68, 9, 36, 44, 30

# Constructing BST from given Traversal Sequences

Case 1 : Inorder and Preorder

Case 2 : Inorder and Postorder

Case 3 : Postorder and Preorder

**The preorder traversal sequence of a binary search tree is 30, 20, 10, 15, 25, 23, 39, 35, 42. Which one of the following is the postorder traversal sequence of the same tree?**

1. 10, 20, 15, 23, 25, 35, 42, 39, 30
2. 15, 10, 25, 23, 20, 42, 35, 39, 30
3. 15, 20, 10, 23, 25, 42, 35, 39, 30
4. 15, 10, 23, 25, 20, 35, 42, 39, 30

## EXPLANATION:

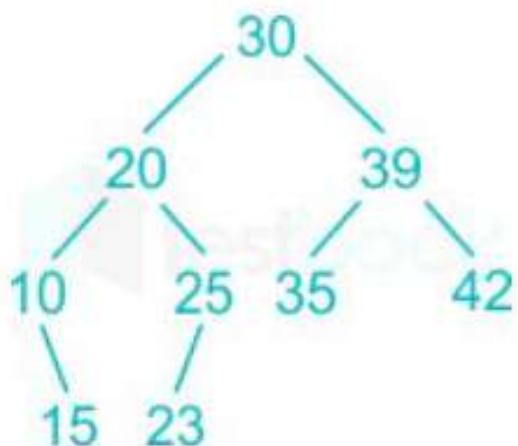
The **pre-order traversal** of given BST is:

30, 20, 10, 15, 25, 23, 39, 35, 42.

So, the **In-order traversal** of the BST is:

10, 15, 20, 23, 25, 30, 35, 39, 42.

The **Binary Search Tree** is:



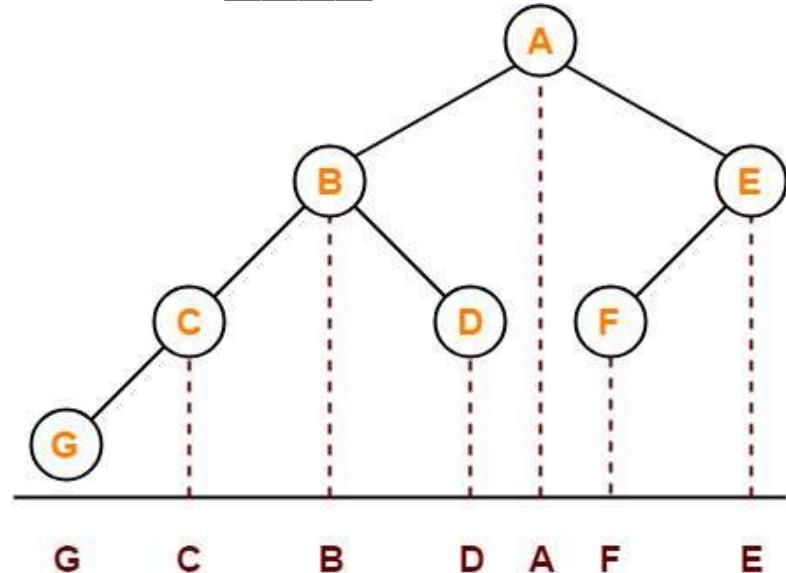
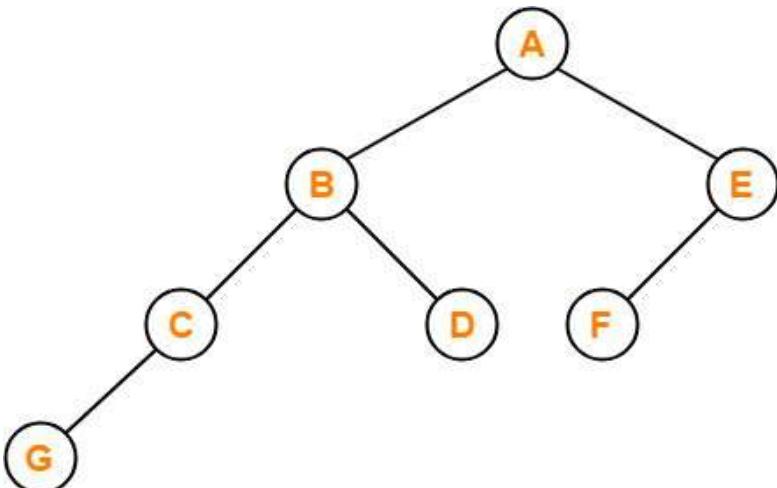
So the post-order traversal of the tree is:

15, 10, 23, 25, 20, 35, 42, 39, 30

Hence, the correct answer is "option 4".

# PRACTICE PROBLEMS BASED ON TREE TRAVERSAL

- If the binary tree in figure is traversed in inorder, then the order in which the nodes will be visited is \_\_\_\_?

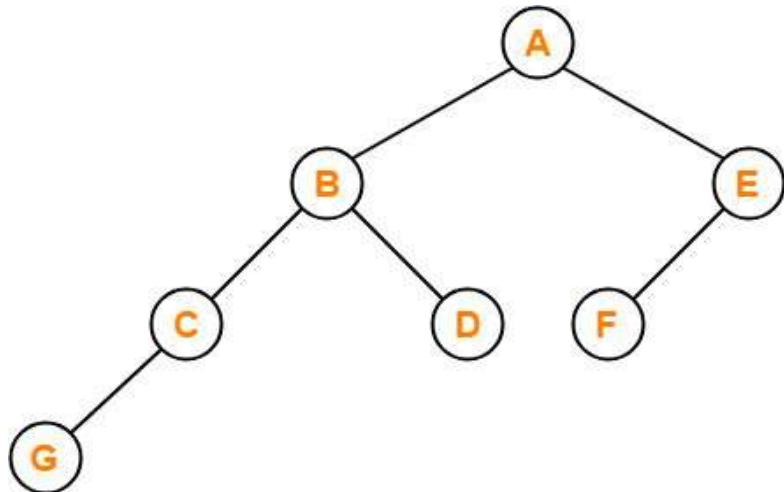


Inorder Traversal : G , C , B , D , A , F , E

# PRACTICE PROBLEMS BASED ON TREE TRAVERSAL

- Which of the following sequences denotes the postorder traversal sequence of the tree shown in figure?

FEGCBDBA  
GCBDAFE  
GCDBFEA  
FDEGCBA

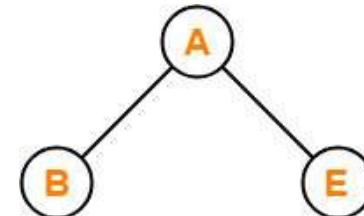


Postorder Traversal : G , C , D , B , F , E , A

Thus, Option (C) is correct.

# PRACTICE PROBLEMS BASED ON TREE TRAVERSAL

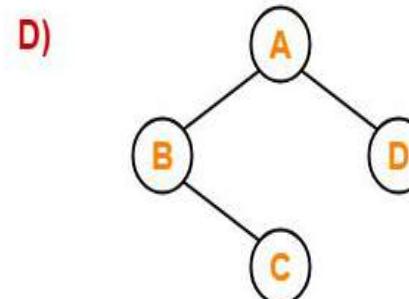
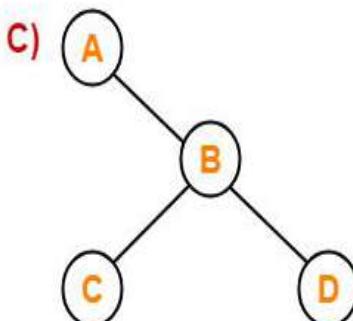
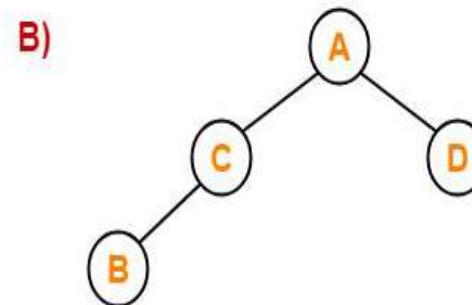
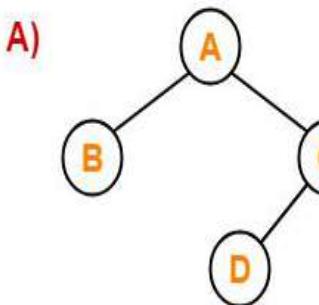
- Let LASTPOST, LASTIN, LASTPRE denote the last vertex visited in a postorder, inorder and preorder traversal respectively of a complete binary tree. Which of the following is always true?
  - A) LASTIN = LASTPOST
  - B) LASTIN = LASTPRE
  - C) LASTPRE = LASTPOST
  - D) None of these
- Consider the following complete binary tree-
- Preorder Traversal : A , B, E
- Inorder Traversal : B , A , E
- Postorder Traversal : B , E , A
- Clearly, LASTIN = LASTPRE. Thus, Option (B) is correct.



# PRACTICE PROBLEMS BASED ON TREE TRAVERSAL

- Which of the following binary trees has its inorder and preorder traversals as BCAD and ABCD respectively

- Solution D



# Applications of Trees

- **Binary search trees**
  - A simple data structure for sorted lists
- **Decision trees**
  - Minimum comparisons in sorting algorithms
- **Prefix codes**
  - Huffman coding

# Applications of Trees

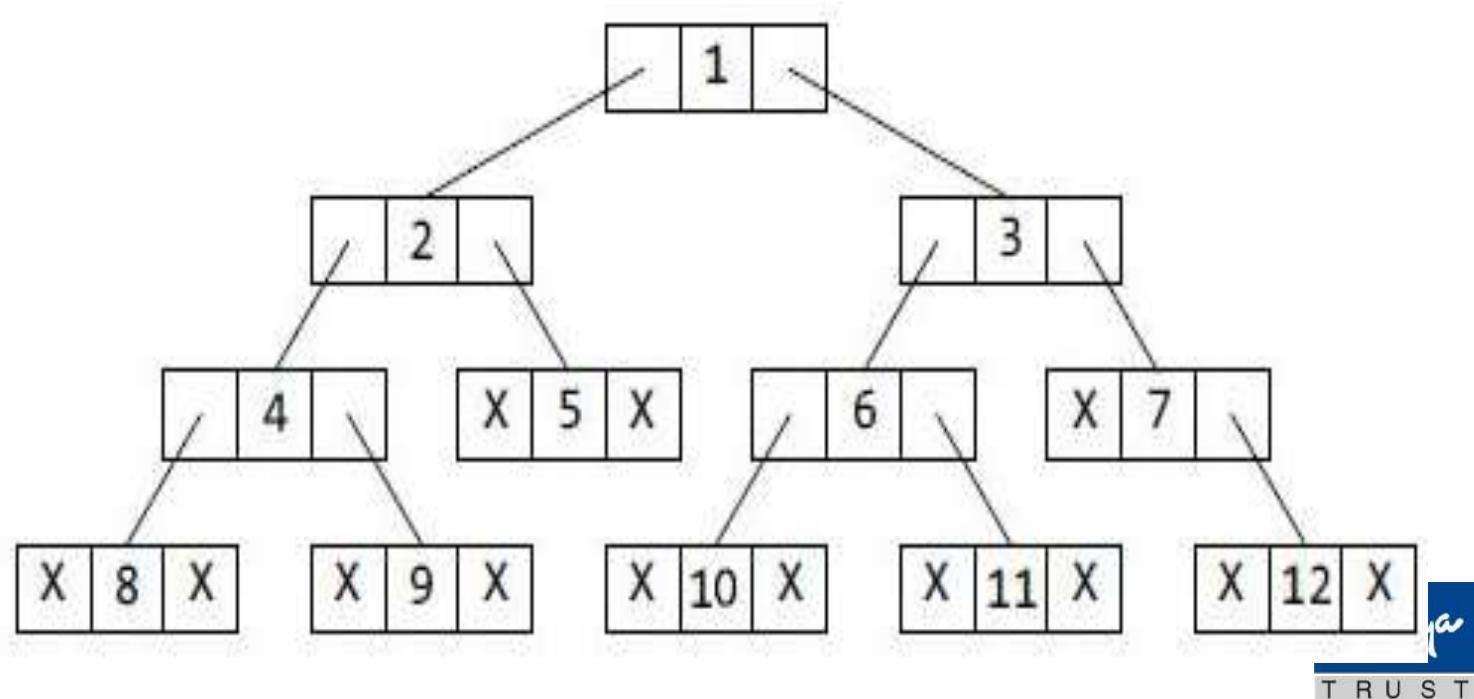
Trees are very important data structures in computing.

They are suitable for:

- Hierarchical structure representation, e.g.,
  - File directory.
  - Organizational structure of an institution.
  - Class inheritance tree.
- Problem representation, e.g.,
  - Expression tree.
  - Decision tree.
- Efficient algorithmic solutions, e.g.,
  - Search trees.
  - Efficient priority queues via heaps.

# THREADED BINARY TREES

- A threaded binary tree is the **same as that of a binary tree** but with a **difference in storing the NULL pointers**.
- Consider the linked representation of a binary tree (without threading) as given in Fig.
- *The in-order traversal of the tree is given as 8, 4, 9, 2, 5, 1, 10, 6, 11, 3, 7, 12*



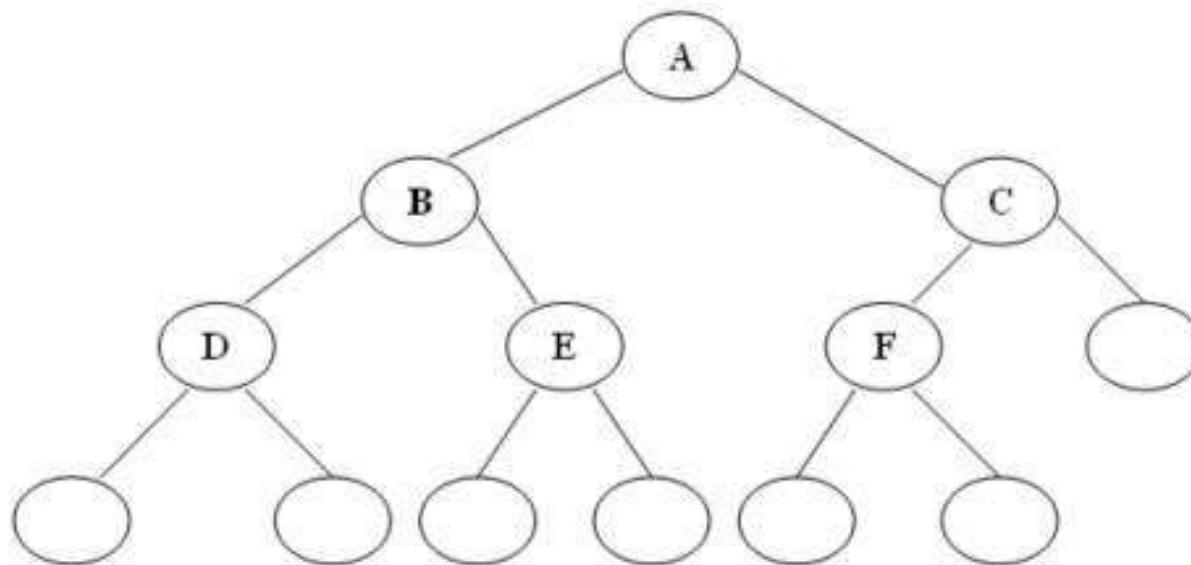
# THREADED BINARY TREES

- In the linked representation, a number of nodes contain a **NULL pointer**, either in their left or right fields or in both.
- The **space is wasted** in storing a NULL pointer can be efficiently used to store some other useful piece of information.
- For example, the NULL entries can be replaced to store a pointer to the in-order predecessor or the in-order successor of the node. These **special pointers are called *threads*** and **binary trees containing threads are called *threaded trees***.
- In the linked representation of a threaded binary tree, threads will be **denoted using arrows**.
- A threaded binary tree may correspond to **one-way threading** or a **two way threading**.

# Threaded Binary Tree

In a linked representation of a binary tree, the number of null links (null pointers) are actually more than non-null pointers.

Consider the following binary tree:



A Binary tree with the null pointers

# Threaded Binary Tree

In above binary tree, there are 7 null pointers & actual 5 pointers.

In all there are 12 pointers.

We can generalize it that for any binary tree with  $n$  nodes there will be  $(n+1)$  null pointers and  $2n$  total pointers.

The objective here to make effective use of these null pointers.

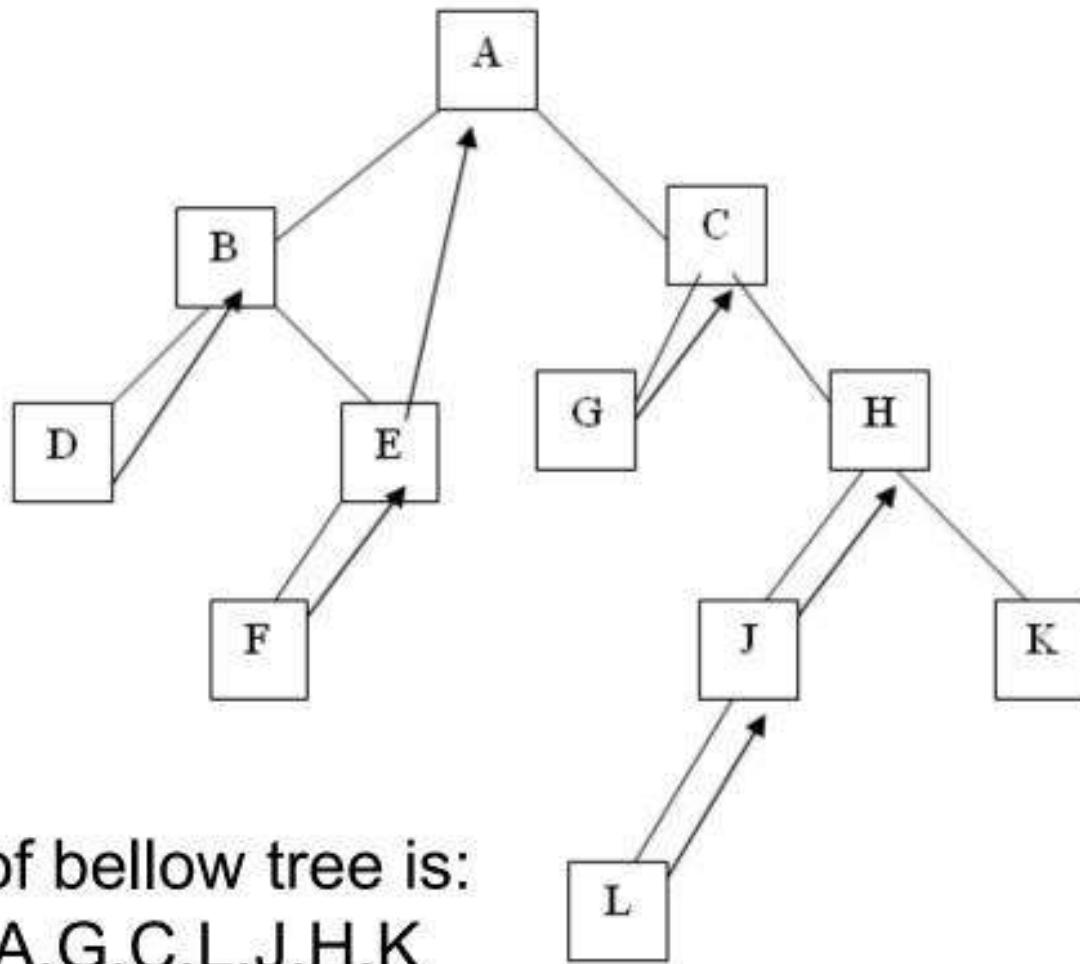
A. J. perils & C. Thornton jointly proposed idea to make effective use of these null pointers.

According to this idea we are going to replace all the null pointers by the appropriate pointer values called threads.

# THREADED BINARY TREES

- In **one-way threading**, a **thread will appear** either in the right field or the left field of the node. A one-way threaded tree is also called a **single-threaded tree**.
- If the thread appears in the left field, then the left field will be made to point to the **in-order predecessor of the node**. Such a one-way threaded tree is called a **left-threaded binary tree**.
- On the contrary, if the thread appears in the right field, then it will point to the **in-order successor** of the node. Such a one-way threaded tree is called a **right threaded binary tree**.

# Threaded Binary Tree: One-Way

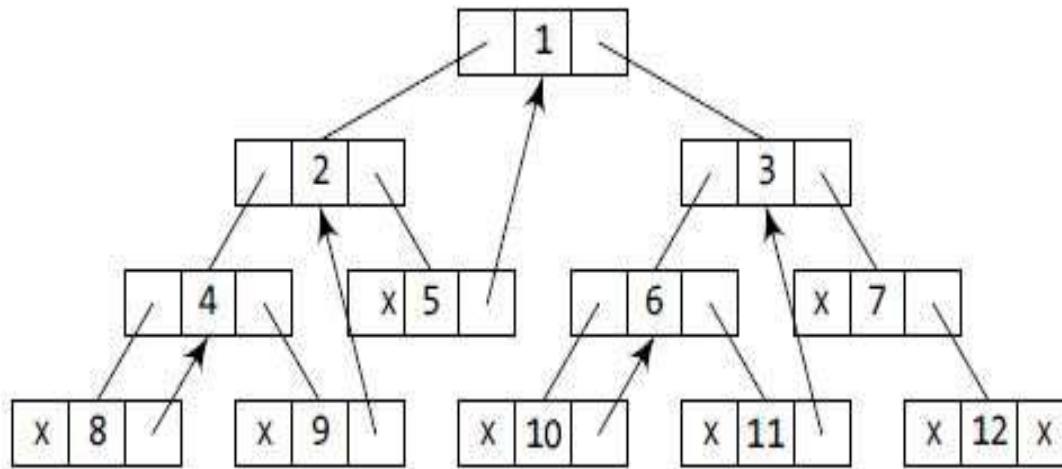


Inorder of bellow tree is:  
D,B,F,E,A,G,C,L,J,H,K

One-way inorder threading

# One-way Threading

- Node 5 contains a NULL pointer in its RIGHT field, so it will be replaced to point to node 1, which is its in-order successor. (8, 4, 9, 2, 5, 1, 10, 6, 11, 3, 7, 12)
- Similarly, the RIGHT field of node 8 will point to node 4, the RIGHT field of node 9 will point to node 2, the RIGHT field of node 10 will point to node 6, the RIGHT field of node 11 will point to node 3, and the RIGHT field of node 12 will contain NULL because it has no in-order successor.

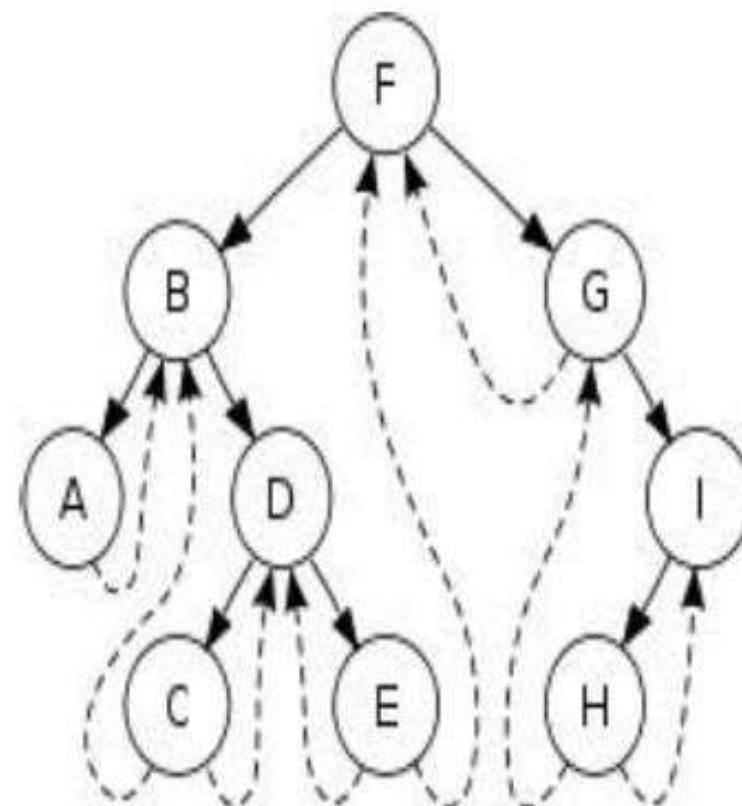


# THREADED BINARY TREES

- In a **two-way threaded tree**, also called a double-threaded tree, threads will appear in both the left and the right field of the node. While the left field will point to the **in-order predecessor of the node**, **the right field will point to its successor**. A two-way threaded binary tree is also called a **fully threaded binary tree**.
- Threading will correspond to in-order traversal of tree.

# Threaded Binary Tree

- A **threaded binary tree** defined as:
- "A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node"

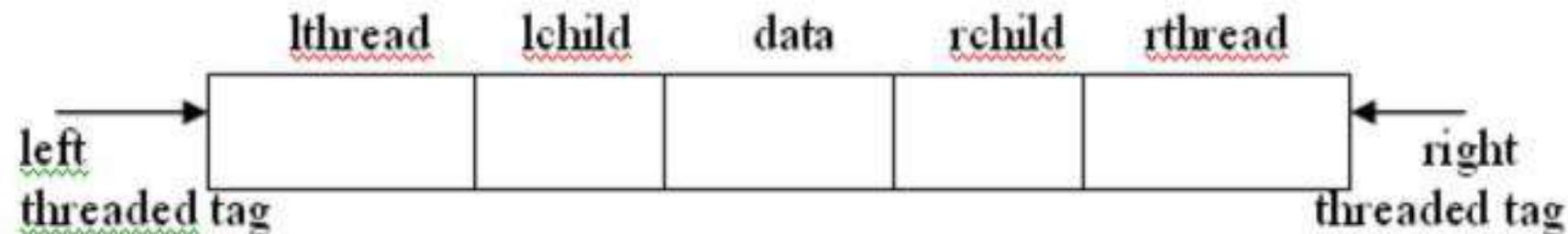


# Threaded Binary Tree

- And binary tree with such pointers are called threaded tree.
- In the memory representation of a threaded binary tree, it is necessary to distinguish between a normal pointer and a thread.

# Threaded Binary Tree

- Therefore we have an alternate node representation for a threaded binary tree which contains five fields as shown below:



For any node  $p$ , in a threaded binary tree.

$\text{lthred}(p)=1$  indicates  $\text{lchild}(p)$  is a thread pointer

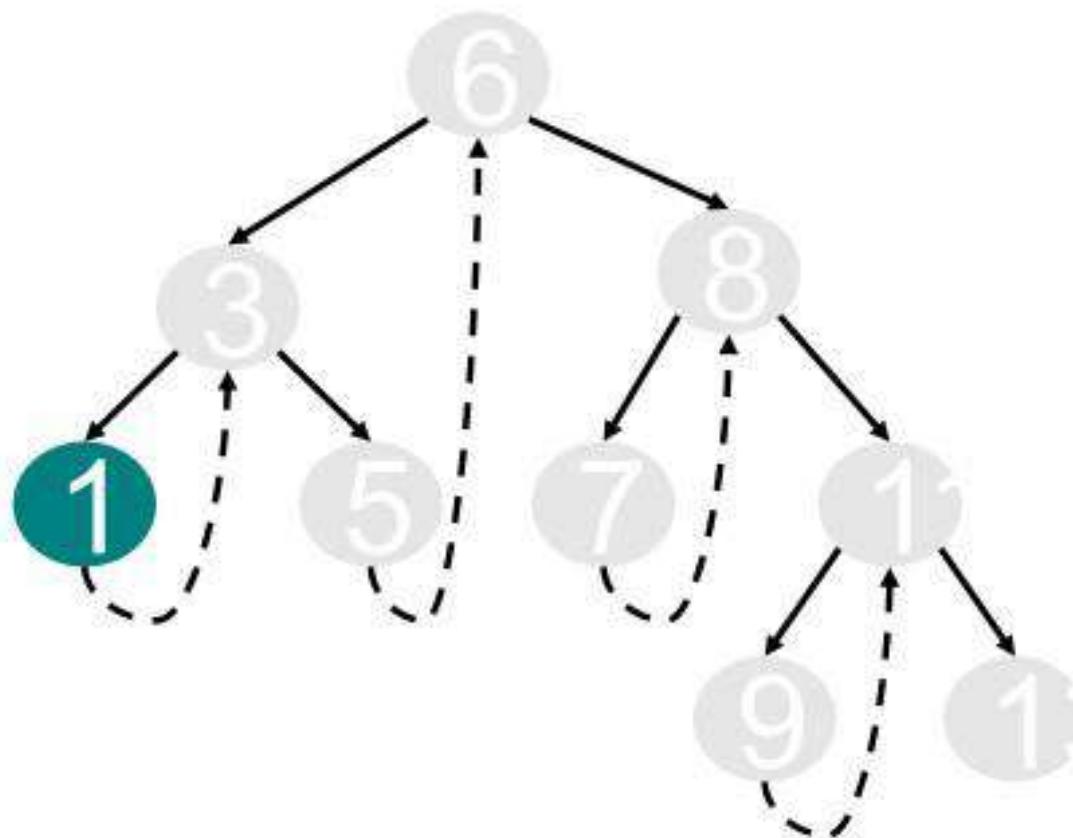
$\text{lthred}(p)=0$  indicates  $\text{lchild}(p)$  is a normal

$\text{rthred}(p)=1$  indicates  $\text{rchild}(p)$  is a thread

$\text{rthred}(p)=0$  indicates  $\text{rchild}(p)$  is a normal pointer



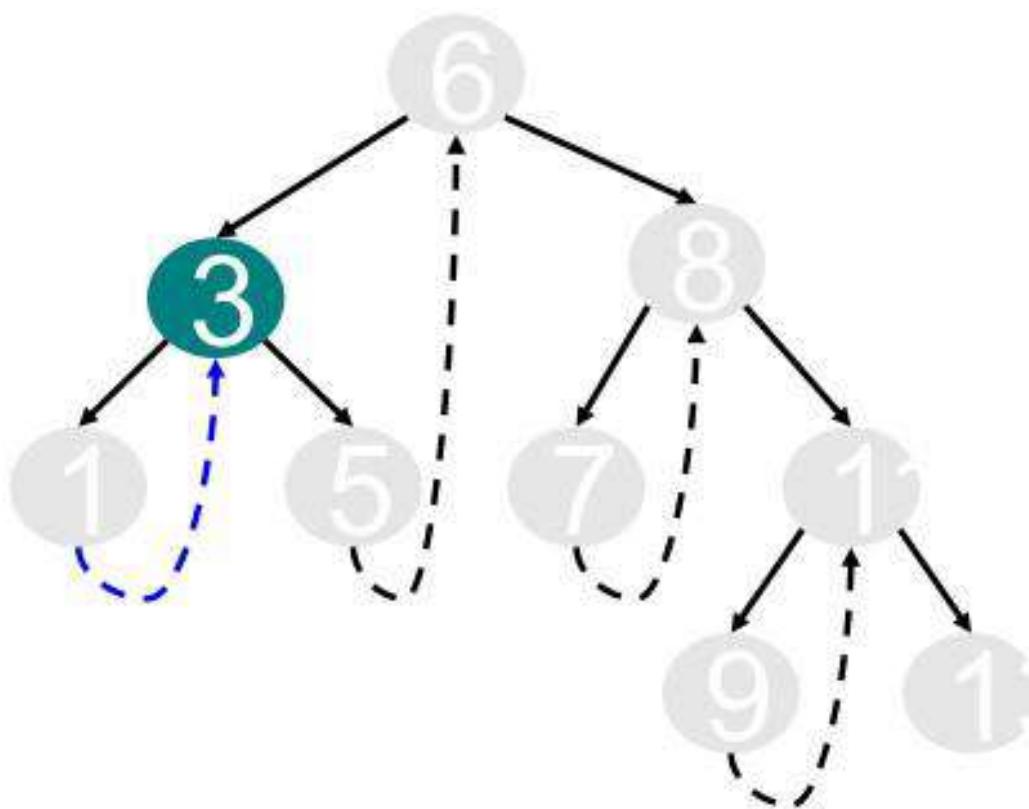
# Threaded Tree Traversal



Output  
1

Start at leftmost node, print it

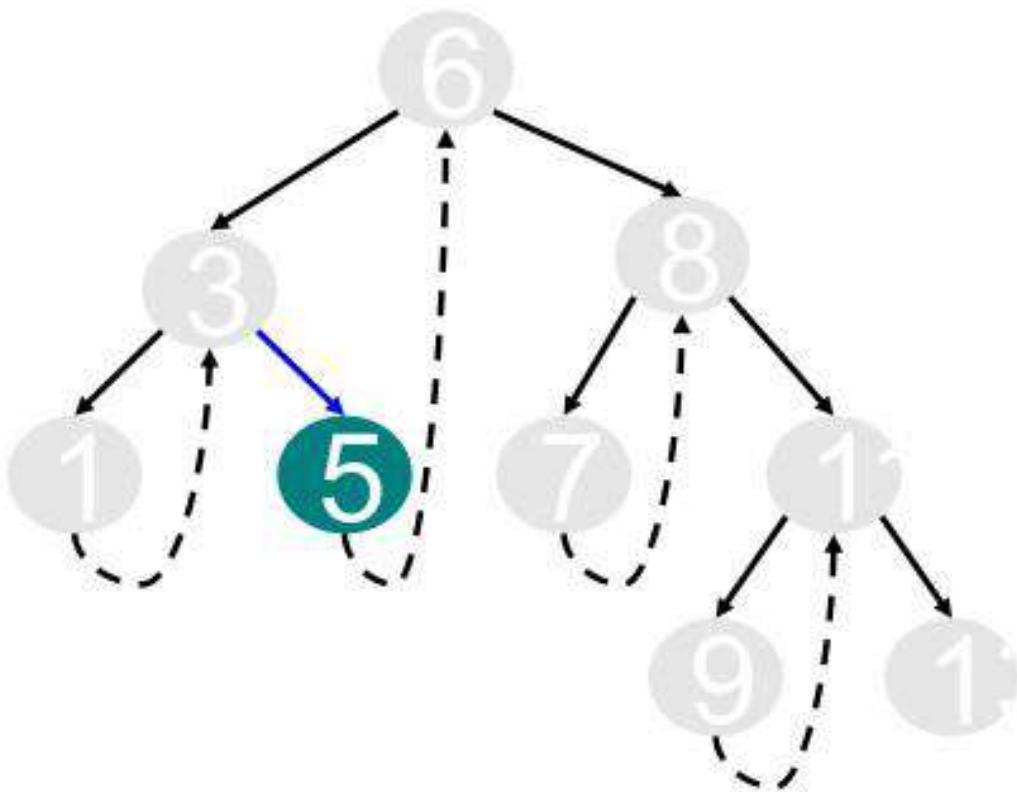
# Threaded Tree Traversal



Output  
1  
3

Follow thread to right, print node

# Threaded Tree Traversal



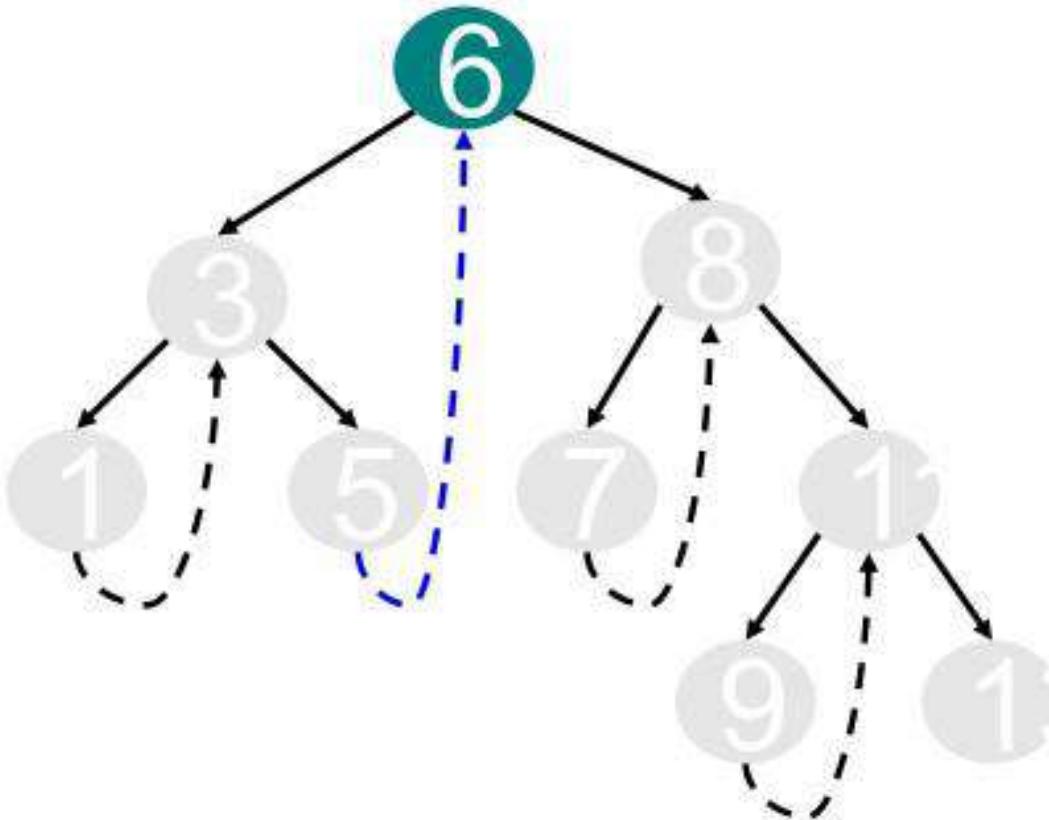
Output  
1  
3  
5

Follow link to right, go to leftmost node and print



**SOMAIYA**  
VIDYAVIHAR UNIVERSITY  
K J Somaiya College of Engineering

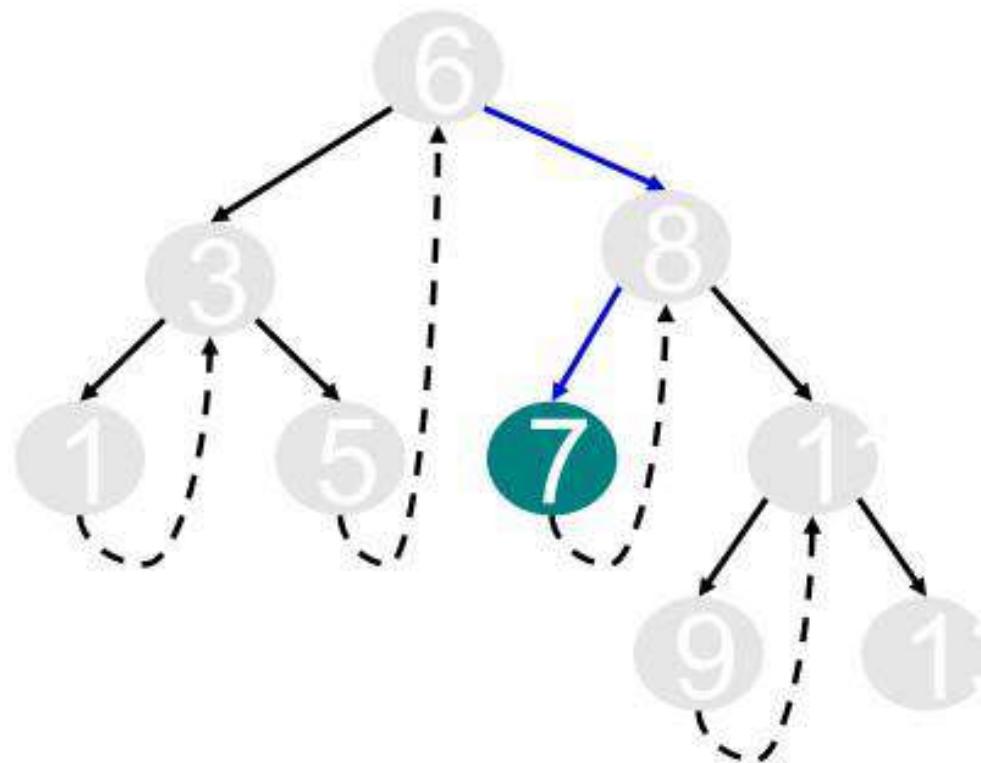
# Threaded Tree Traversals



Output

Follow thread to right, print node

# Threaded Tree Traversal



<u>Output</u>
1
3
5
6
7

Follow link to right, go to  
leftmost node and print

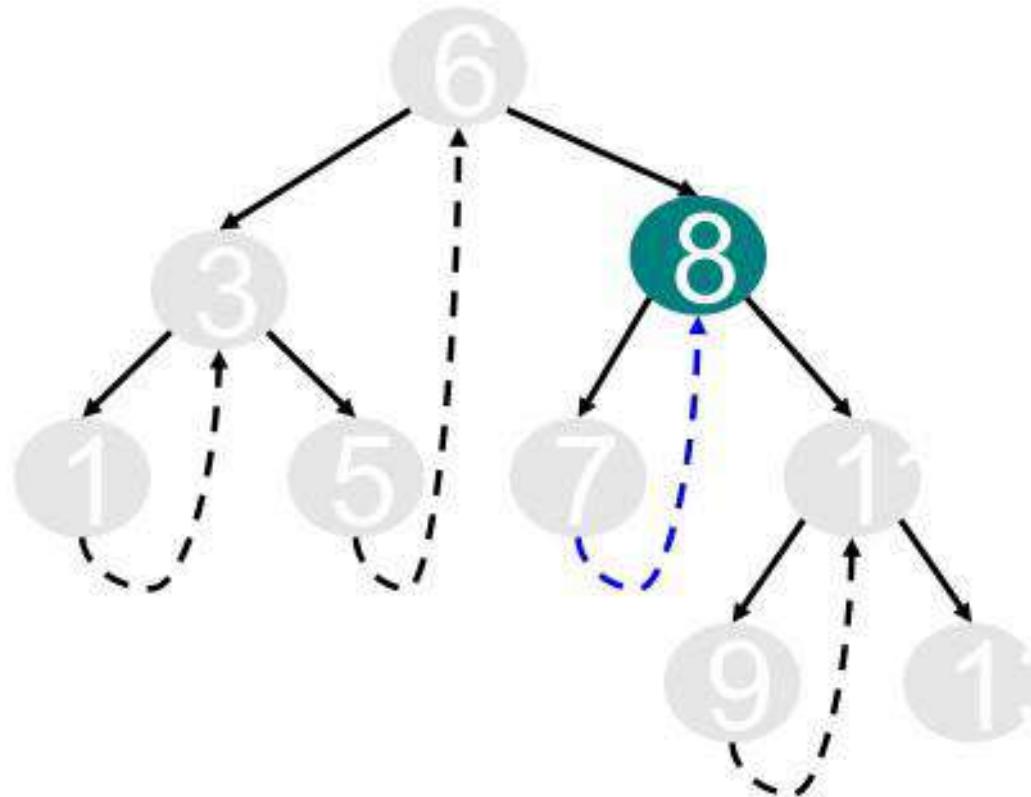


**SOMAIYA**

VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

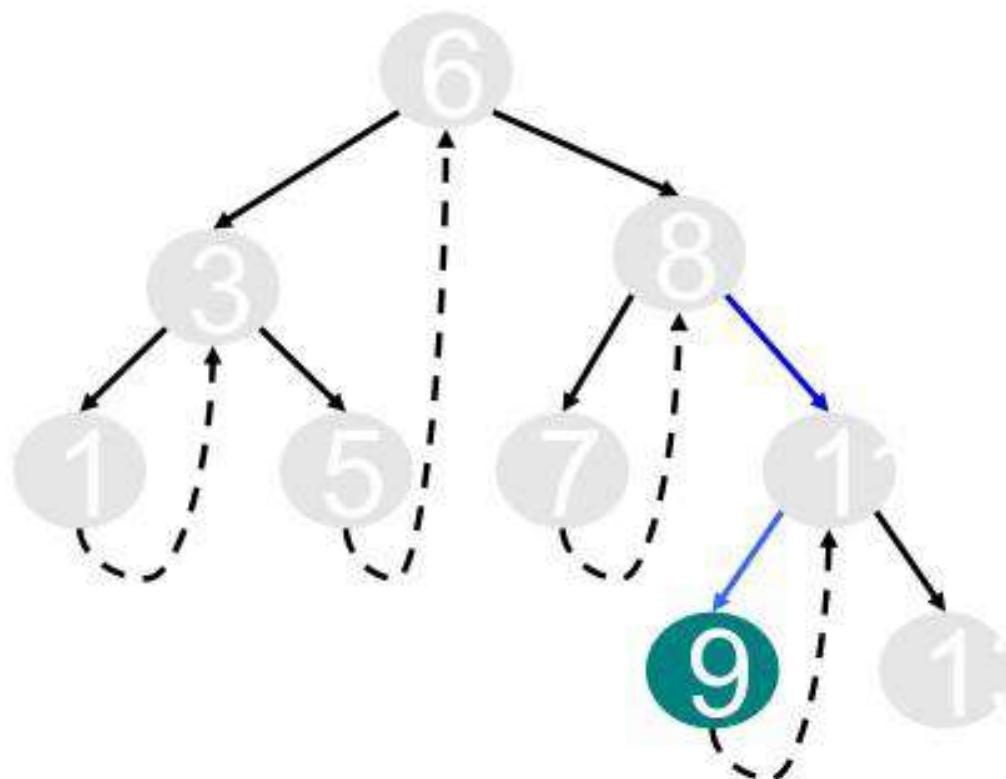
# Threaded Tree Traversal



<u>Output</u>
1
3
5
6
7
8

Follow thread to right, print node

# Threaded Tree Traversal



<u>Output</u>
1
3
5
6
7
8
9

Follow link to right, go to leftmost node and print

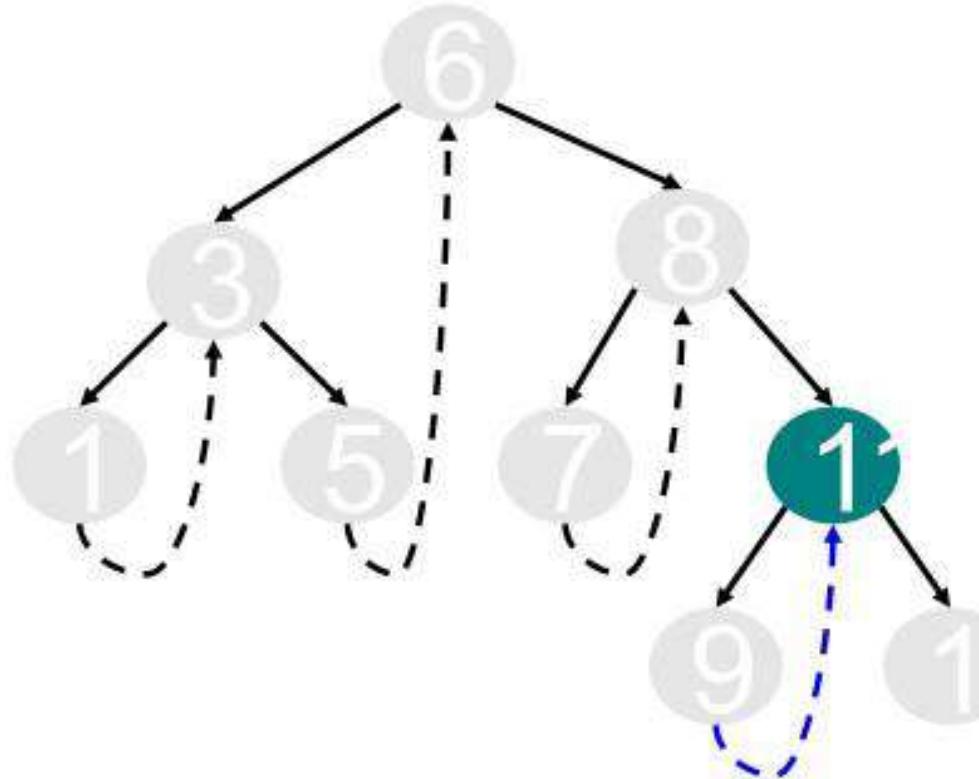


SOMAIYA

VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

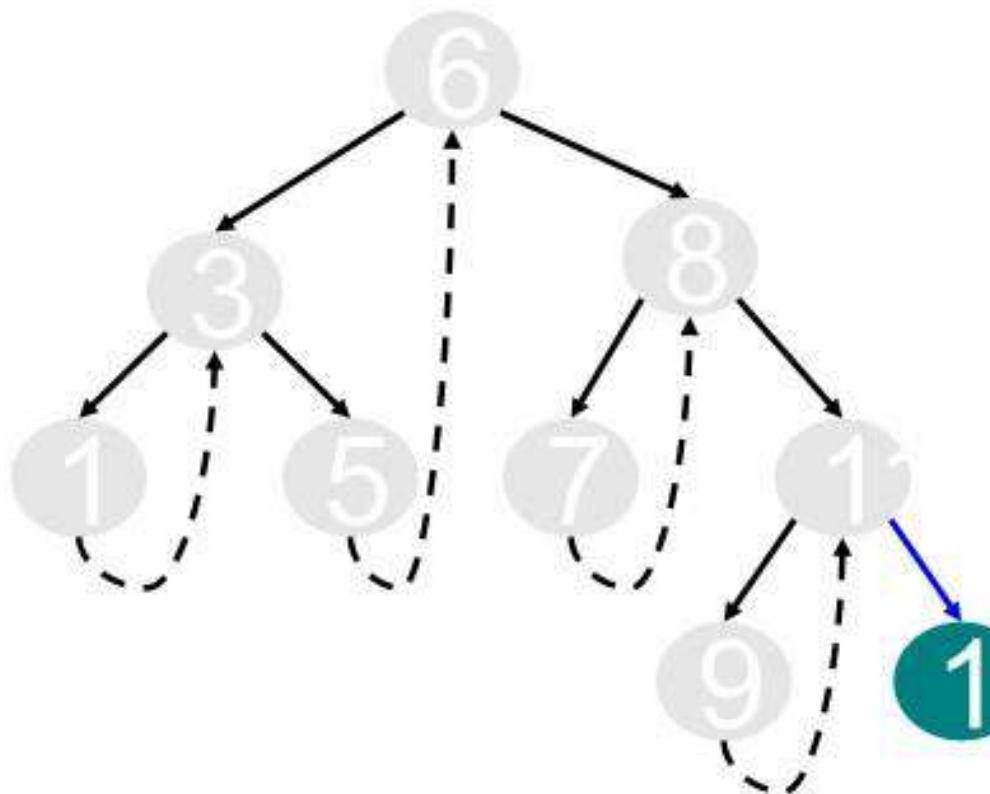
# Threaded Tree Traversal



<u>Output</u>
1
3
5
6
7
8
9
11

Follow thread to right, print node

# Threaded Tree Traversal



## Output

1  
3  
5  
6  
7  
8  
9  
11  
13

Follow link to right, go to  
leftmost node and print



# Threaded Binary Tree

## Two-Way

- In the two-way threading of T.
- A thread will also appear in the left field of a node and will point to the preceding node in the **in-order** traversal of tree T.
- Furthermore, the left pointer of the first node and the right pointer of the last node (in the **in-order** traversal of T) will contain the null value when T does not have a header node.



SOMAIYA

VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

# Threaded Binary Tree

## Two-Way

- Next figure show two-way **in-order** threading.
- Here, right pointer=next node of **in-order** traversal and left pointer=previous node of **in-order** traversal



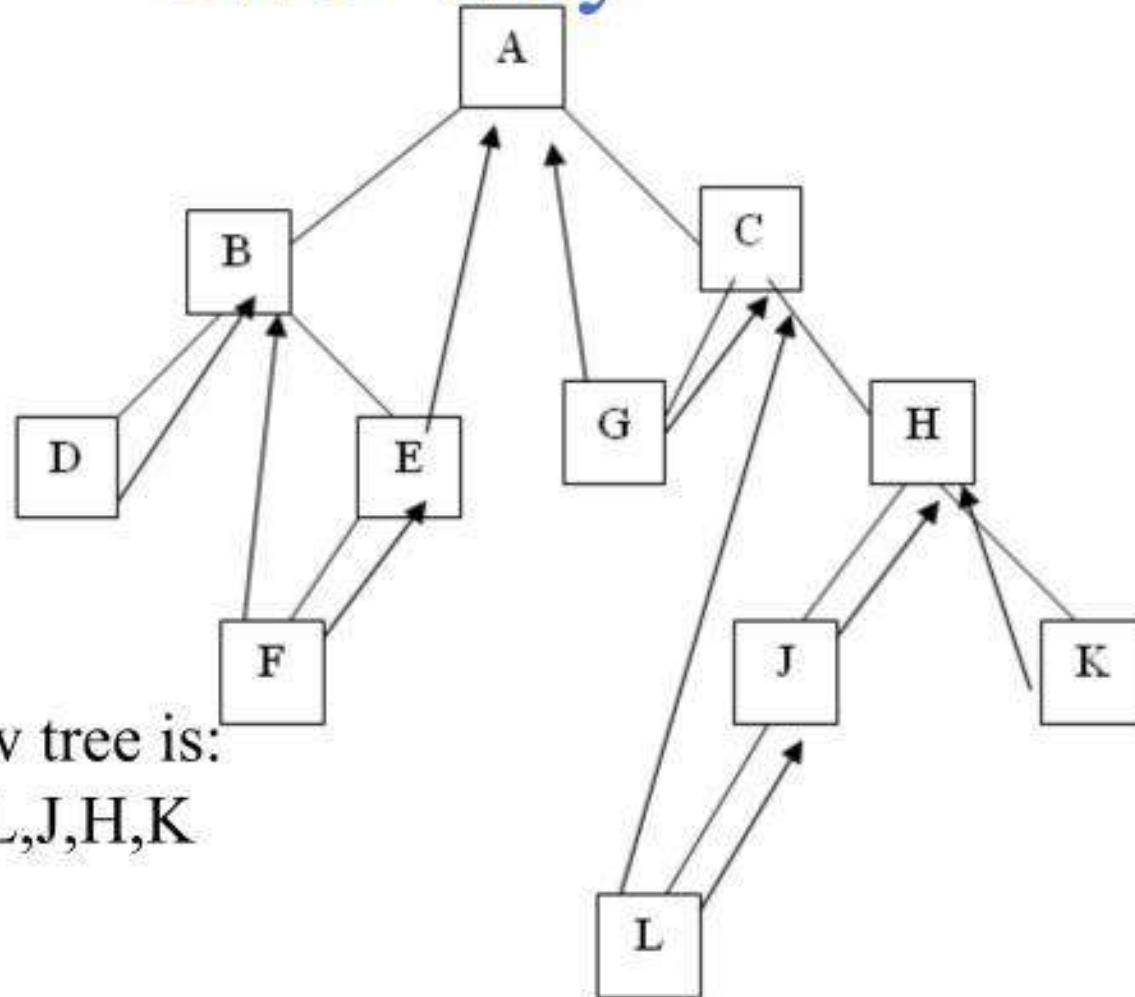
**SOMAIYA**

VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

# Threaded Binary Tree

## Two-Way



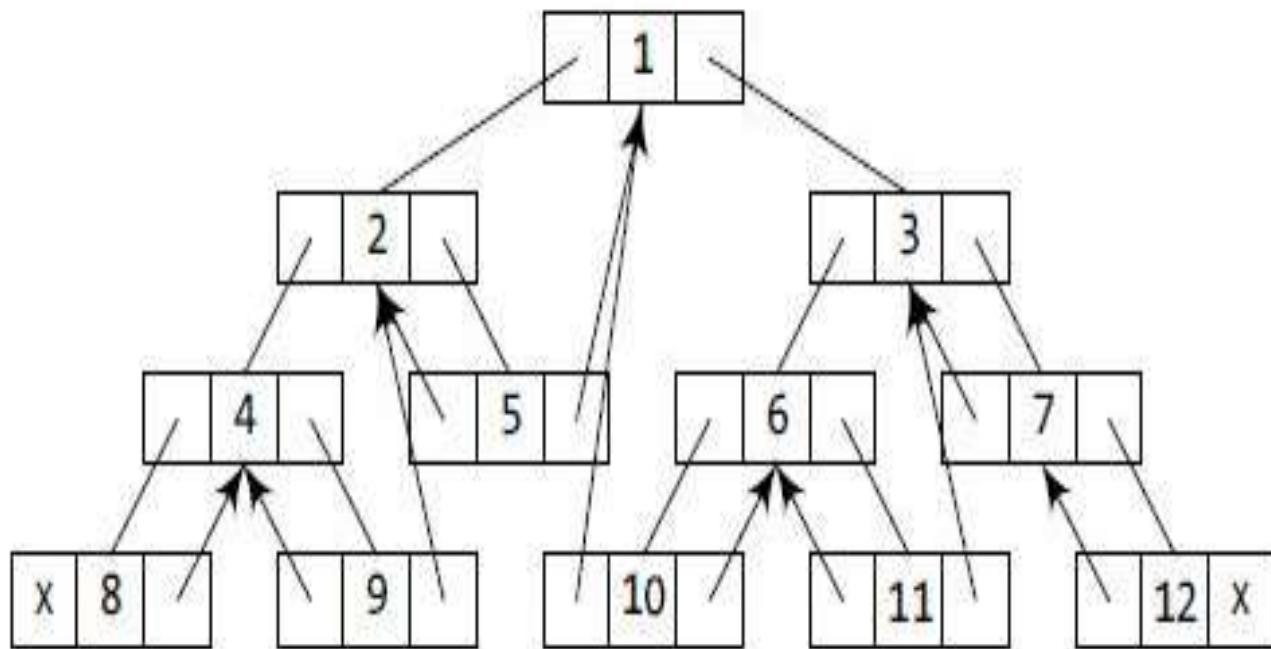
Inorder of bellow tree is:  
D,B,F,E,A,G,C,L,J,H,K

Two-way inorder threading

# Two-way Threading

(8, 4, 9, 2, 5, 1, 10, 6, 11, 3, 7, 12)

- Node 5 contains a NULL pointer in its LEFT field, so it will be replaced to point to node 2, which is its in-order predecessor. Similarly, the LEFT field of node 8 will contain NULL because it has no in-order predecessor, the LEFT field of node 7 will point to node 3, the LEFT field of node 9 will point to node 4, the LEFT field of node 10 will point to node 1, the LEFT field of node 11 will contain 6, and the LEFT field of node 12 will point to node 7.



# Threaded Binary Tree

## Advantages of threaded binary tree:

- The traversal operation is more faster than that of its unthreaded version, because with threaded binary tree non-recursive implementation is possible which can run faster and does not require the botheration of stack management.
- The second advantage is more understated with a threaded binary tree, we can efficiently determine the predecessor and successor nodes starting from any node. In case of unthreaded binary tree, however, this task is more time consuming and difficult. For this case a stack is required to provide upward pointing information in the tree whereas in a threaded binary tree, without having to include the overhead of using a stack mechanism the same can be carried out with the threads.
- Any node can be accessible from any other node. Threads are usually more to upward whereas links are downward. Thus in a threaded tree, one can move in their direction and nodes are in fact circularly linked. This is not possible in unthreaded counter part because there we can move only in downward direction starting from root. Insertion into and deletions from a threaded tree are although time consuming operations but these are very easy to implement.



# Advantages of Threaded Binary Tree

- It enables linear traversal of elements in the tree.
- Linear traversal eliminates the use of stacks which in turn consume a lot of memory space and computer time.
- It enables to find the parent of a given element without explicit use of parent pointers.
- Since nodes contain pointers to in-order predecessor and successor, the threaded tree enables forward and backward traversal of the nodes as given by in-order fashion.
- Thus, we see the basic difference between a binary tree and a threaded binary tree is that in binary trees a node stores a NULL pointer if it has no child and so there is no way to traverse back.

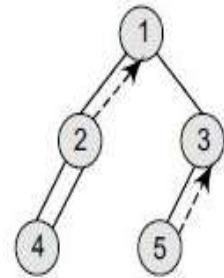
# Threaded Binary Tree

## **Disadvantages of threaded binary tree:**

- Insertion and deletion from a threaded tree are very time consuming operation compare to non-threaded binary tree.
- This tree require additional bit to identify the threaded link.

# Traversing a Threaded Binary Tree

- Node 1 has a left child i.e., 2 which has not been visited. So, add 2 in the list of v nodes, make it as the current node.
- Node 2 has a left child i.e., 4 which has not been visited. So, add 4 in the list of v nodes, make it as the current node.
- Node 4 does not have any left or right child, so print 4 and check for its threaded link. It has a threaded link to node 2, so make node 2 the current node.
- Node 2 has a left child which has already been visited. Does not have a right child. Now, print 2 and follow its threaded link to node 1. Make node 1 the current node.
- Node 1 has a left child that has been already visited. So print 1. Node 1 has a right child 3 which has not yet been visited, so make it the current node.
- Node 3 has a left child (node 5) which has not been visited, so make it the current node.
- Node 5 does not have any left or right child. So print 5. However, it does have a threaded link which points to node 3. Make node 3 the current node.
- Node 3 has a left child which has already been visited. So print 3. Now there are no nodes left, so we end here. **The sequence of nodes printed is—4 2 1 5 3.**



# Traversing a Threaded Binary Tree

- Step 1:** Check if the current node has a left child that has not been visited. If a left child exists that has not been visited, go to Step 2, else go to Step 3.
- Step 2:** Add the left child in the list of visited nodes. Make it as the current node and then go to Step 6.
- Step 3:** If the current node has a right child, go to Step 4 else go to Step 5.
- Step 4:** Make that right child as current node and go to Step 6.
- Step 5:** Print the node and if there is a threaded node make it the current node.
- Step 6:** If all the nodes have visited then END else go to Step 1.

# Types of Tree

- General tree
- Binary tree
- Binary search tree
- Threaded binary tree
- AVL Tree
- B tree
- B+ Tree
- Trie
- Red black tree **#self learning topic**

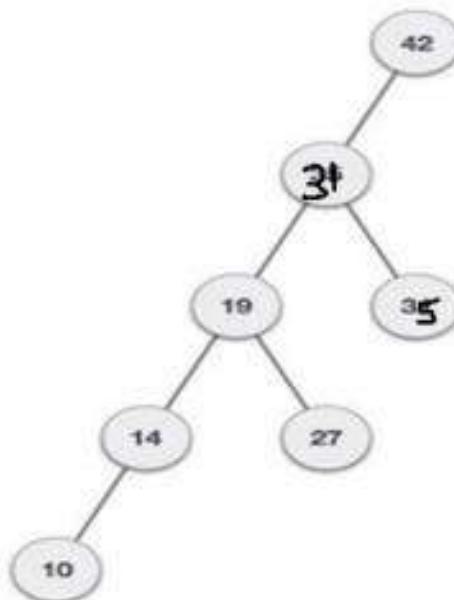
# Height Balanced Search Trees – AVL

# AVL Search Trees

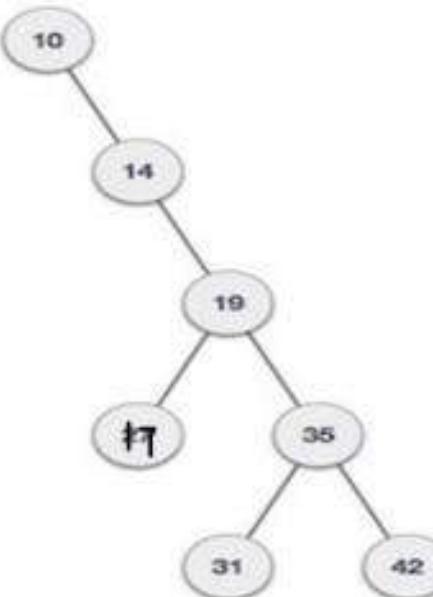
- Problem with BST
- Why AVL Trees?
- What is an AVL Tree?
- AVL Rotations

# What happens to BST?

- Insertion or deletion in an ordinary Binary Search Tree can lead to almost linear structure.
- So what? Skewed BST.



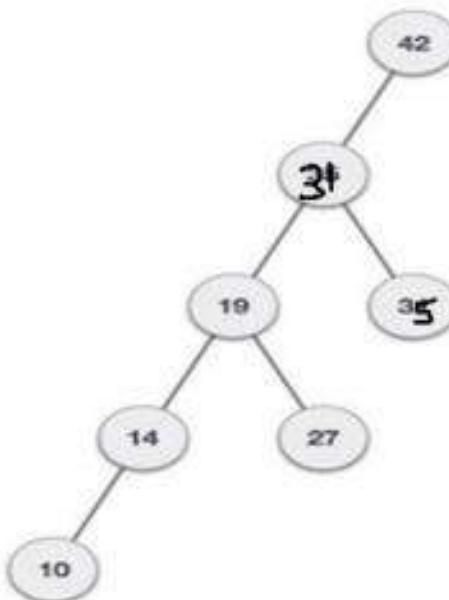
If input 'appears' non-increasing manner



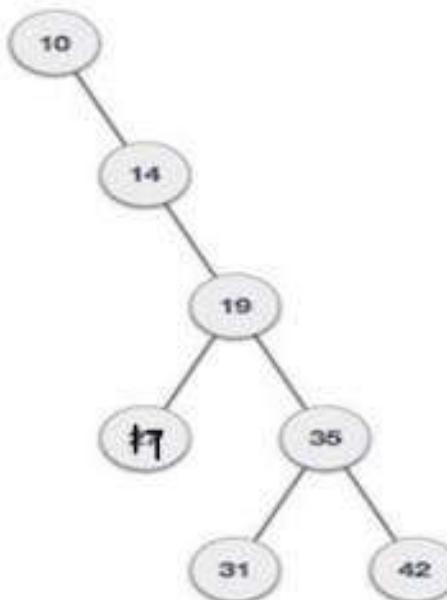
If input 'appears' in non-decreasing manner

# Why AVL Trees?

- Insertion or deletion in an ordinary Binary Search Tree can cause large imbalances.
- Makes search inefficient, like linear structure.



If input 'appears' non-increasing manner

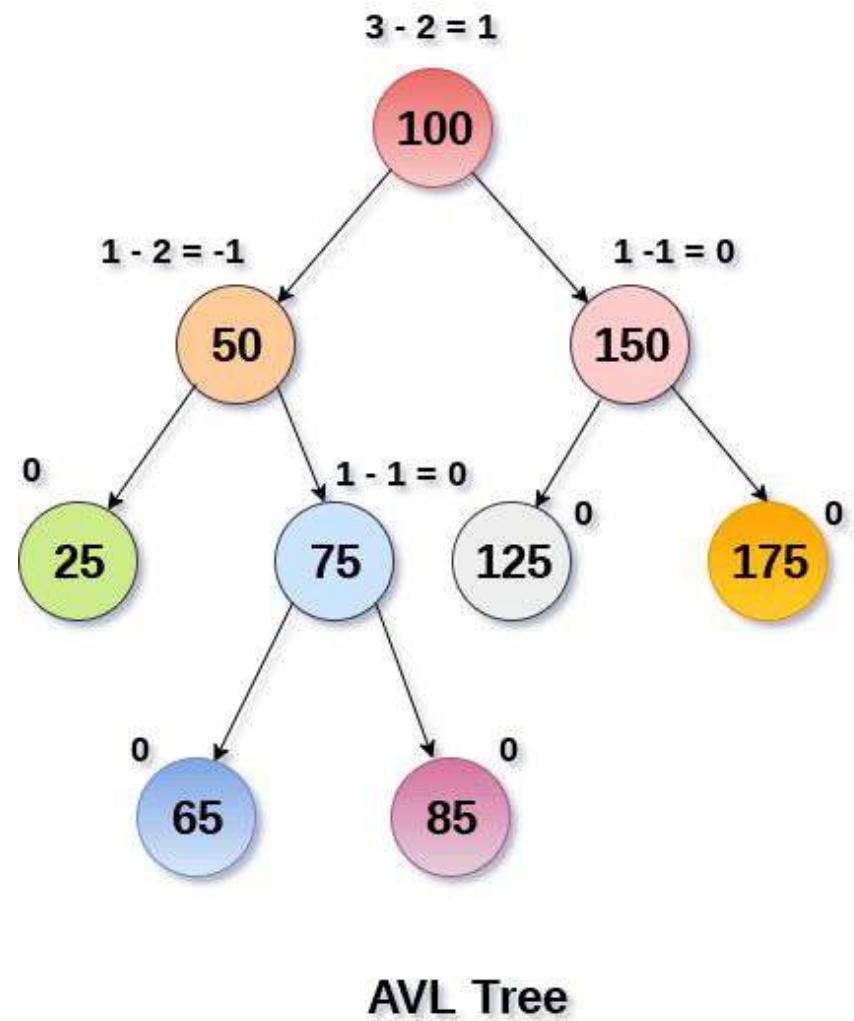


If input 'appears' in non-decreasing manner

# What is an AVL Tree?

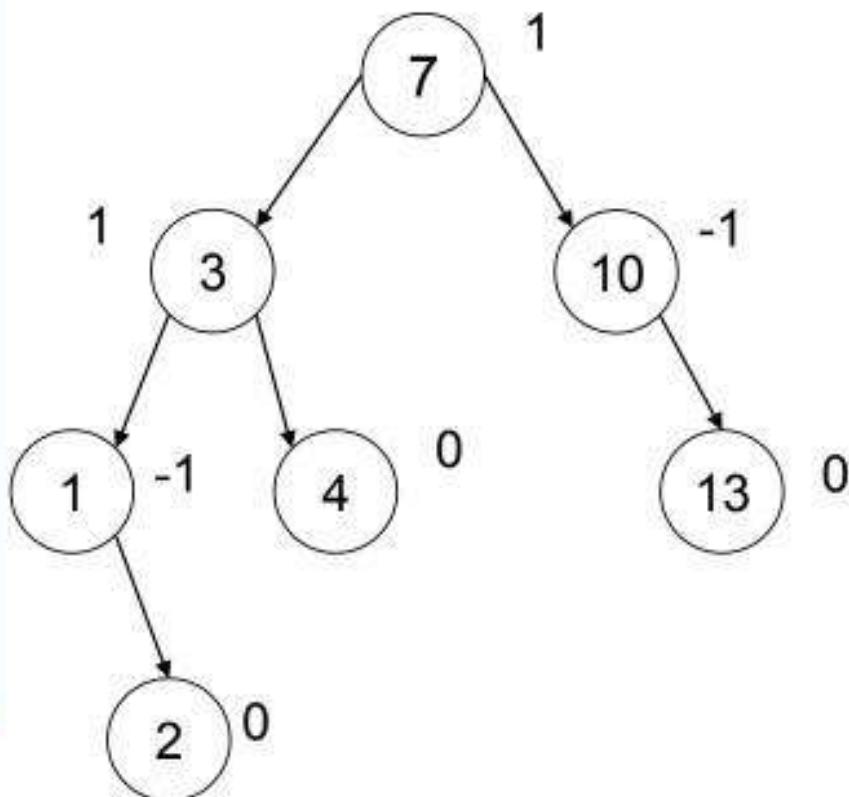
- An **AVL tree** (*named after inventors Adelson-Velsky and Landis*) is a self-balancing binary search tree.
- An AVL tree is a BST with a height balance property:
  - For each node  $v$ , the heights of the subtrees of  $v$  differ by at most 1.
- A subtree of an AVL tree is also an AVL tree.
- For each node of an AVL tree:  
$$\text{Balance factor} = \text{height(left subtree)} - \text{height(right subtree)}$$
- An AVL node can have a balance factor of **-1, 0, or +1**.

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.
- An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an **example of AVL tree**.

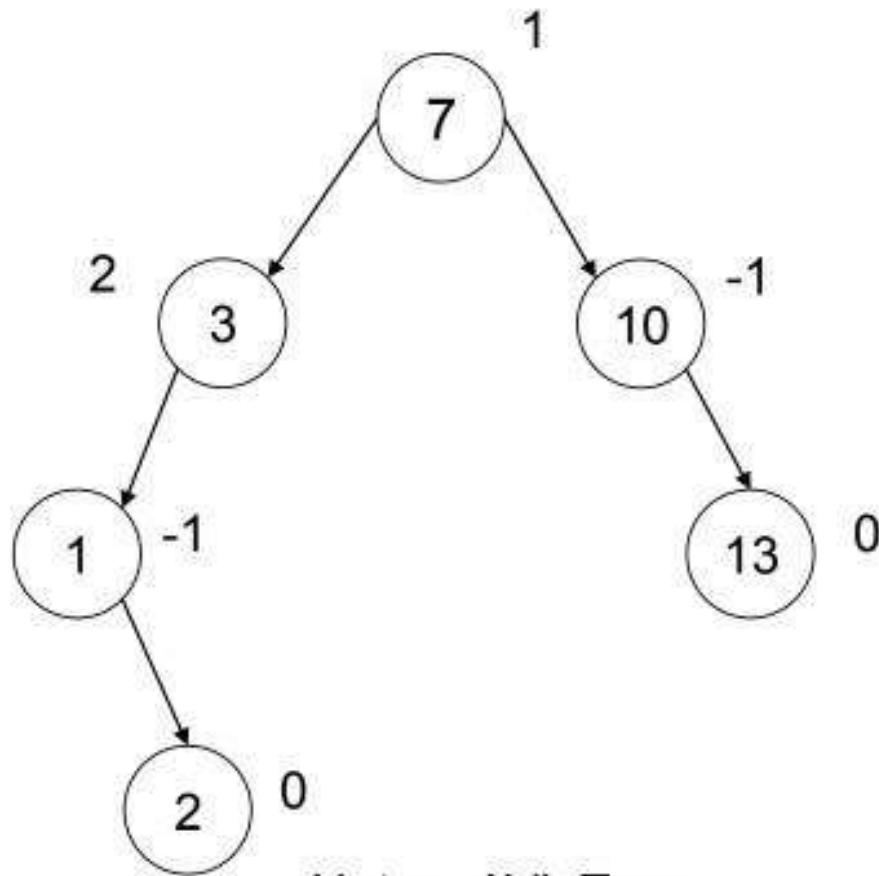


# Is it an AVL Tree?

- An AVL node can have a balance factor of -1, 0, or +1.

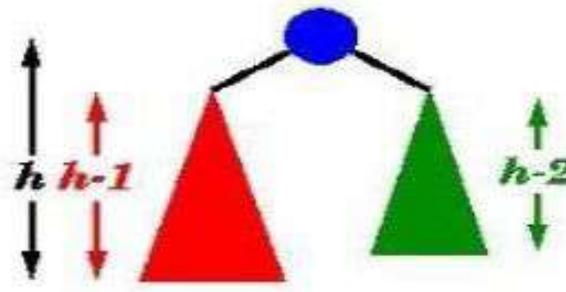


AVL Tree

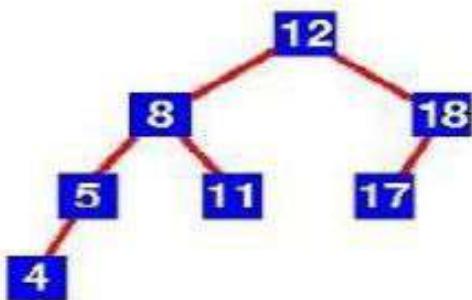


Not an AVL Tree

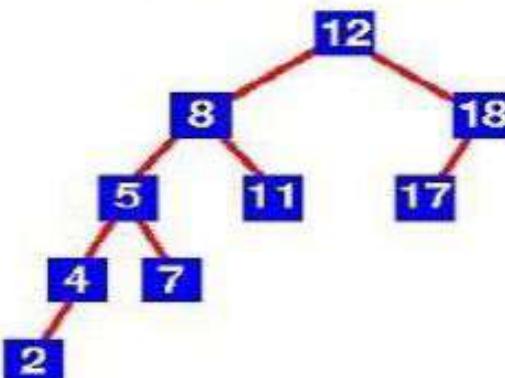
Balance requirement for an AVL tree: the left and right sub-trees differ by at most 1 in height.



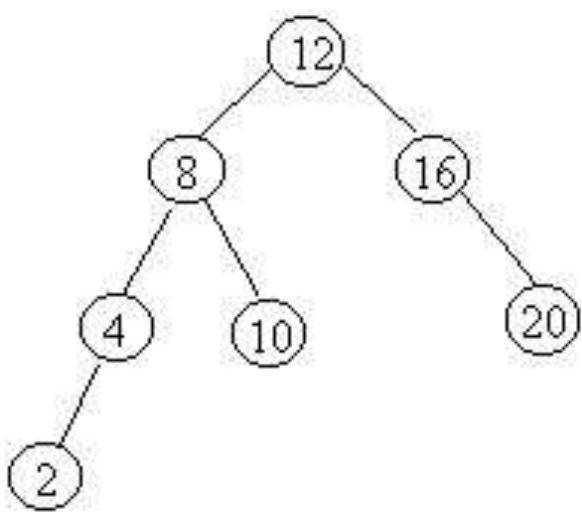
For example, here are some trees:



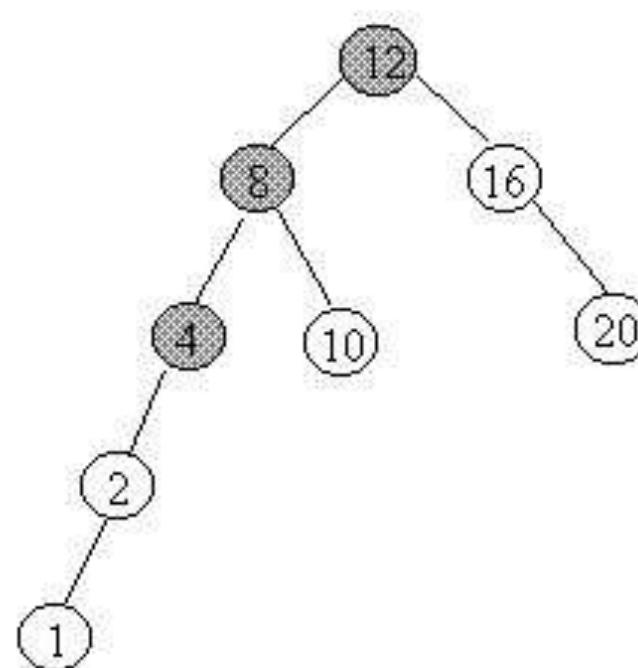
Yes this is an AVL tree. Examination shows that *each* left sub-tree has a height 1 greater than each right sub-tree.



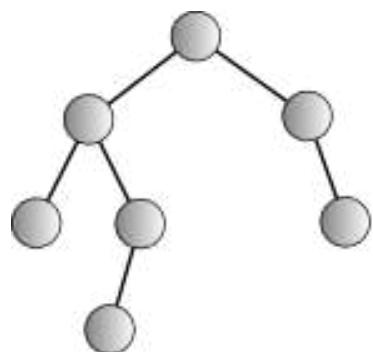
No this is not an AVL tree. Sub-tree with root 8 has height 4 and sub-tree with root 18 has height



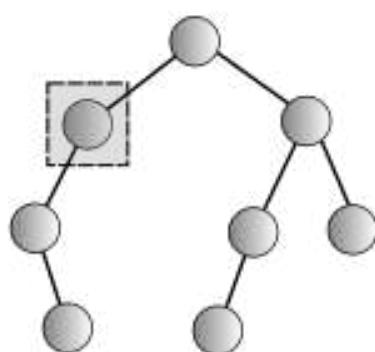
an AVL tree



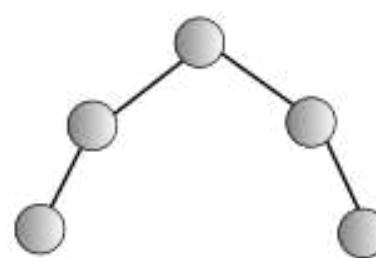
a non-AVL tree



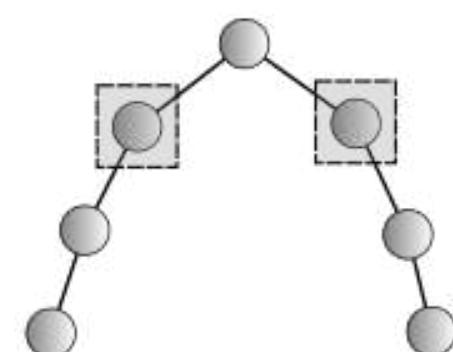
AVL



Not AVL



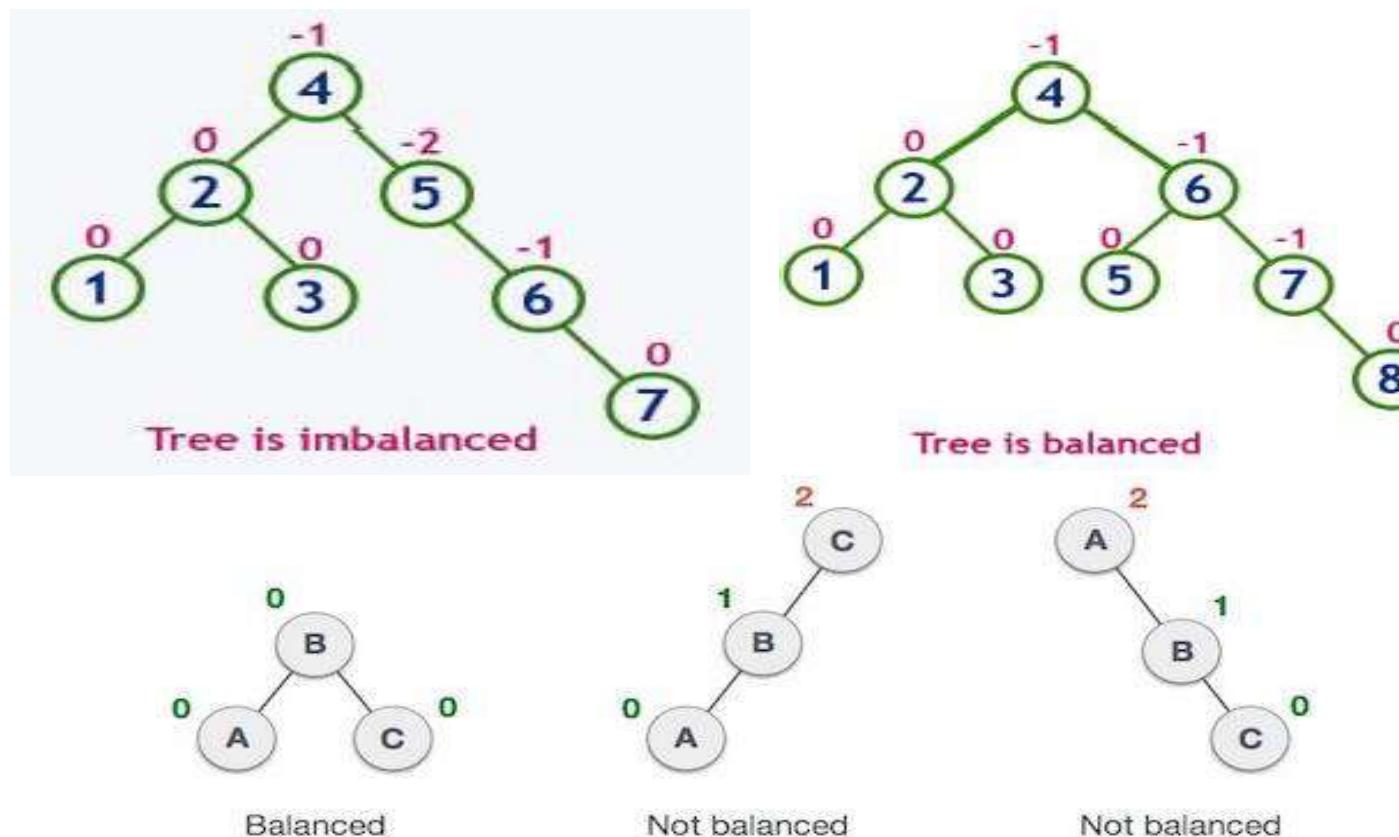
AVL



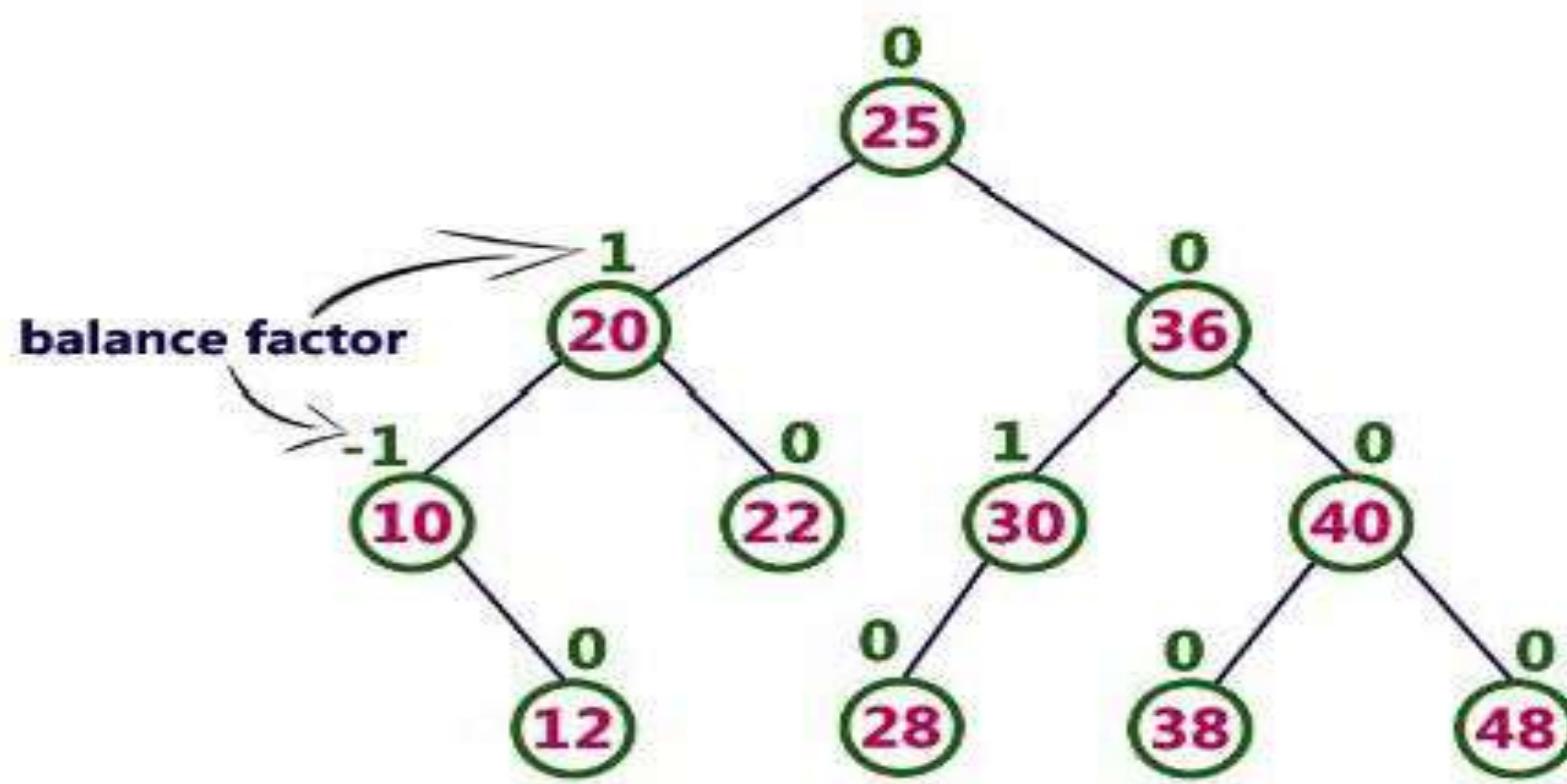
Not AVL

# AVL trees

- Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.



# AVL trees



The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

# Balance Factor

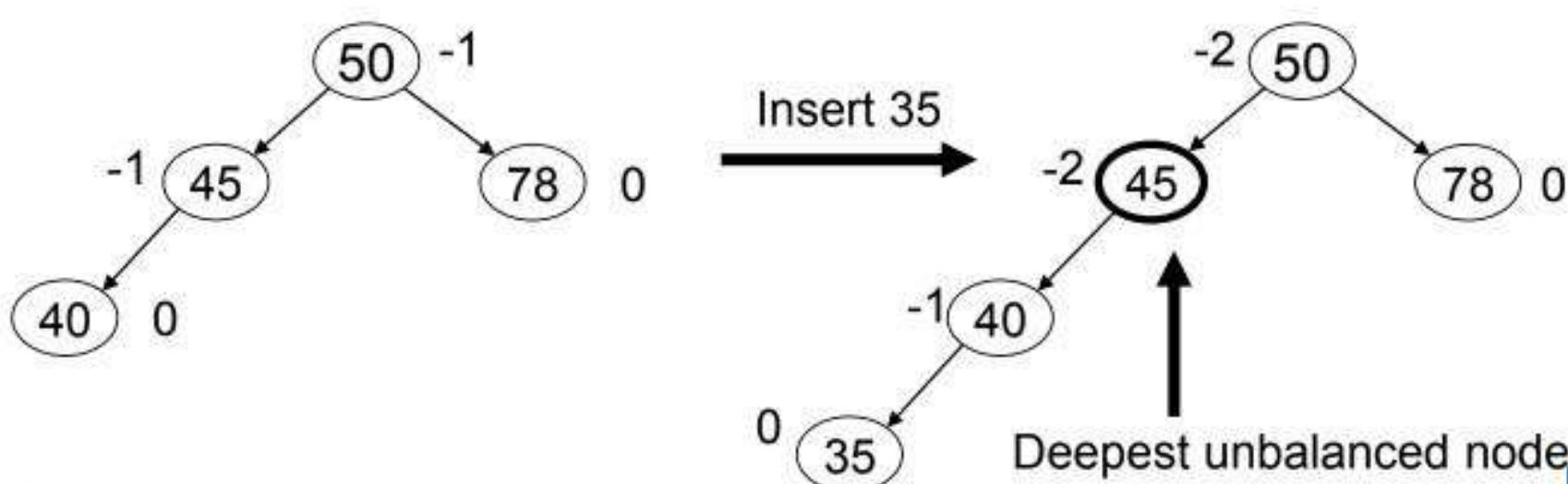
- To implement our AVL tree we need to keep track of a **balance factor** for each node in the tree. We do this by looking at the heights of the left and right subtrees for each node. More formally, we define the balance factor for a node as the difference between the height of the left subtree and the height of the right subtree.

*balanceFactor=height(leftSubTree)–height(rightSubTree)*

- Using the definition for balance factor given above we say that a subtree is left-heavy if the balance factor is greater than zero. If the balance factor is less than zero then the subtree is right heavy. If the balance factor is zero then the tree is perfectly in balance.

# What is a Rotation?

- A rotation is a process of switching children and parents among two or three adjacent nodes to restore balance to a tree.
- **An insertion or deletion may cause an imbalance in an AVL tree.**
  - The deepest node, which is an ancestor of a deleted or an inserted node, and whose balance factor has changed to -2 or +2 requires rotation to rebalance the tree.

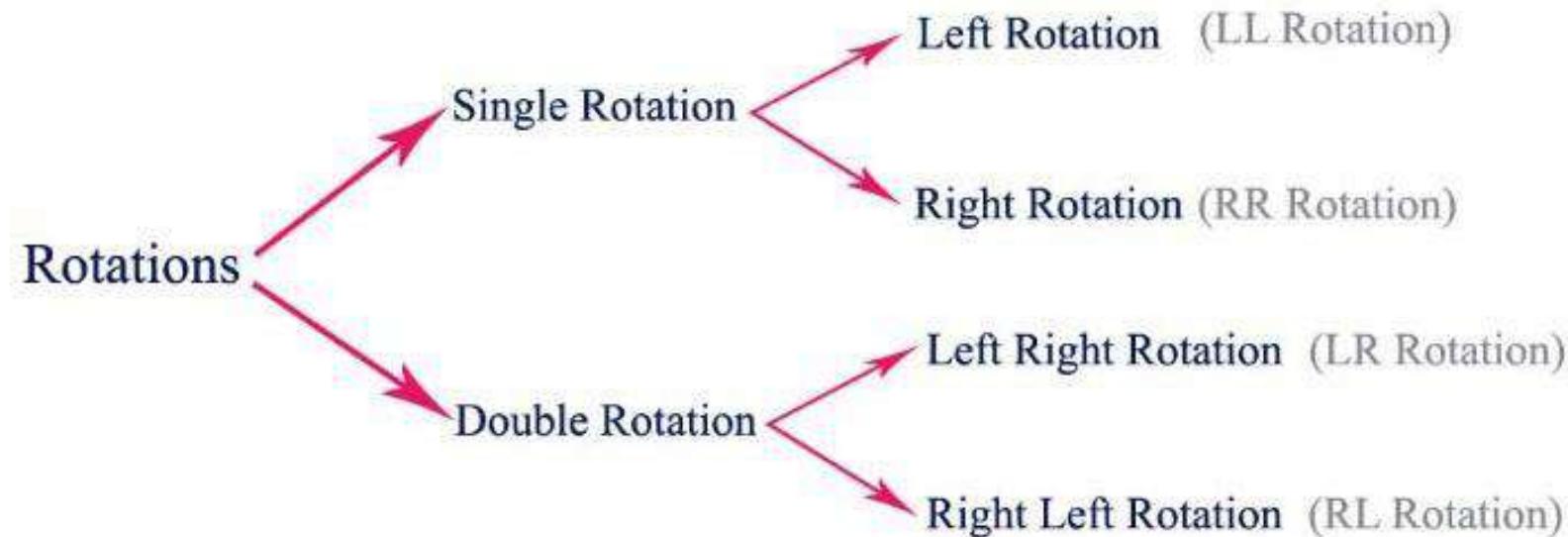


# AVL Tree Rotations

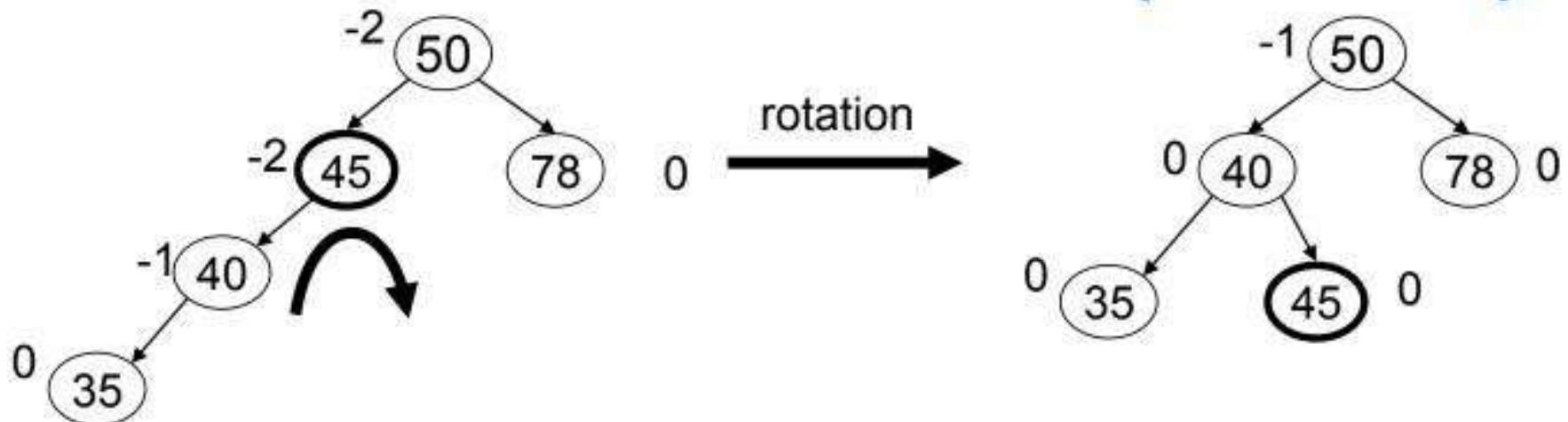
- In AVL tree, after performing operations like insertion and deletion we need to check the **balance factor** of every node in the tree.
- If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced.
- Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.

# AVL Tree Rotations

- Rotation operations are used to make the tree balanced.
- Rotation is the process of moving nodes either to left or to right to make the tree balanced.



# What is a Rotation? (contd.)



There are two kinds of single rotation:

Right Rotation.



Left Rotation.

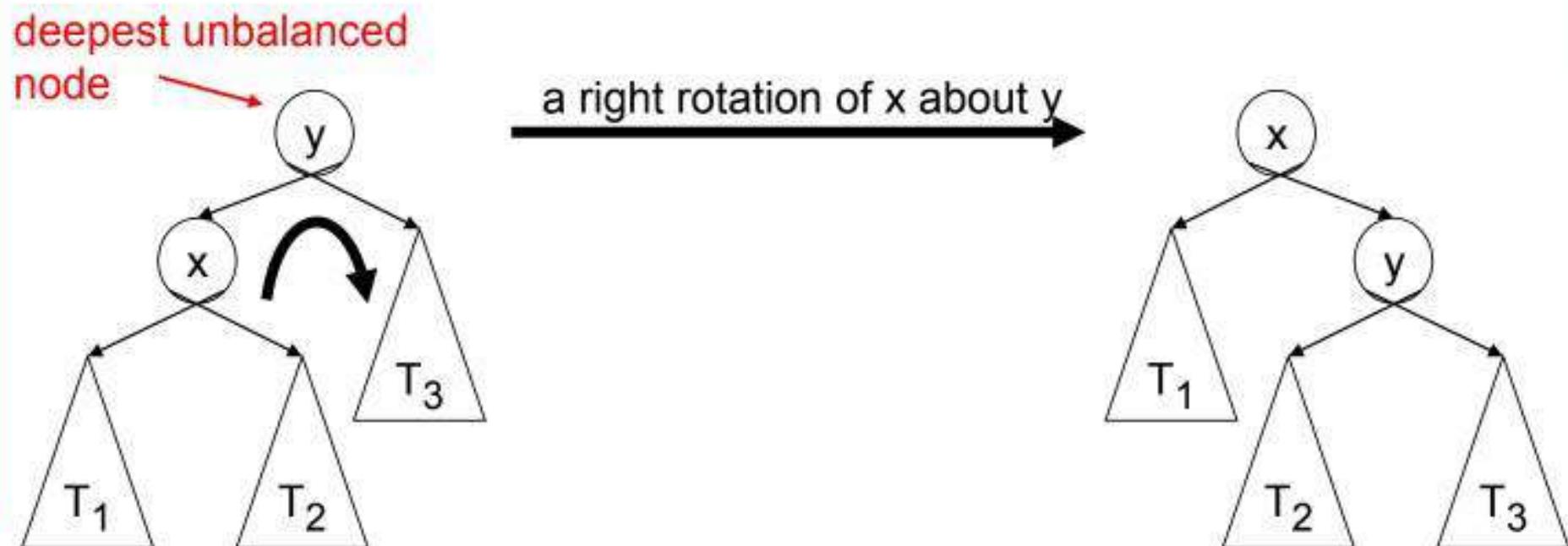


A double right-left rotation is a right rotation followed by a left rotation.  
A double left-right rotation is a left rotation followed by a right rotation.

# Single Right Rotation

Single right rotation:

- The left child  $x$  of a node  $y$  becomes  $y$ 's parent.
- $y$  becomes the right child of  $x$ .
- The right child  $T_2$  of  $x$ , if any, becomes the left child of  $y$ .

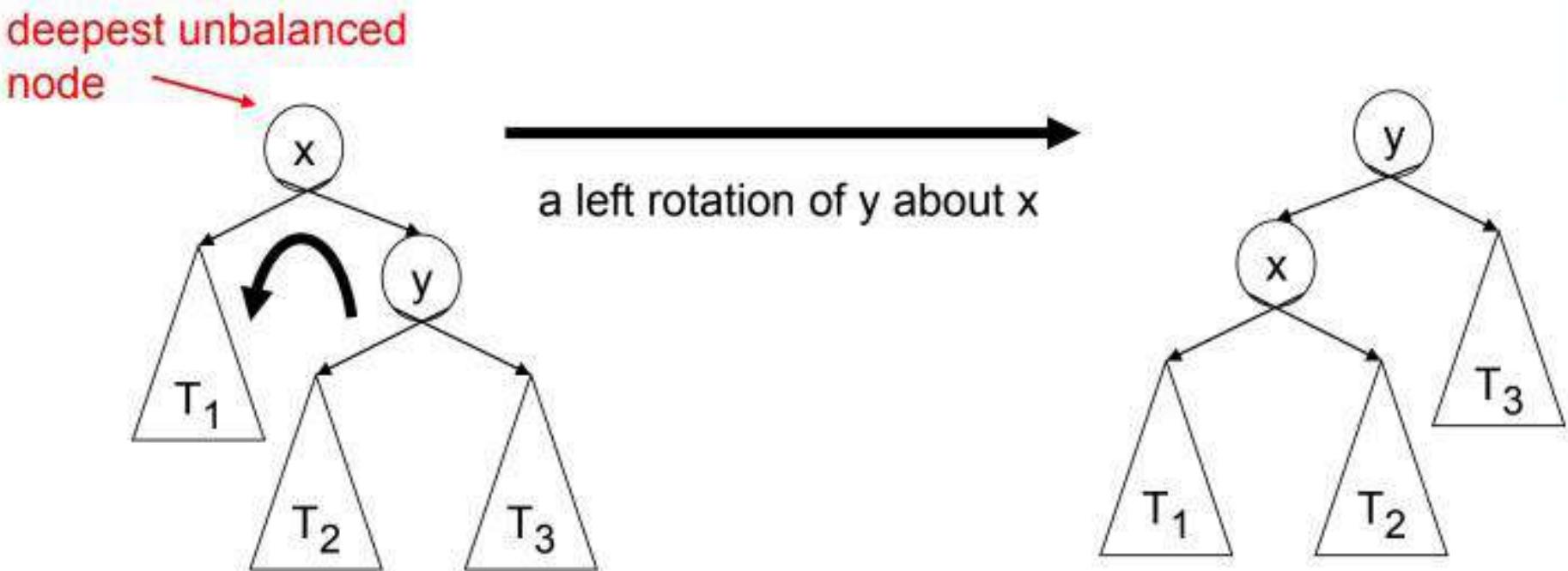


*Note: The pivot of the rotation is the deepest unbalanced node*

# Single Left Rotation

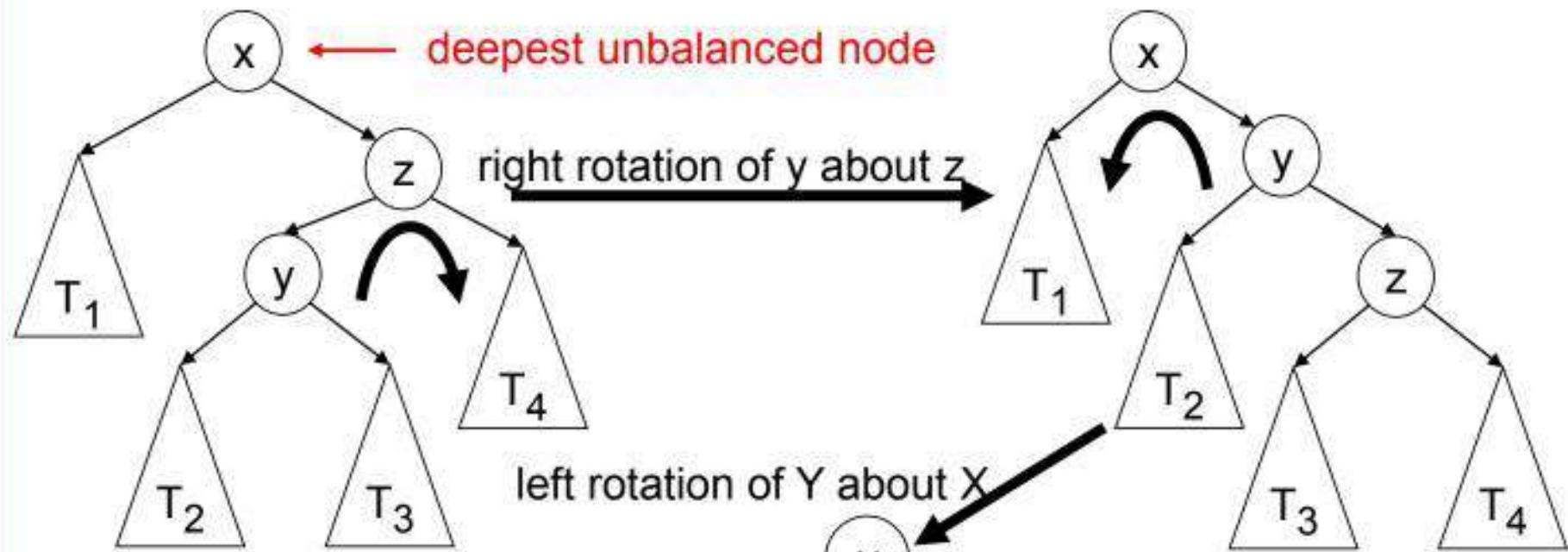
Single left rotation:

- The right child  $y$  of a node  $x$  becomes  $x$ 's parent.
- $x$  becomes the left child of  $y$ .
- The left child  $T_2$  of  $y$ , if any, becomes the right child of  $x$ .



*Note: The pivot of the rotation is the deepest unbalanced node*

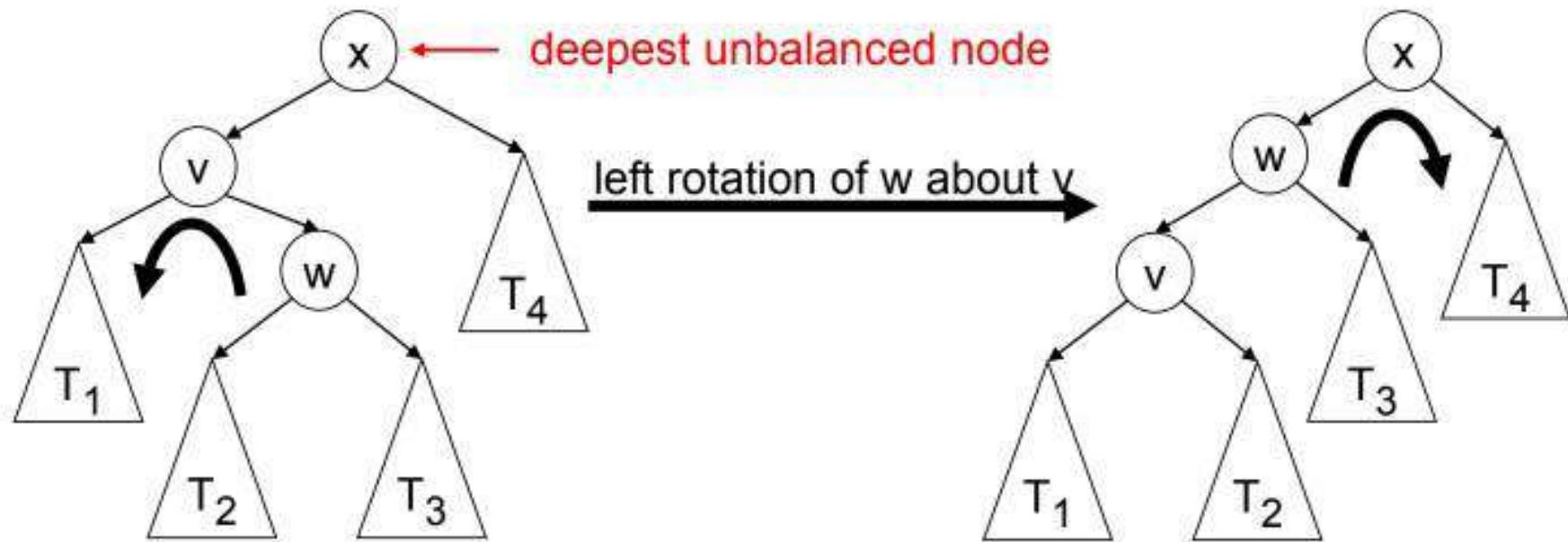
# Double Right-Left Rotation



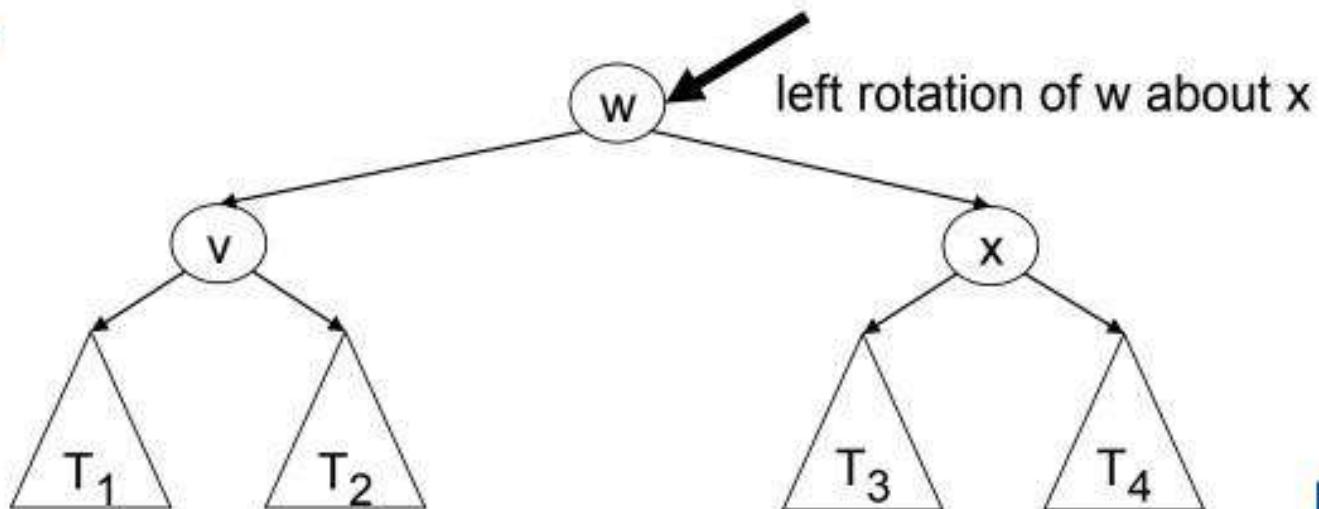
*Note: First pivot is the right child of the deepest unbalanced node; second pivot is the deepest unbalanced node*



# Double Left-Right Rotation

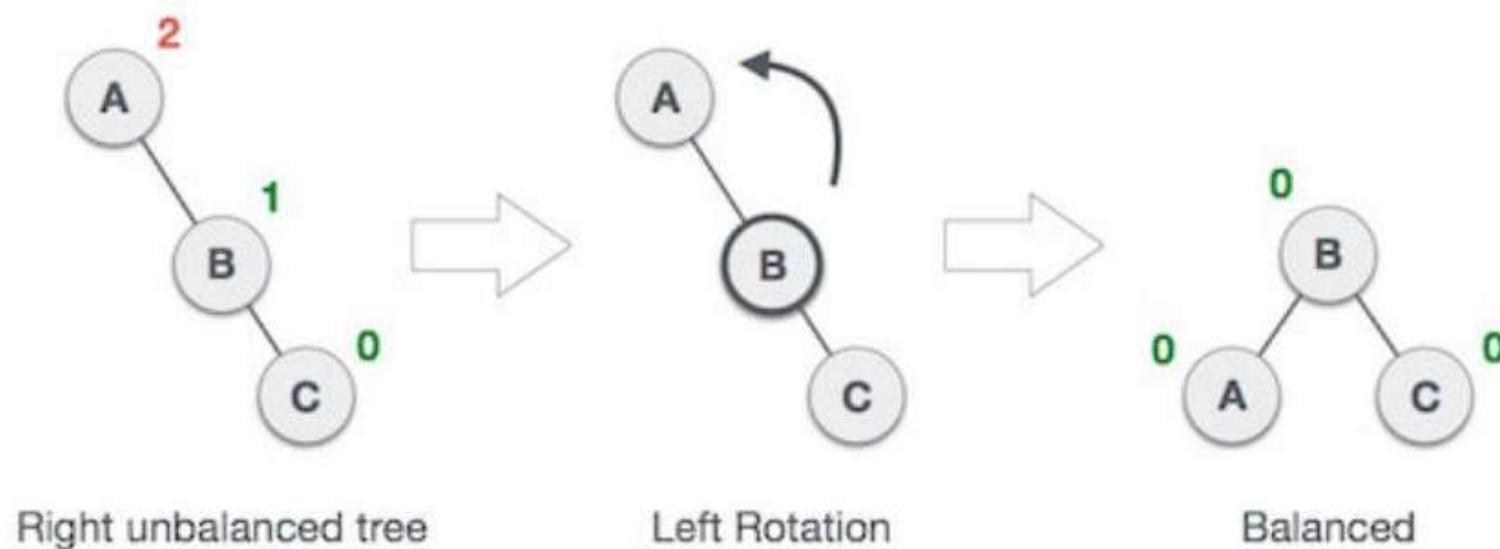


*Note: First pivot is the left child of the deepest unbalanced node; second pivot is the deepest unbalanced node*



## 1. RR Rotation

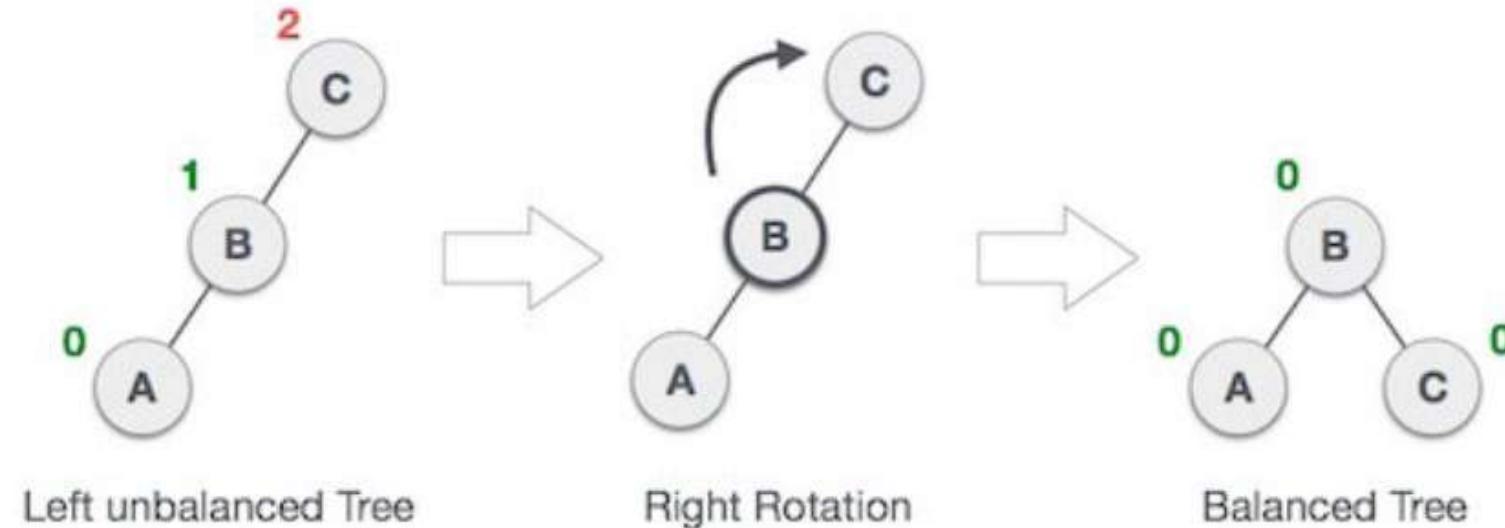
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, **RR rotation** is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

## 2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, **LL rotation** is clockwise rotation, which is applied on the edge below a node having balance factor 2.

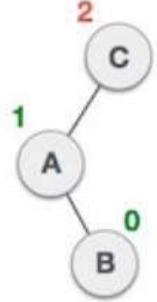
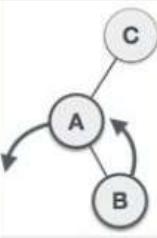


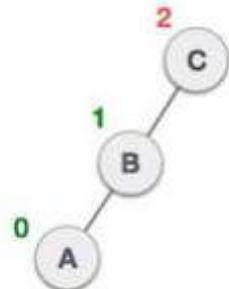
In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

### 3. LR Rotation

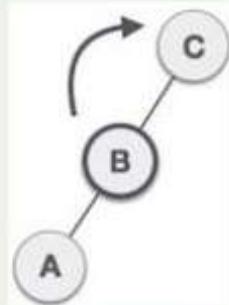
Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

**Let us understand each and every step very clearly:**

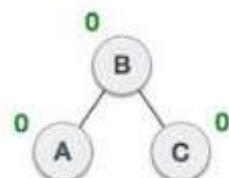
State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node <b>A</b> , has become the left subtree of <b>B</b> .



After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**



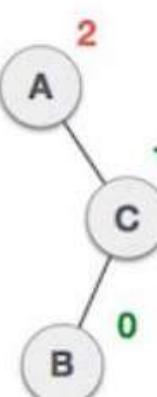
Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B

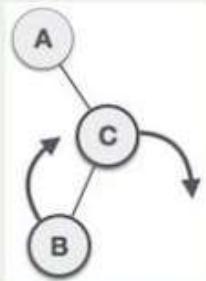


Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

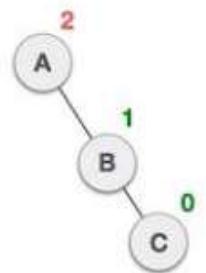
## 4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. **R L rotation** = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

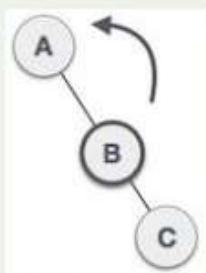
State	Action
	A node <b>B</b> has been inserted into the left subtree of <b>C</b> the right subtree of <b>A</b> , because of which <b>A</b> has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of <b>A</b>



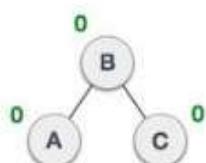
As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**.



After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.



Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.



Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

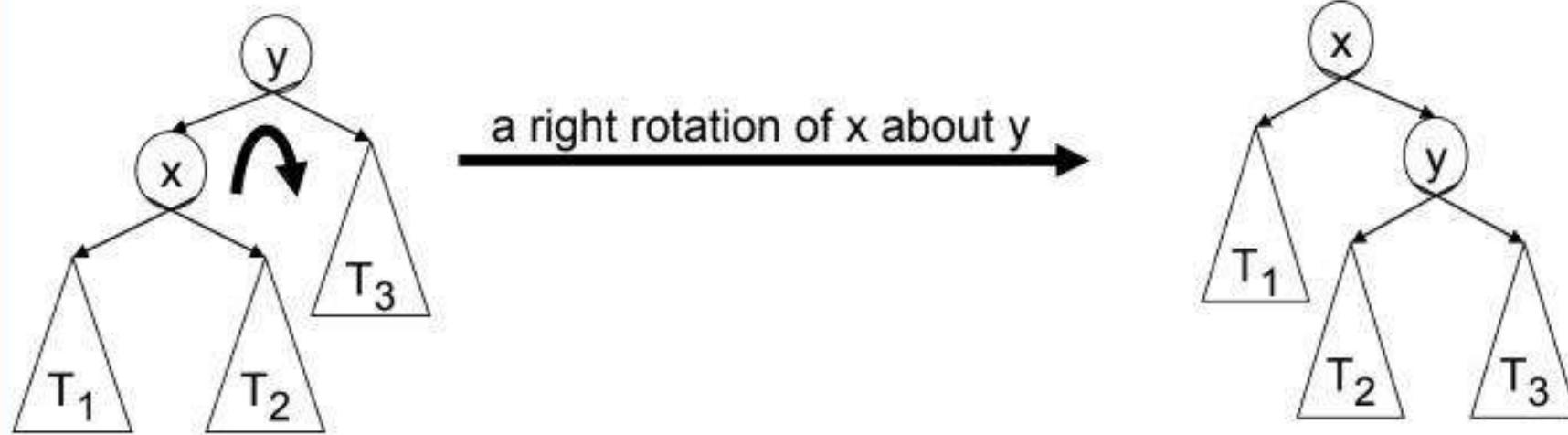
# SUMMARY

*To balance that node:*

- Count three nodes in the direction of the leaf node.
- Then, use the concept of AVL Tree Rotations to rebalance the tree.
  - **LL Rotation** - *In LL rotation, every node moves one position to left from the current position.*
  - **RR Rotation** - *In RR rotation, every node moves one position to right from the current position.*
  - **LR Rotation** - *In LR rotation, at first, every node moves one position to the left and then one position to right from the current position.*
  - **RL Rotation** - *In RL rotation, at first every node moves one position to right and then one position to left from the current position.*

# BST ordering property after a rotation

A rotation does not affect the ordering property of a BST (Binary Search Tree).



BST ordering property requirement: BST ordering property requirement:

$$T_1 < x < y$$

$$x < T_2 < y$$

$$x < y < T_3$$

**Similar**

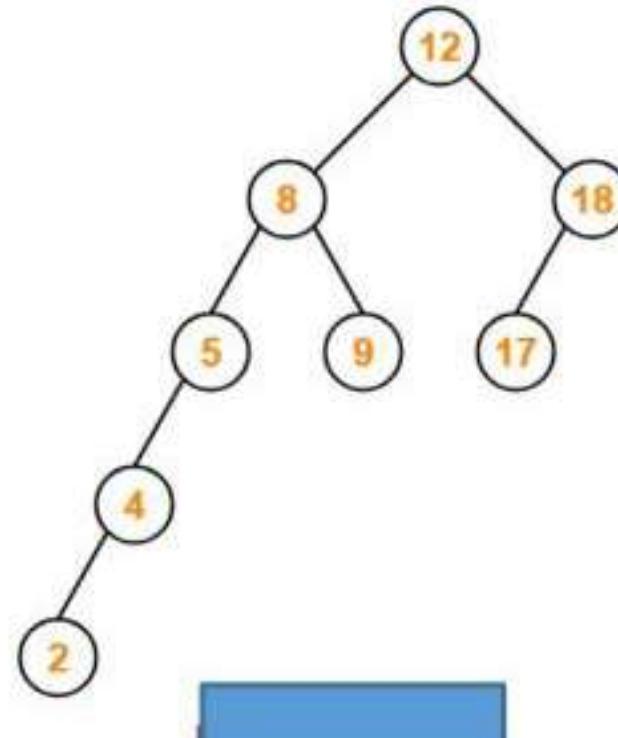
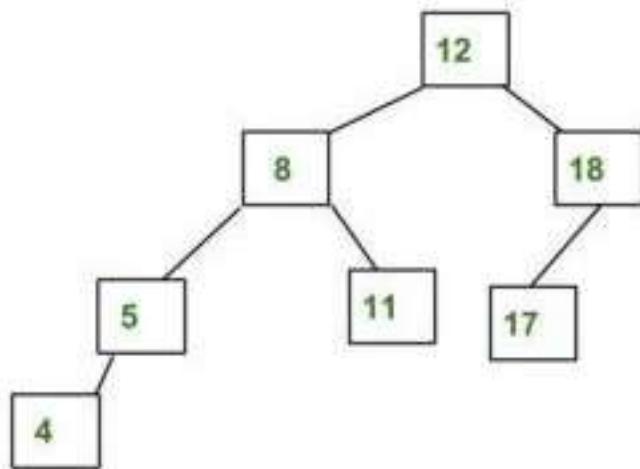
$$T_1 < x < y$$

$$x < T_2 < y$$

$$x < y < T_3$$

- Similarly for a left rotation.

Find Balance factor for each node on the given BST and comment on AVL or Not-AVL



# AVL Tree Insertion

- Insertion in AVL tree is performed in the same way as it is performed in a binary search tree.
- The new node is added into AVL tree as the leaf node. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing.
- The tree can be balanced by applying rotations. Rotation is required only if, the balance factor of any node is disturbed upon inserting the new node, otherwise the rotation is not required.

## Steps for Creating AVL Tree :

Suppose we are given an AVL tree  $T$  and  $N$  is the new node to be inserted.

- Perform a standard BST insertion of node  $N$  in the AVL tree  $T$ .
- Starting from node  $N$ , traverse up until the first unbalanced node is not found. *Let  $z$  be the first unbalanced node,  $y$  be the child of  $z$  that comes on the path from  $p$  to  $z$  and  $x$  be the grandchild of  $z$  that comes on the path from  $p$  to  $z$ .*
- Rebalance the tree by performing suitable rotations on the subtree. There can be the following 4 possible cases and after choosing what case is it, we have to do appropriate rotation explained above.
  1. Left-Left Case –  $y$  is left child of  $z$  and  $x$  is left child of  $y$
  2. Left-Right Case –  $y$  is left child of  $z$  and  $x$  is the right child of  $y$
  3. Right-Right Case –  $y$  is the right child of  $z$  and  $x$  is the right child of  $y$
  4. Right-Left Case –  $y$  is the right child of  $z$  and  $x$  is left child of  $y$

Note: We only need to re-balance the subtree where the first imbalance has occurred using the above cases and the complete tree automatically becomes height-balanced.

# AVL Tree

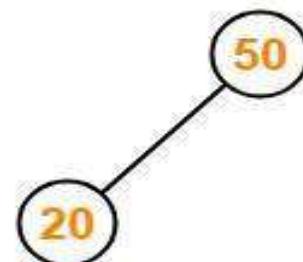
- Construct AVL Tree for the following sequence of numbers- 50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **Step-01: Insert 50**



Tree is Balanced

# AVL Tree

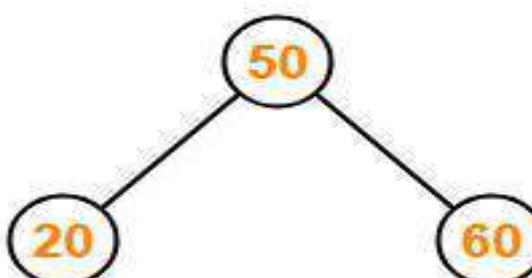
- Construct AVL Tree for the following sequence of numbers- 50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **Step-02: Insert 20**
- As  $20 < 50$ , so insert 20 in 50's left sub tree.



Tree is Balanced

# AVL Tree

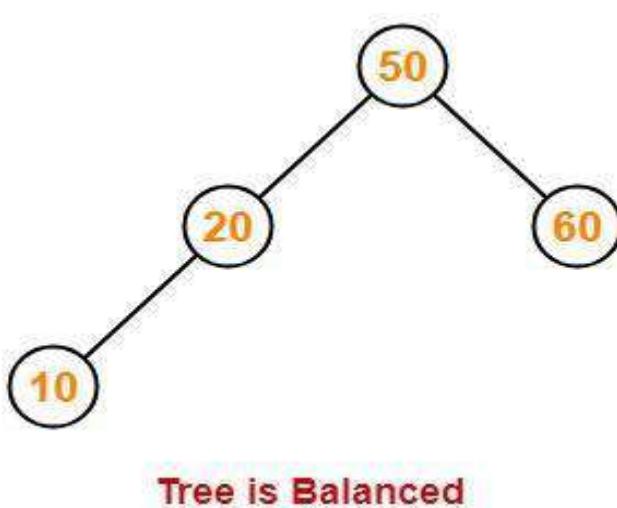
- Construct AVL Tree for the following sequence of numbers- 50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **Step-03: Insert 60**
- As  $60 > 50$ , so insert 60 in 50's right sub tree.



Tree is Balanced

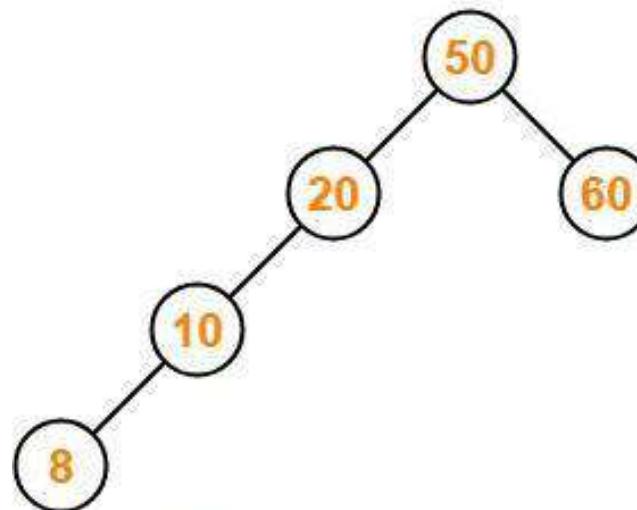
# AVL Tree

- Construct AVL Tree for the following sequence of numbers- 50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **Step-04: Insert 10**
  - As  $10 < 50$ , so insert 10 in 50's left sub tree.
  - As  $10 < 20$ , so insert 10 in 20's left sub tree.



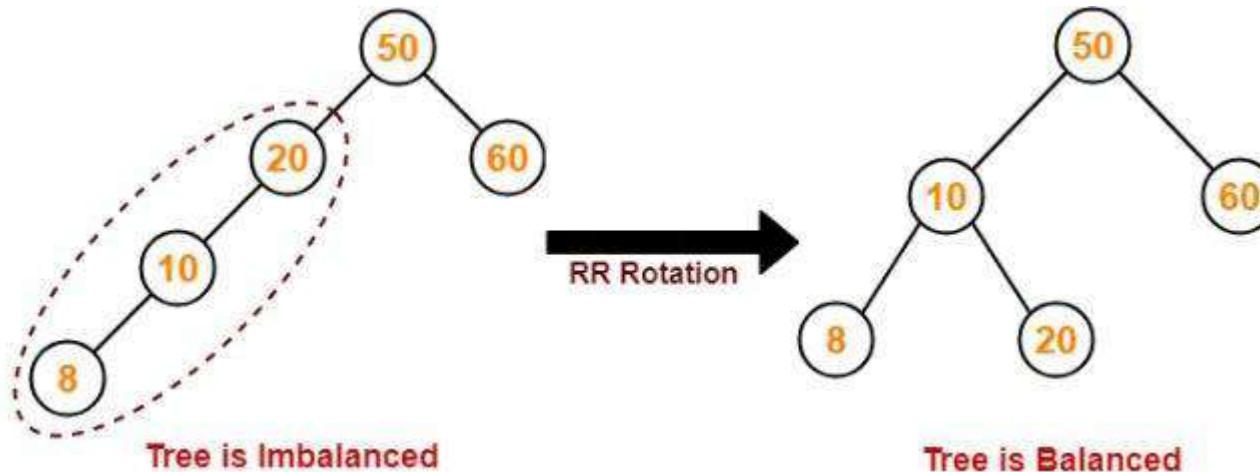
# AVL Tree

- Construct AVL Tree for the following sequence of numbers-  
50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **Step-05: Insert 8**
  - As  $8 < 50$ , so insert 8 in 50's left sub tree.
  - As  $8 < 20$ , so insert 8 in 20's left sub tree.
  - As  $8 < 10$ , so insert 8 in 10's left sub tree.



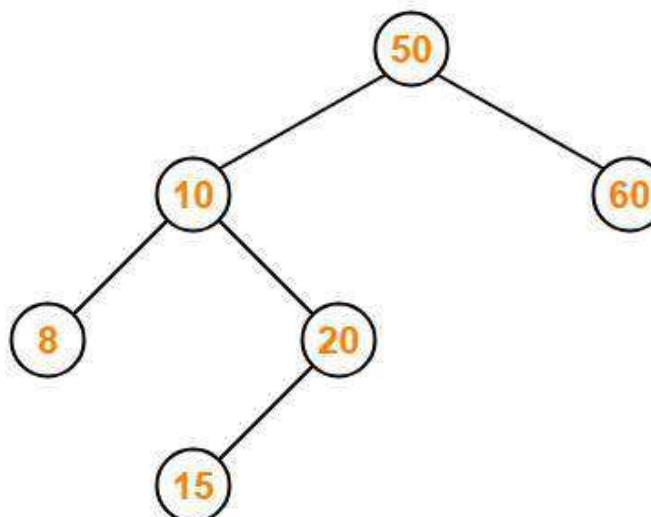
# AVL Tree

- To balance the tree,
- Find the first imbalanced node on the path from the newly inserted node (node 8) to the root node.
- The first imbalanced node is node 20.
- Now, count three nodes from node 20 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.



# AVL Tree

- Construct AVL Tree for the following sequence of numbers-  
50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **Step-06: Insert 15**
  - As  $15 < 50$ , so insert 15 in 50's left sub tree.
  - As  $15 > 10$ , so insert 15 in 10's right sub tree.
  - As  $15 < 20$ , so insert 15 in 20's left sub tree.

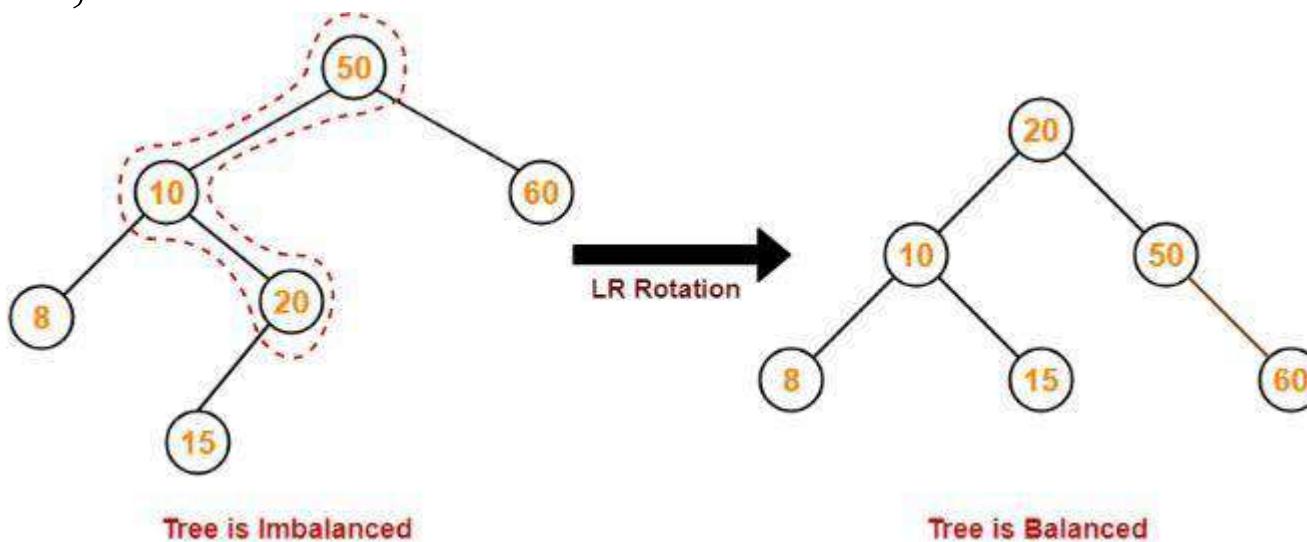


Tree is Imbalanced

# AVL Tree

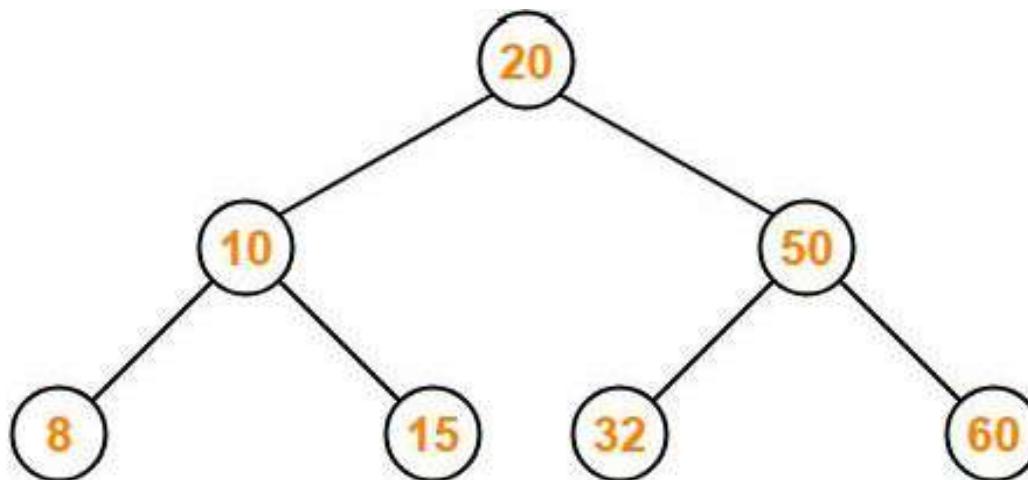
## To balance the tree-

- Find the first imbalanced node on the path from the newly inserted node (node 15) to the root node.
- The first imbalanced node is node 50.
- Now, count three nodes from node 50 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.



# AVL Tree

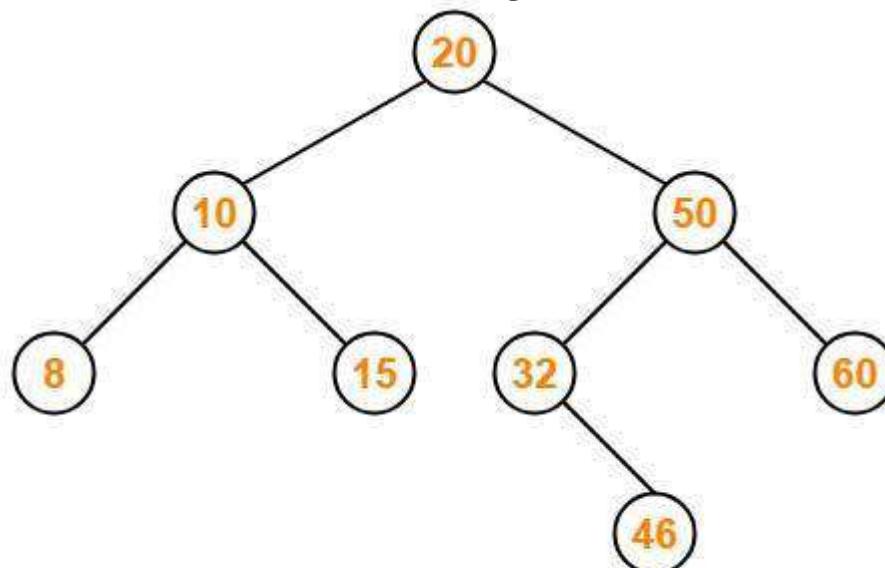
- Construct AVL Tree for the following sequence of numbers-  
50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **Step-07: Insert 32**
  - As 32 > 20, so insert 32 in 20's right sub tree.
  - As 32 < 50, so insert 32 in 50's left sub tree.



Tree is Balanced

# AVL Tree

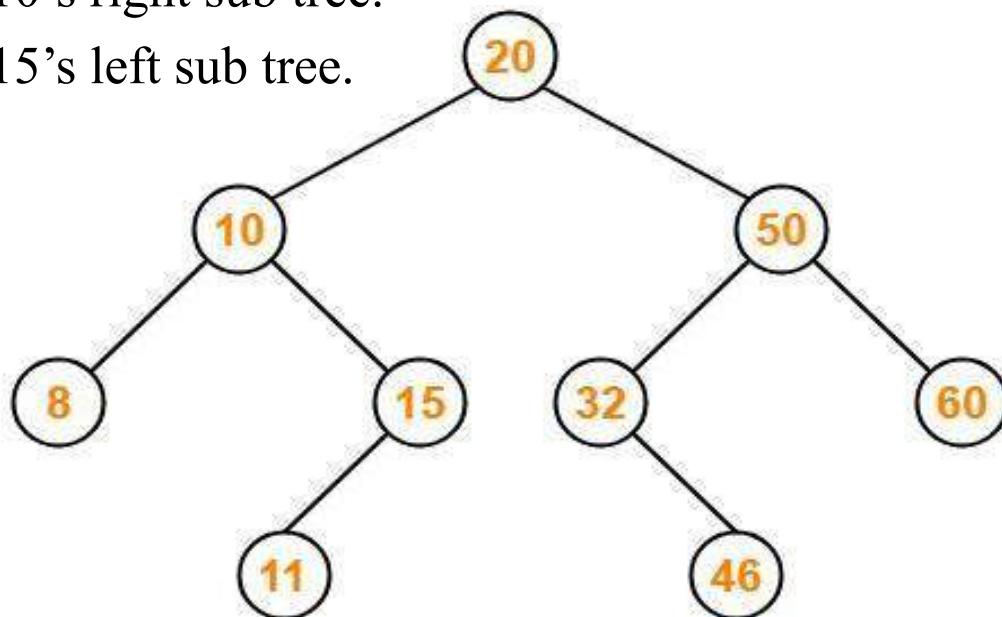
- Construct AVL Tree for the following sequence of numbers-  
50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **Step-08: Insert 46**
  - As 46 > 20, so insert 46 in 20's right sub tree.
  - As 46 < 50, so insert 46 in 50's left sub tree.
  - As 46 > 32, so insert 46 in 32's right sub tree.



Tree is Balanced

# AVL Tree

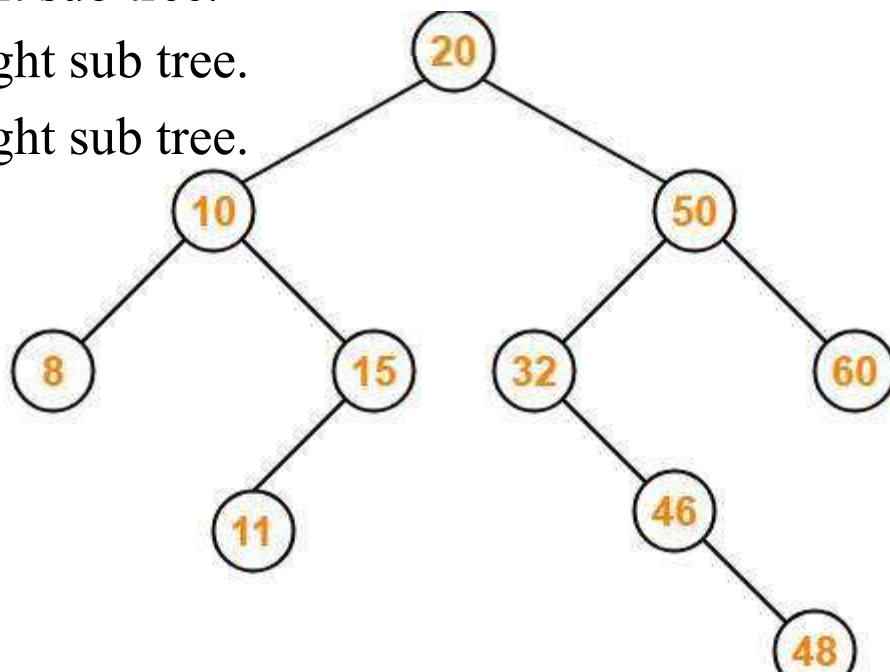
- Construct AVL Tree for the following sequence of numbers-  
50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **Step-09: Insert 11**
  - As 11 < 20, so insert 11 in 20's left sub tree.
  - As 11 > 10, so insert 11 in 10's right sub tree.
  - As 11 < 15, so insert 11 in 15's left sub tree.



Tree is Balanced

# AVL Tree

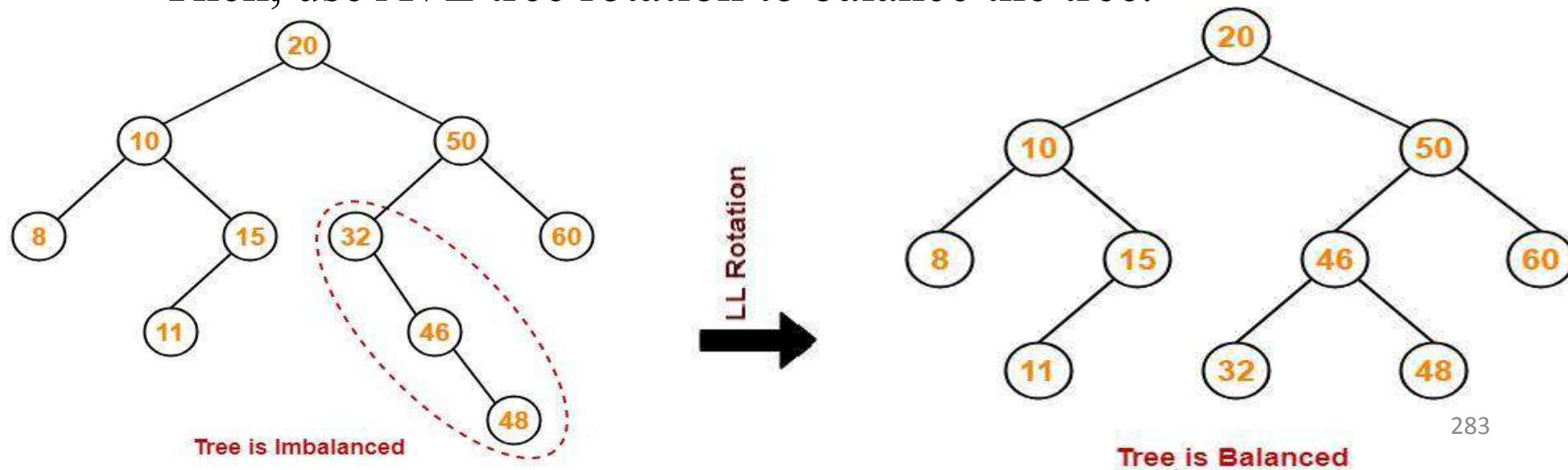
- Construct AVL Tree for the following sequence of numbers-  
50 , 20 , 60 , 10 , 8 , 15 , 32 , 46 , 11 , 48
- **Step-10: Insert 48**
  - As 48 > 20, so insert 48 in 20's right sub tree.
  - As 48 < 50, so insert 48 in 50's left sub tree.
  - As 48 > 32, so insert 48 in 32's right sub tree.
  - As 48 > 46, so insert 48 in 46's right sub tree.



# AVL Tree

## To balance the tree:

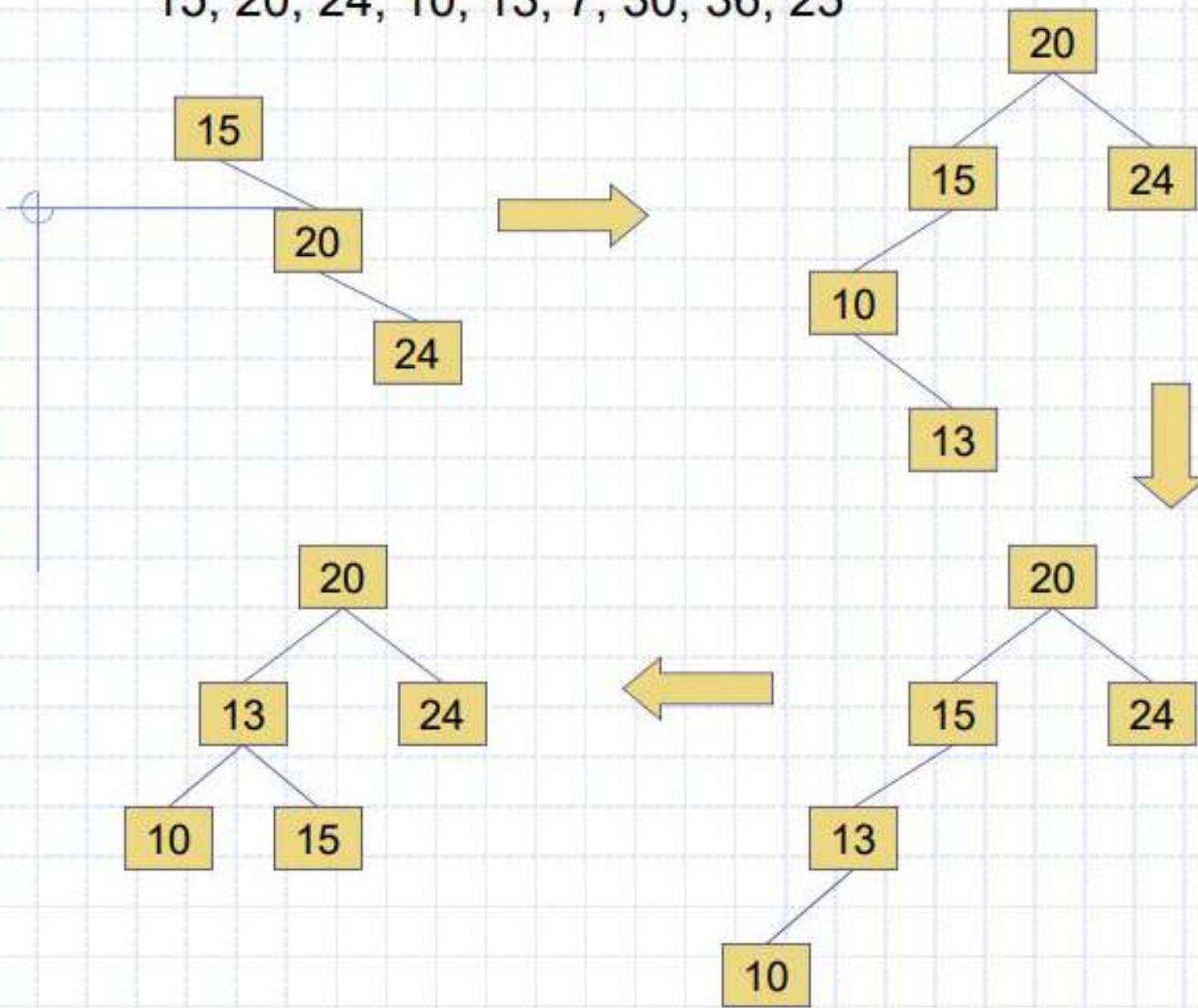
- Find the first imbalanced node on the path from the newly inserted node (node 48) to the root node.
- The first imbalanced node is node 32.
- Now, count three nodes from node 32 in the direction of leaf node.
- Then, use AVL tree rotation to balance the tree.



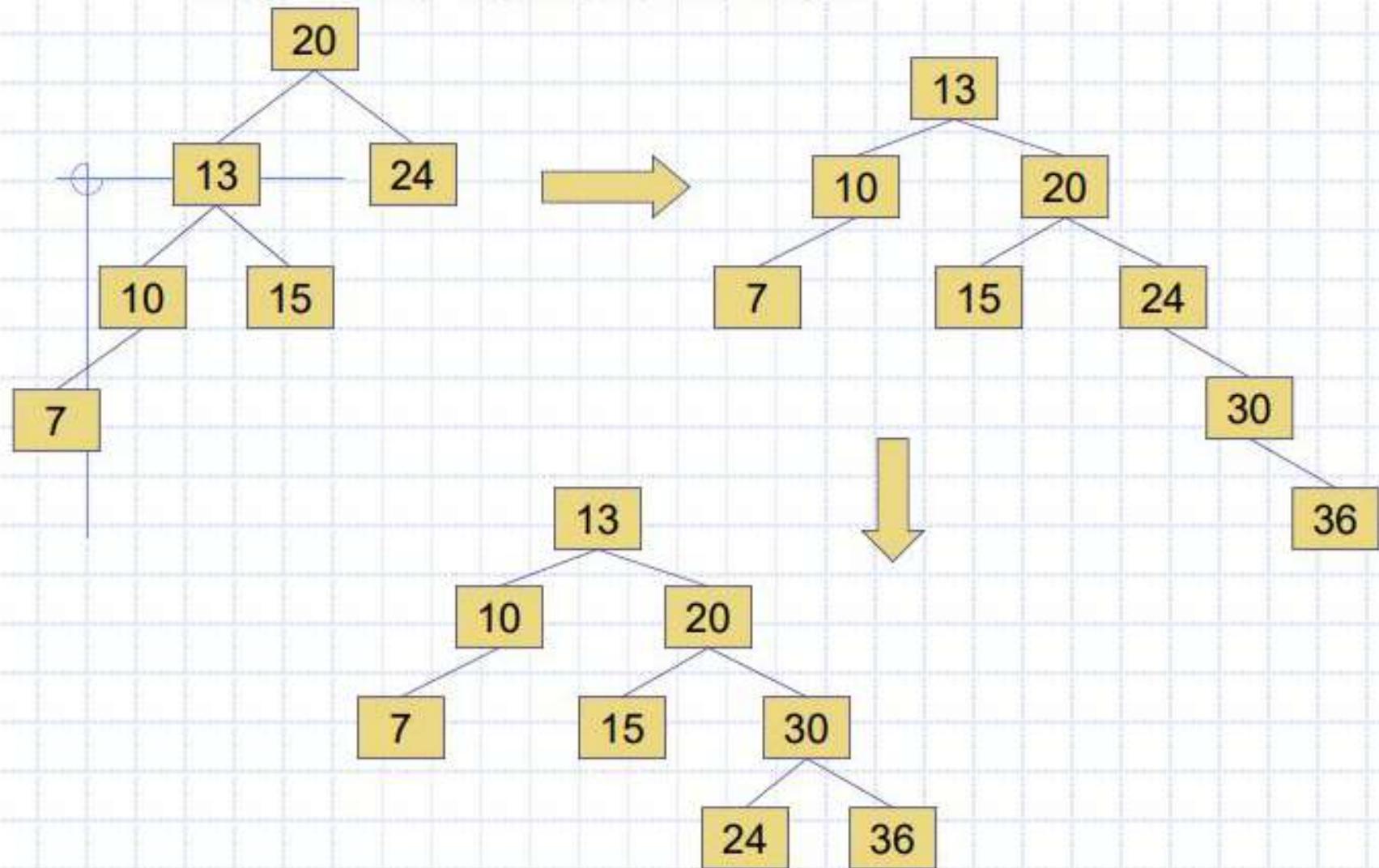
## In Class Exercises

- Build an AVL tree with the following values:  
15, 20, 24, 10, 13, 7, 30, 36, 25

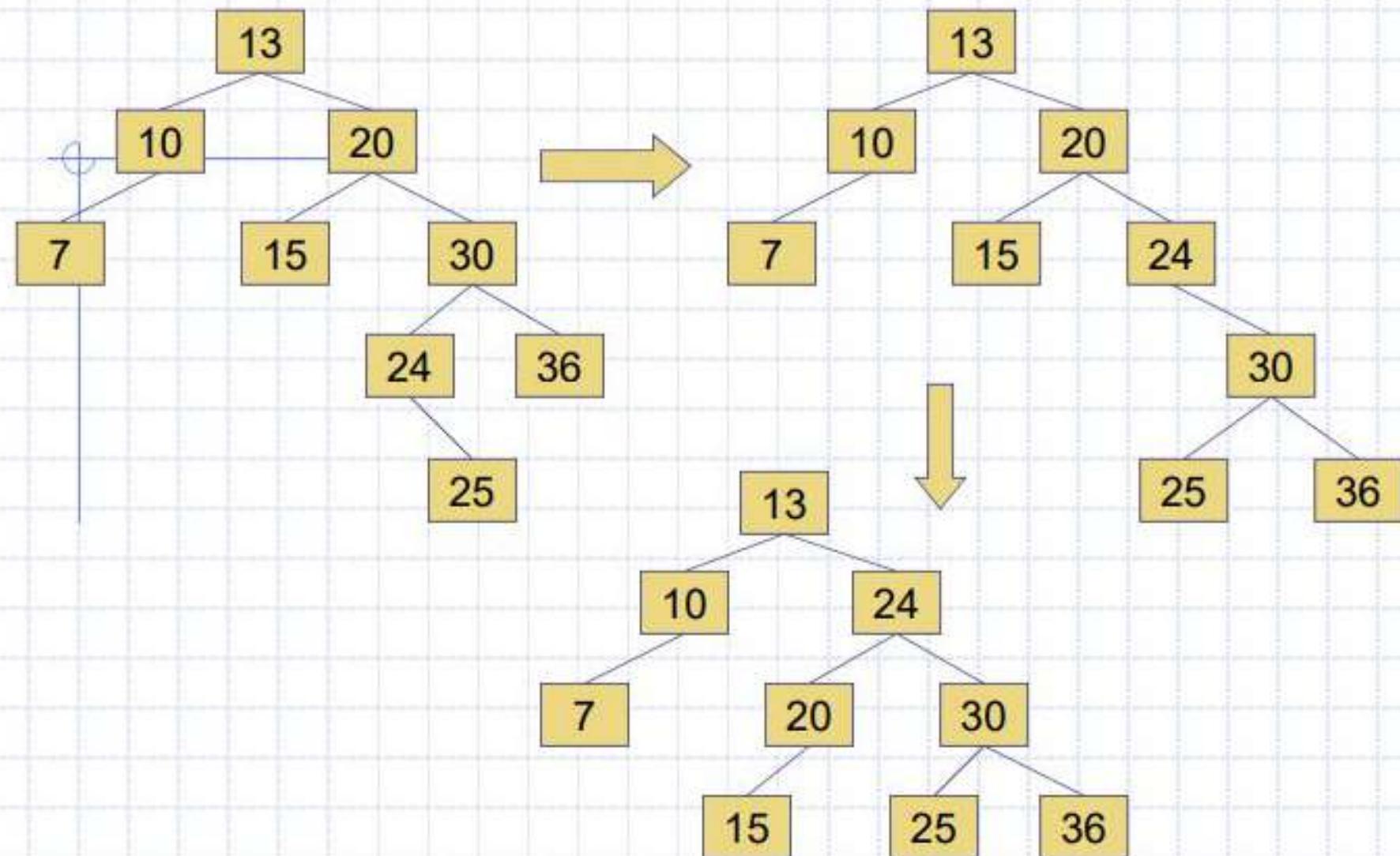
15, 20, 24, 10, 13, 7, 30, 36, 25



15, 20, 24, 10, 13, 7, 30, 36, 25

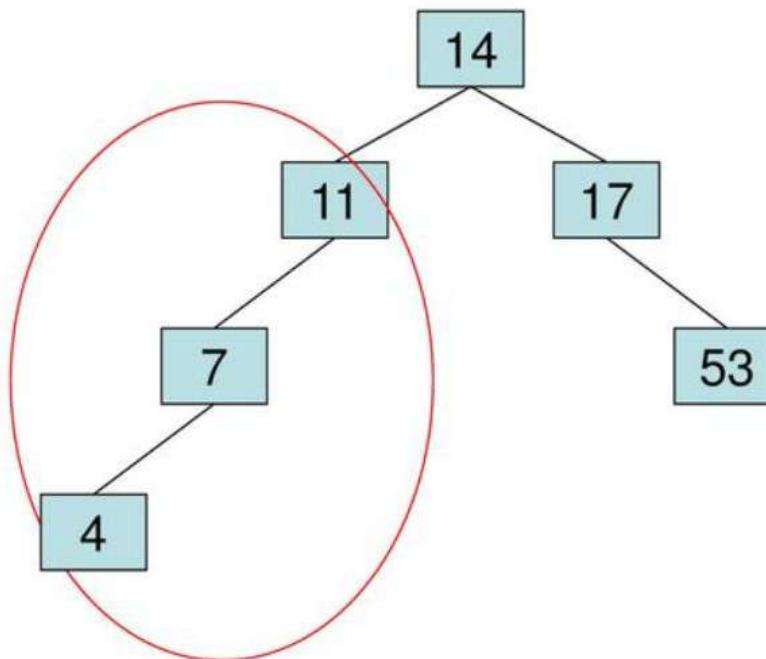


15, 20, 24, 10, 13, 7, 30, 36, 25



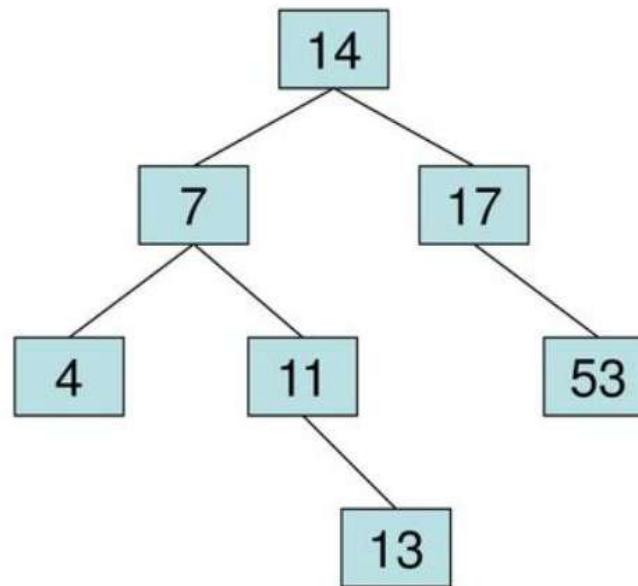
### AVL Tree Example:

- Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree



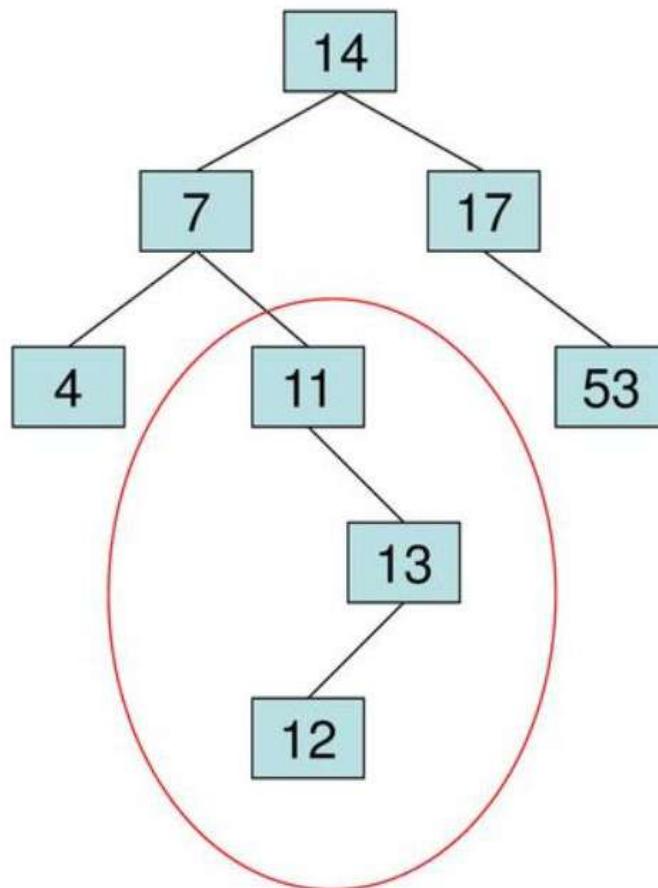
## AVL Tree Example:

- Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree



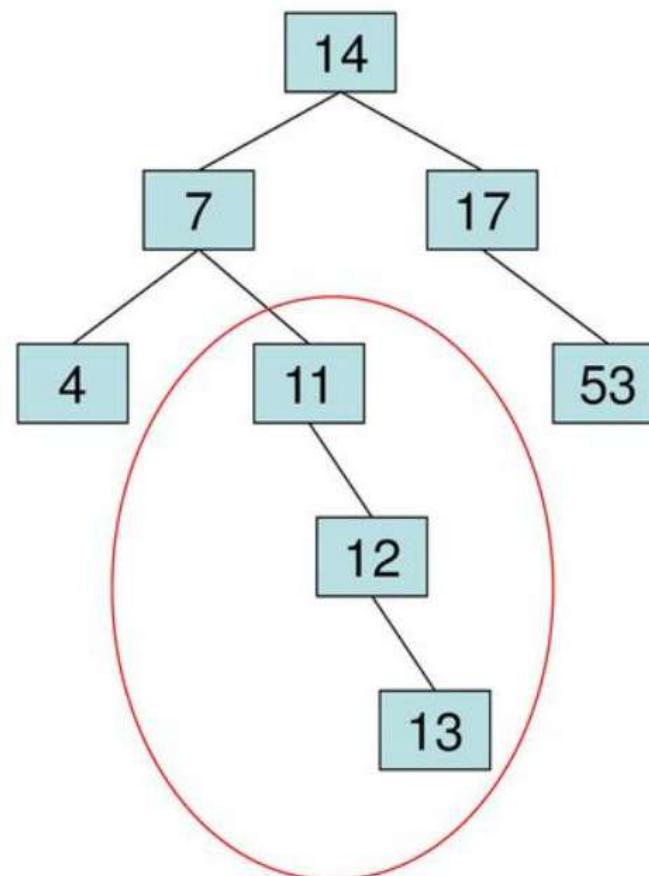
## AVL Tree Example:

- Now insert 12



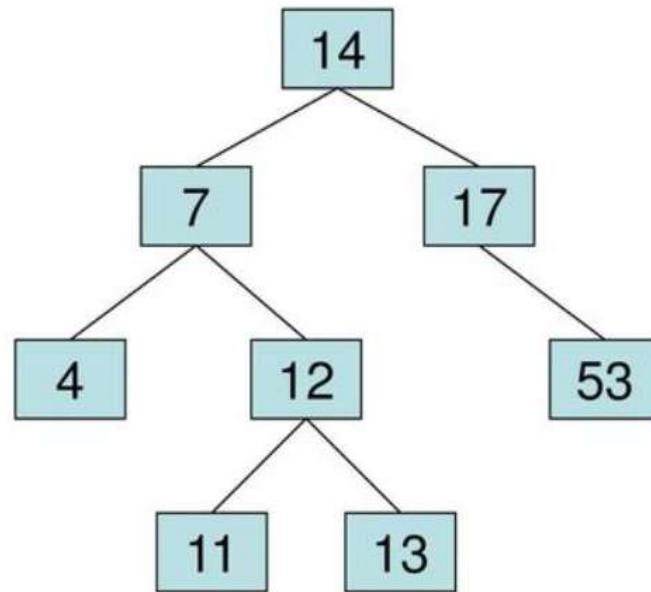
## AVL Tree Example:

- Now insert 12



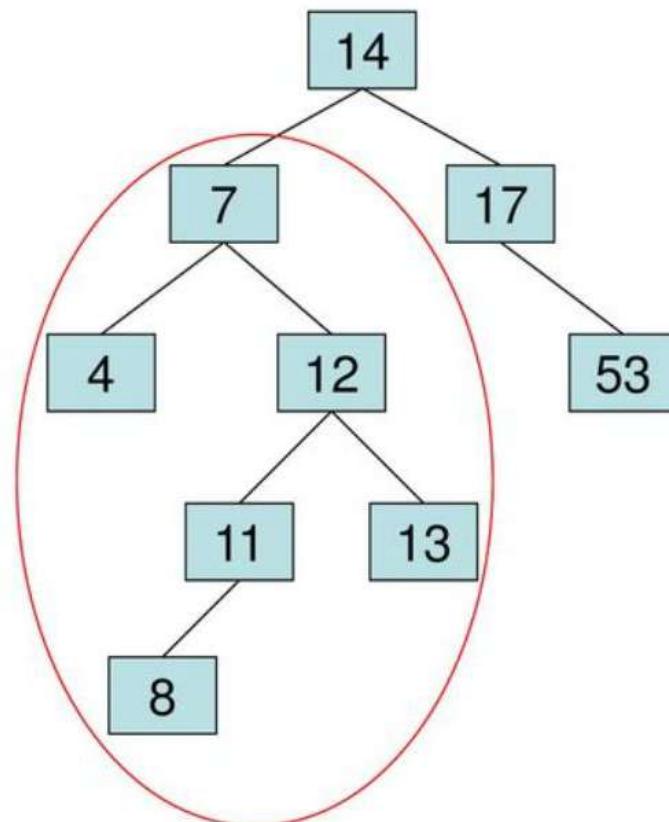
## AVL Tree Example:

- Now the AVL tree is balanced.



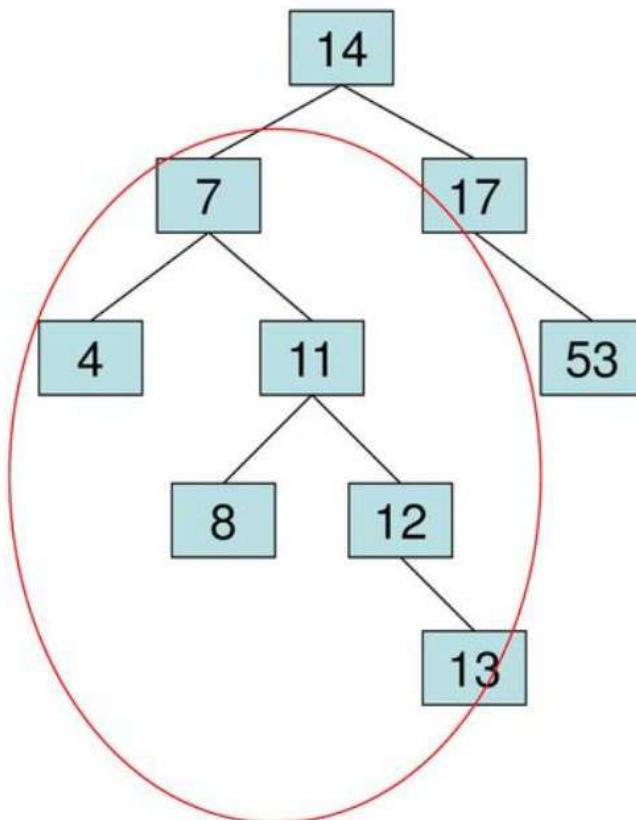
## AVL Tree Example:

- Now insert 8



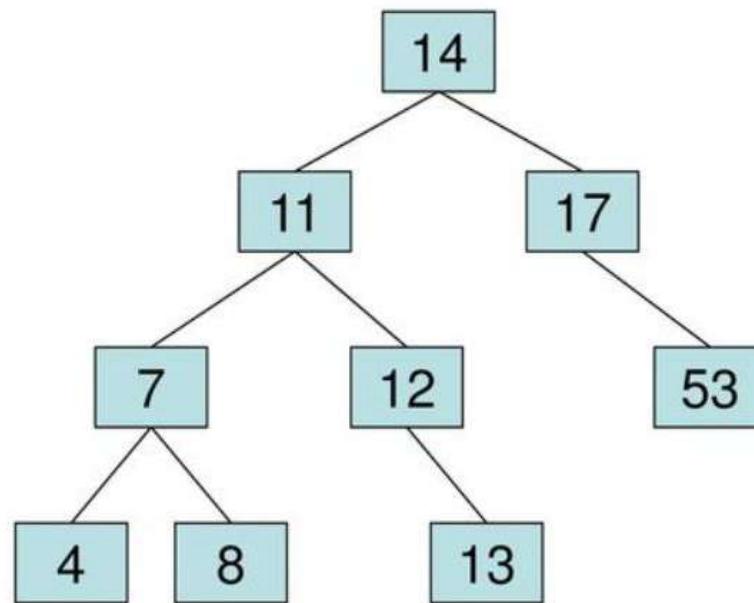
## AVL Tree Example:

- Now insert 8



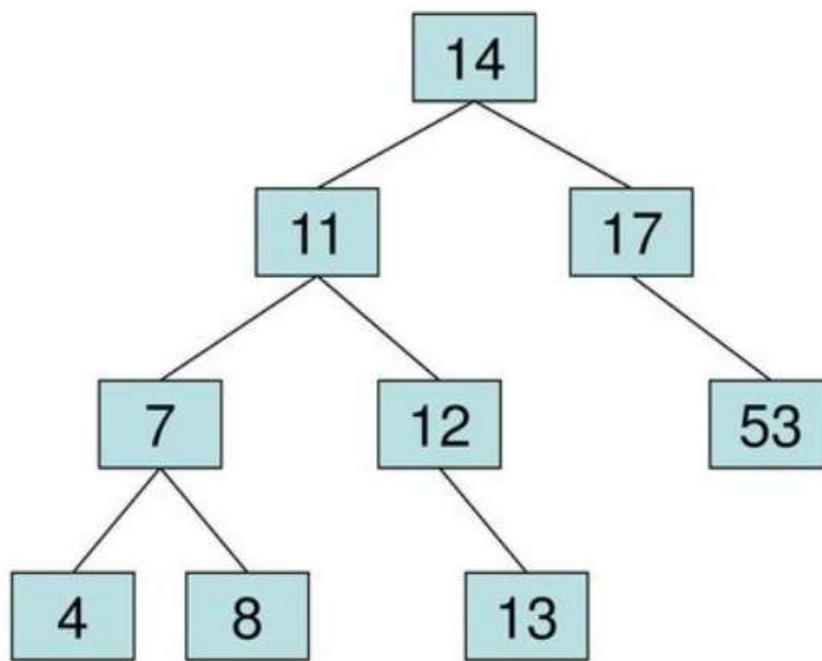
## AVL Tree Example:

- Now the AVL tree is balanced.



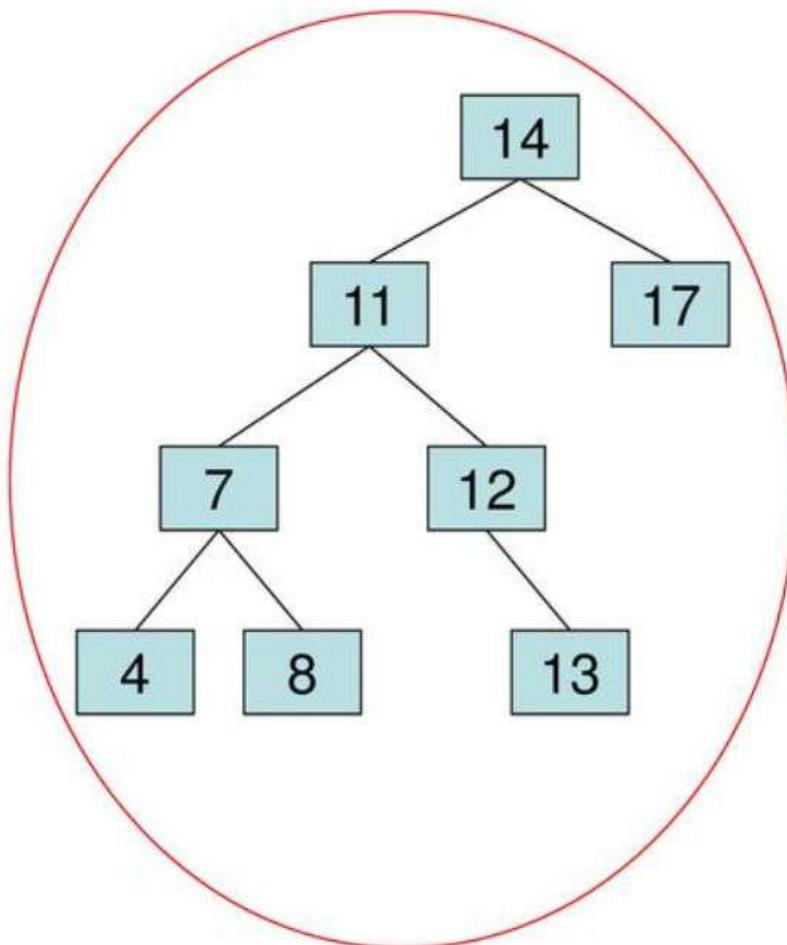
## AVL Tree Example:

- Now remove 53



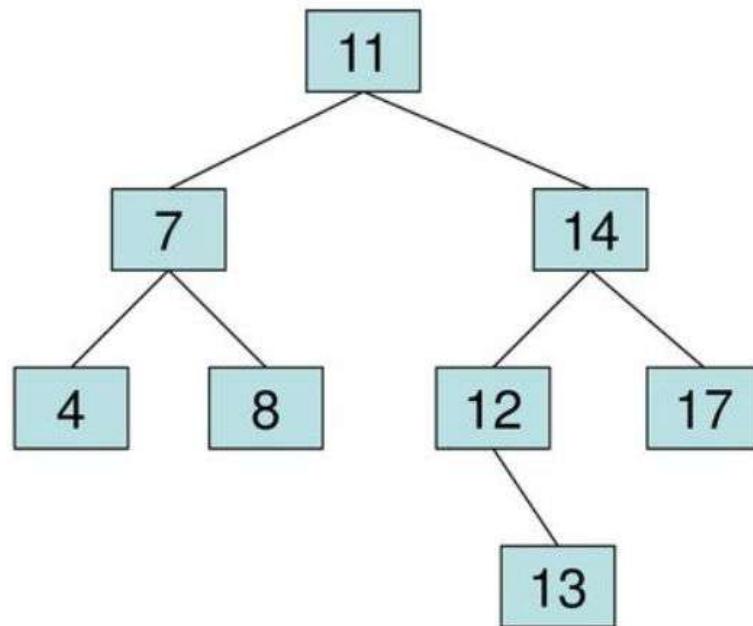
## AVL Tree Example:

- Now remove 53, unbalanced



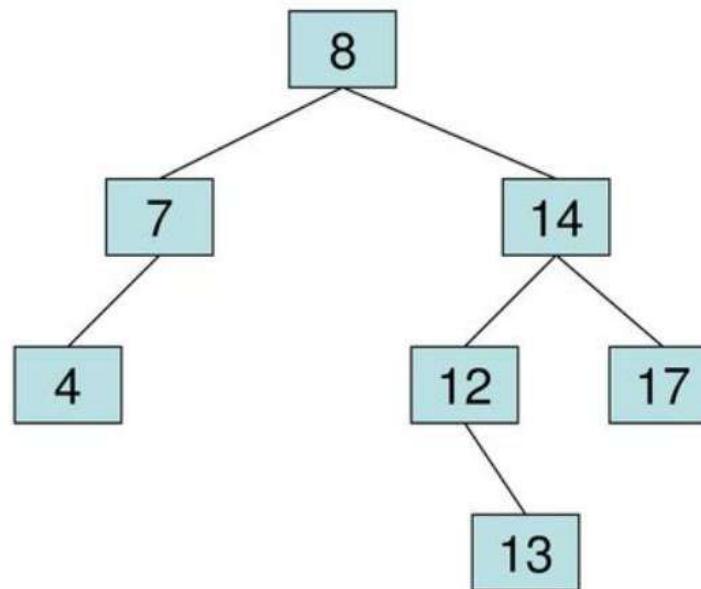
## AVL Tree Example:

- Balanced! Remove 11



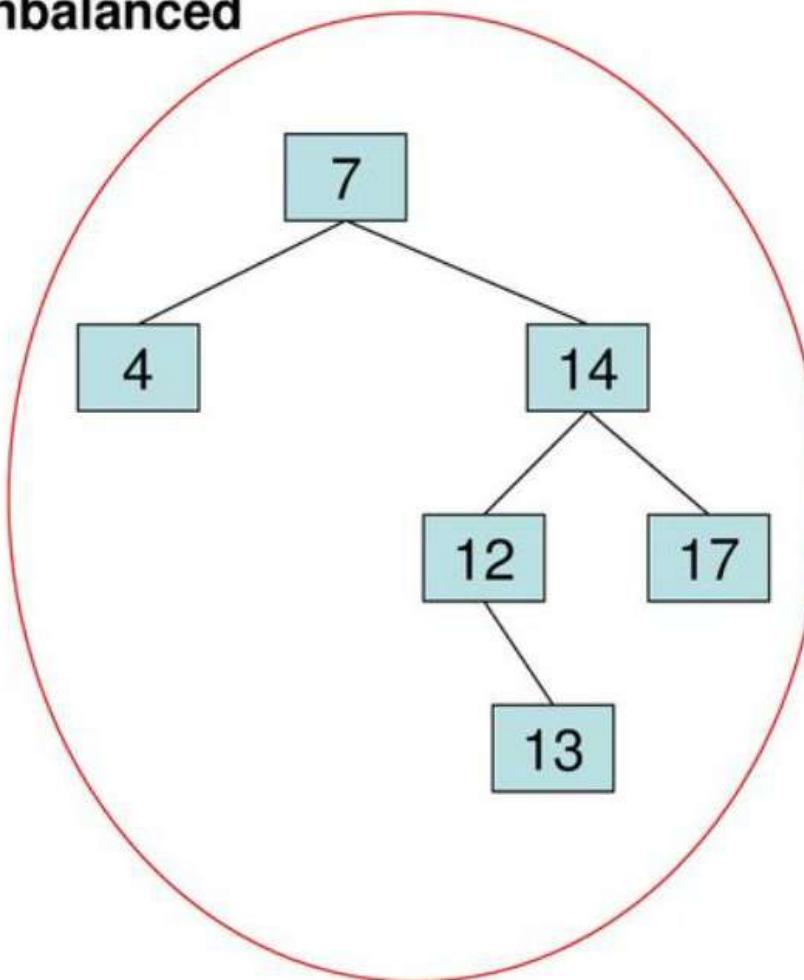
### AVL Tree Example:

- Remove 11, replace it with the largest in its left branch



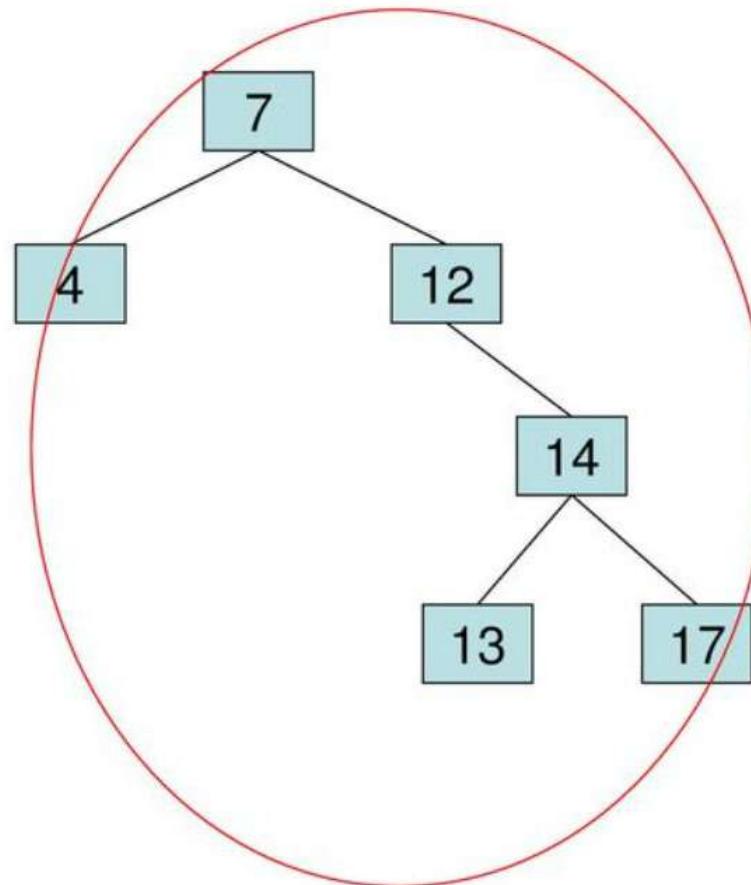
## AVL Tree Example:

- Remove 8, unbalanced



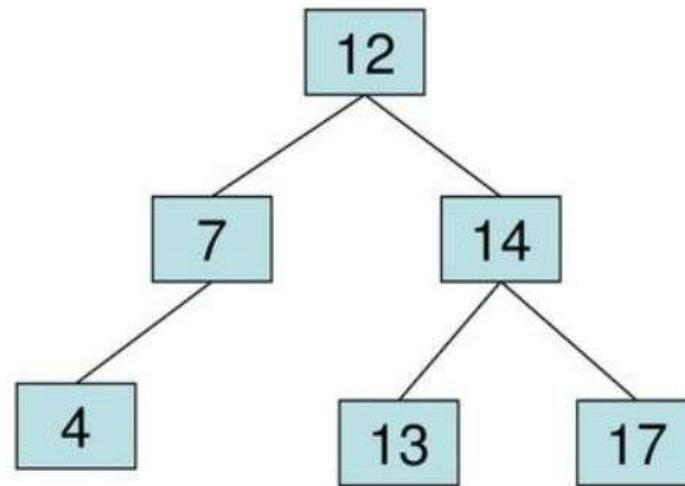
## AVL Tree Example:

- Remove 8, unbalanced



## AVL Tree Example:

- **Balanced!!**



**Construct AVL tree for the following data**

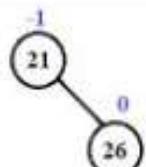
**21,26,30,9,4,14,28,18,15,10,2,3,7**

Step 1 - Insert 21



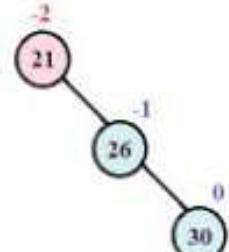
Tree is Balanced

Step 2 - Insert 26



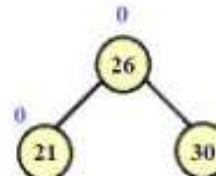
Tree is Balanced

Step 3 - Insert 30



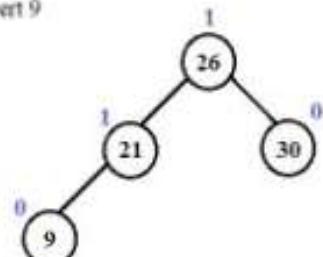
Tree is Not Balanced, Need a Rotation

LL Rotation



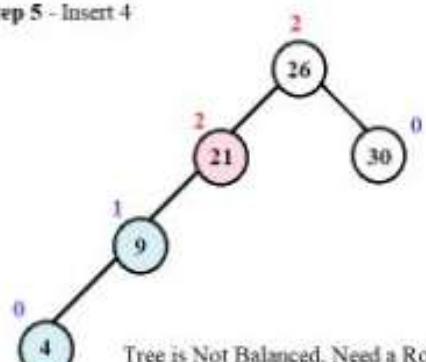
Tree is Balanced

Step 4 - Insert 9



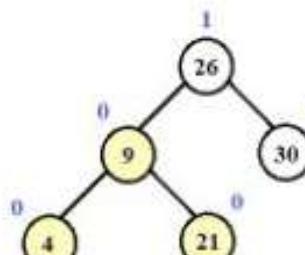
Tree is Balanced

Step 5 - Insert 4



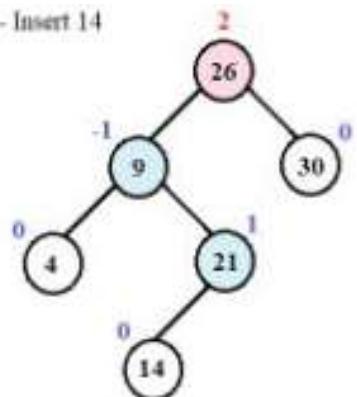
Tree is Not Balanced, Need a Rotation

RR Rotation



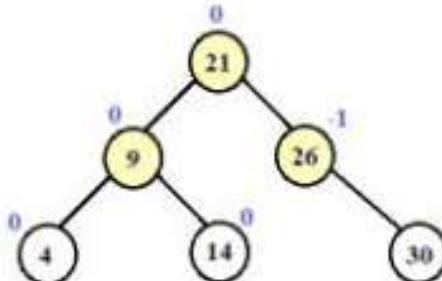
Tree is Balanced

Step 6 - Insert 14



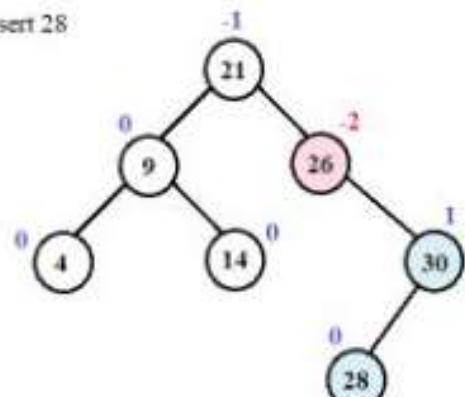
Tree is Not Balanced, Need a Rotation

LR Rotation



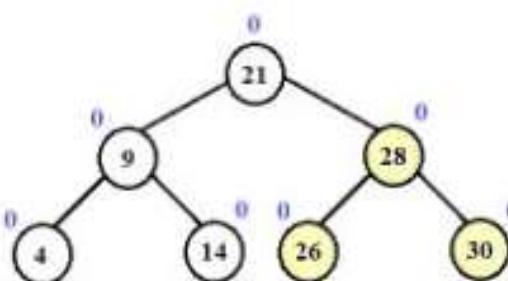
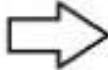
Tree is Balanced

Step 7 - Insert 28



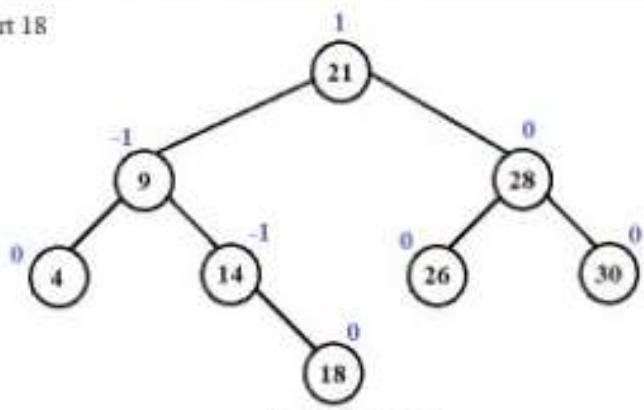
Tree is Not Balanced, Need a Rotation

RL Rotation



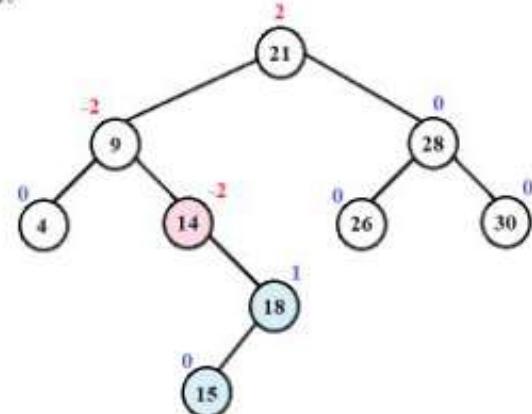
Tree is Balanced

Step 8 - Insert 18

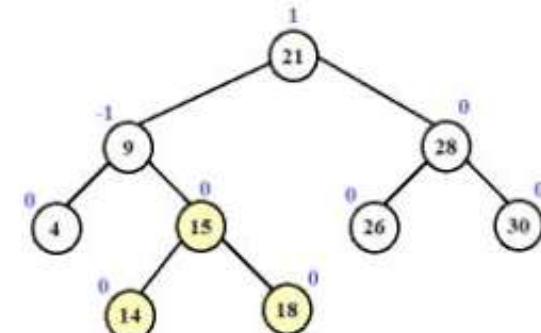
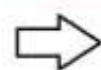


Tree is Balanced

Step 9 - Insert 15

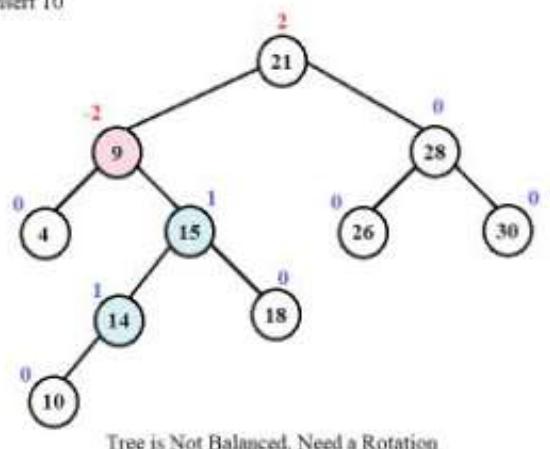


RL Rotation

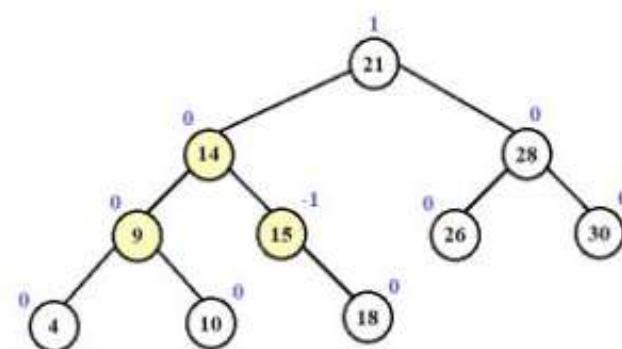
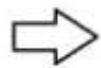


Tree is Balanced

Step 10 - Insert 10

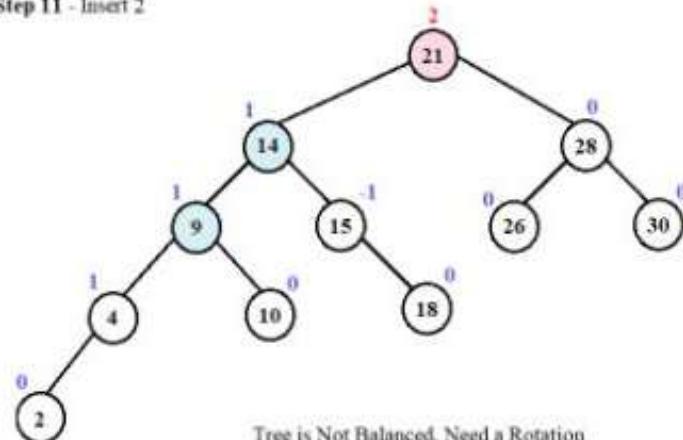


RL Rotation

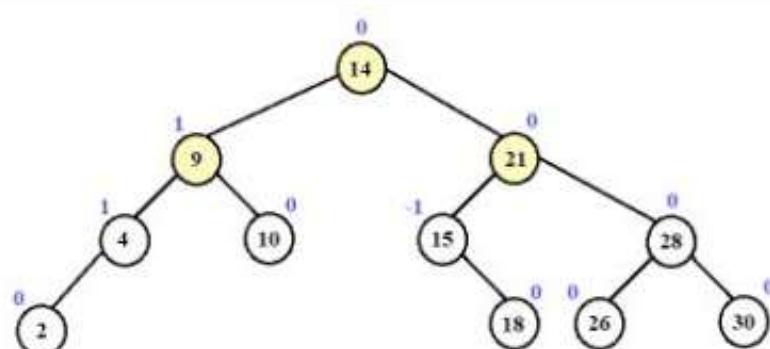
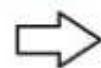


Tree is Balanced

Step 11 - Insert 2

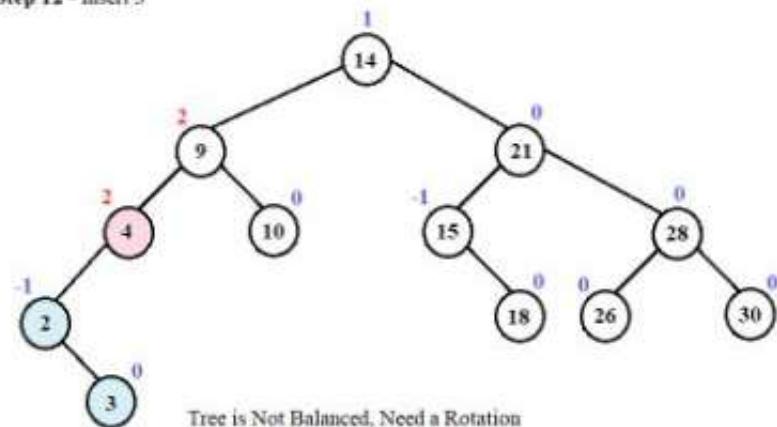


RR Rotation

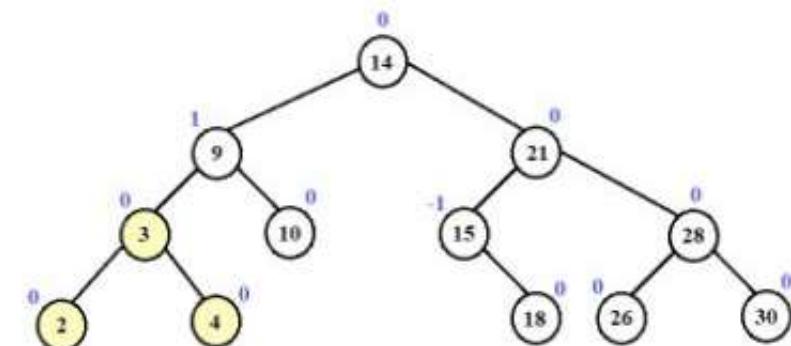


Tree is Balanced

Step 12 - Insert 3

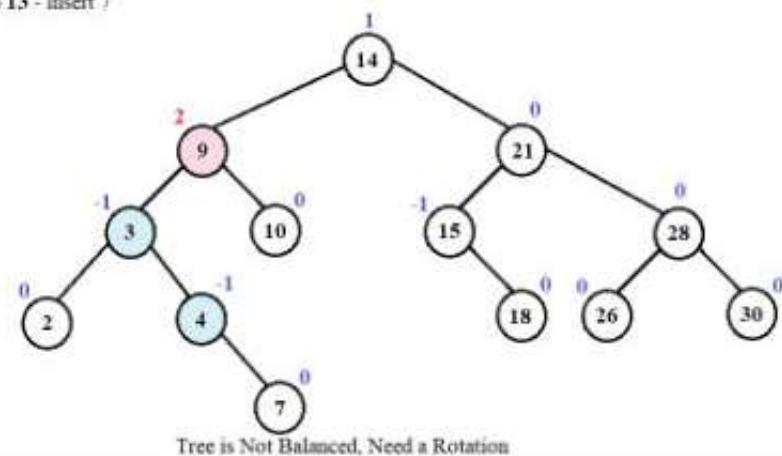


LR Rotation

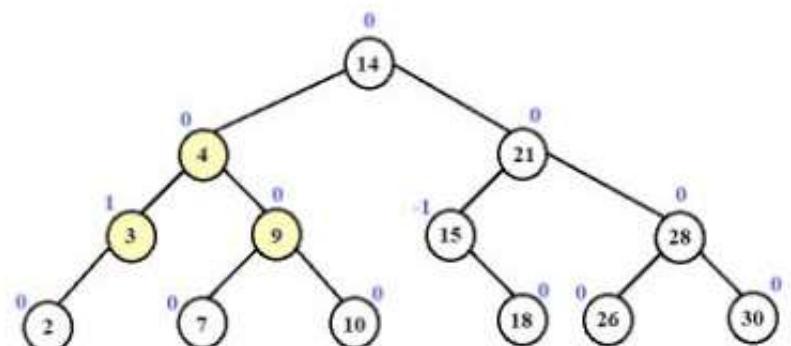


Tree is Balanced

Step 13 - Insert 7



LR Rotation



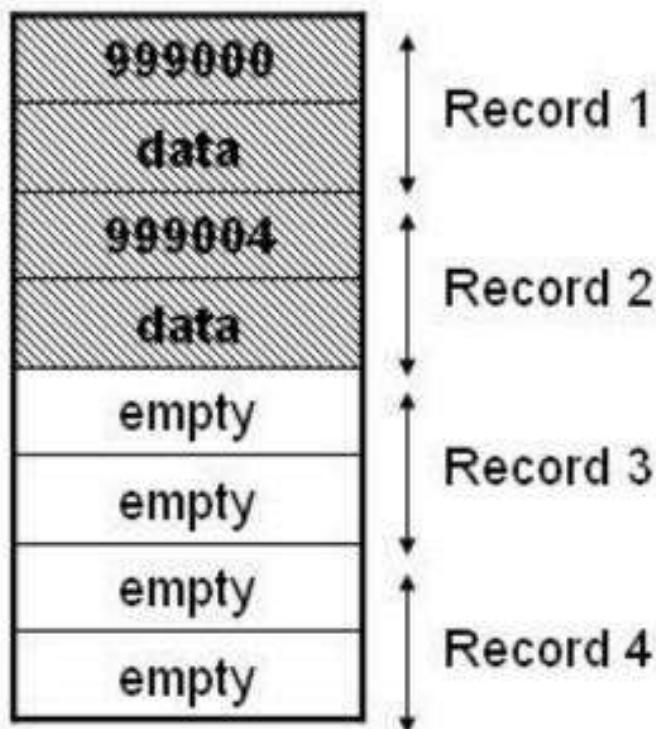
Tree is Balanced

# B-Trees

- Disk Storage
- What is a multiway tree?
- What is a B-tree?
- Why B-trees?
- Comparing B-trees and AVL-trees
- Searching a B-tree
- Insertion in a B-tree
- Deletion in a B-tree

# Disk Storage

- Data is stored on disk (i.e., secondary memory) in blocks.
- A block is the smallest amount of data that can be accessed on a disk.
- Each block has a fixed number of bytes – typically 512, 1024, 2048, 4096 or 8192 bytes
- Each block may hold many data records.

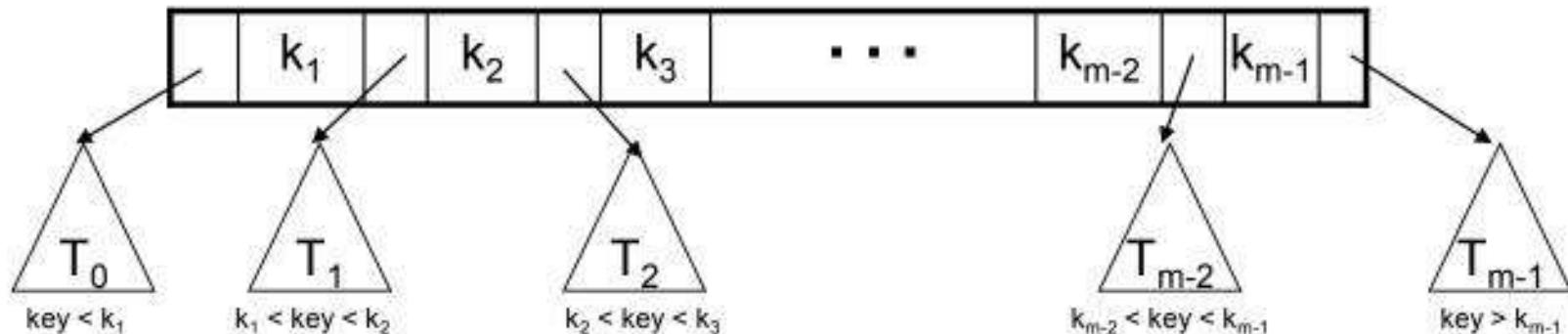


## Motivation for studying Multi-way and B-trees

- A disk access is very expensive compared to a typical computer instruction (mechanical limitations) - One disk access is worth about 200,000 instructions.
- Thus, When data is too large to fit in main memory the number of disk accesses becomes important.
- Many algorithms and data structures that are efficient for manipulating data in primary memory are not efficient for manipulating large data in secondary memory because they do not minimize the number of disk accesses.
- For example, AVL trees are not suitable for representing huge tables residing in secondary memory.
- The height of an AVL tree increases, and hence the number of disk accesses required to access a particular record increases, as the number of records increases.

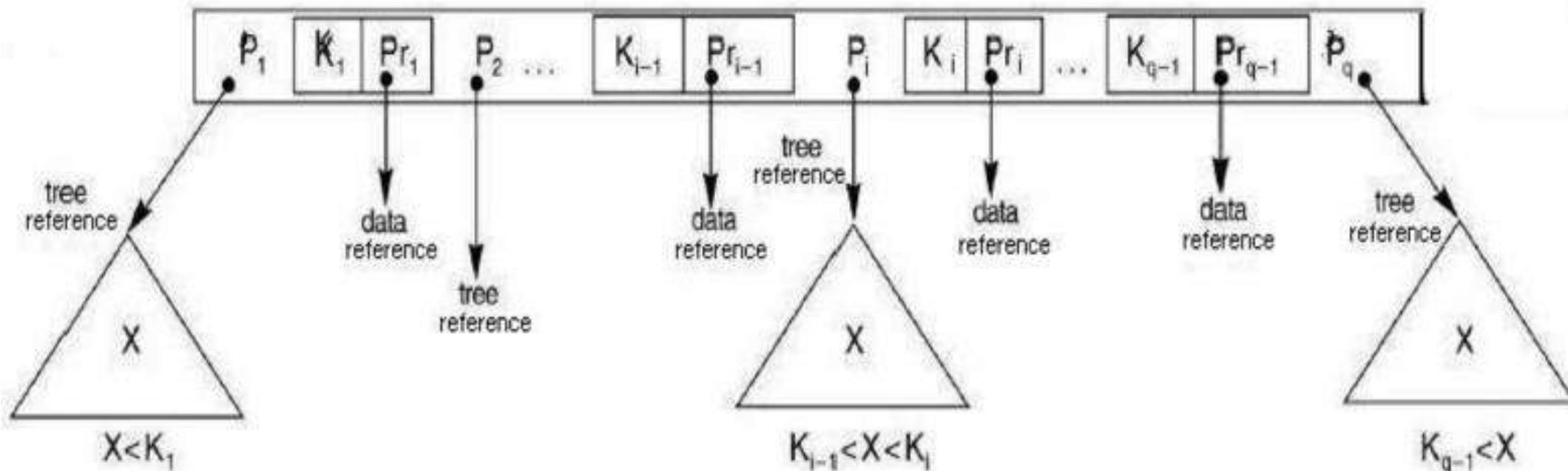
## What is a Multi-way tree?

- A multi-way (or  $m$ -way) search tree of order  $m$  is a tree in which
  - Each node has at-most  $m$  subtrees, where the subtrees may be empty.
  - Each node consists of at least 1 and at most  $m-1$  distinct keys
  - The keys in each node are sorted in increasing order.



- The keys and subtrees of a non-leaf node are ordered as:
  - $T_0, k_1, T_1, k_2, T_2, \dots, k_{m-1}, T_{m-1}$  such that:
    - All keys in subtree  $T_0$  are less than  $k_1$ .
    - All keys in subtree  $T_i$ ,  $1 \leq i \leq m-2$ , are greater than  $k_i$  but less than  $k_{i+1}$ .
    - All keys in subtree  $T_{m-1}$  are greater than  $k_{m-1}$ .

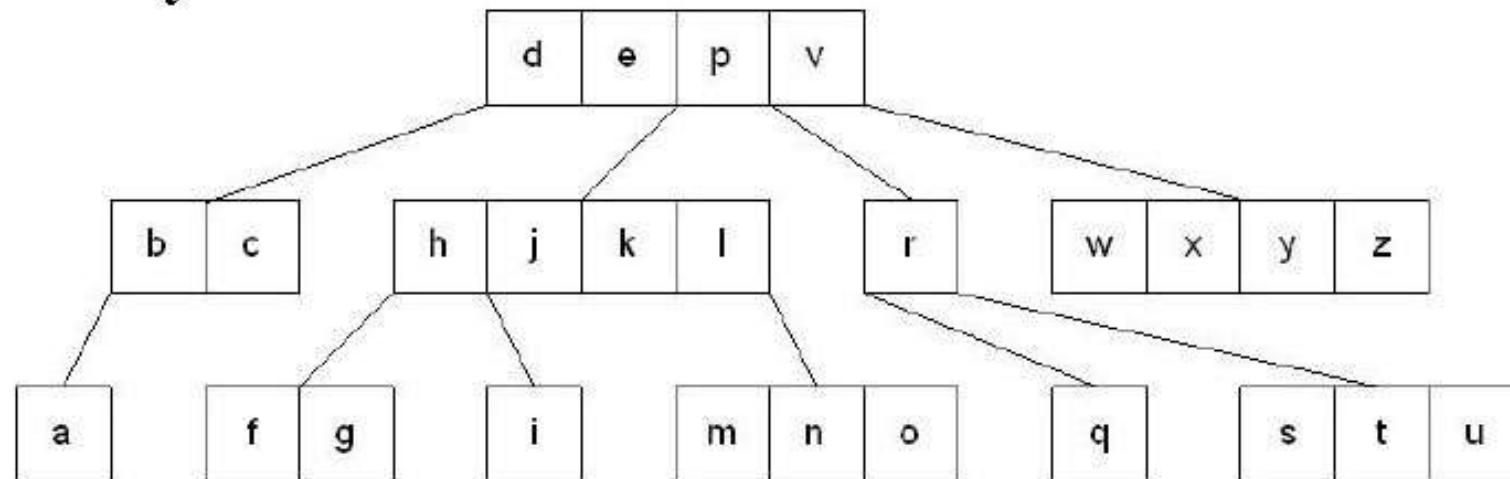
## The node structure of a Multi-way tree



- Note:
  - Corresponding to each key there is a data reference that refers to the data record for that key in secondary memory.
  - In our representations we will omit the data references.
  - If an array is used to represent a node, an additional field storing the number of keys in a node is required in each node.

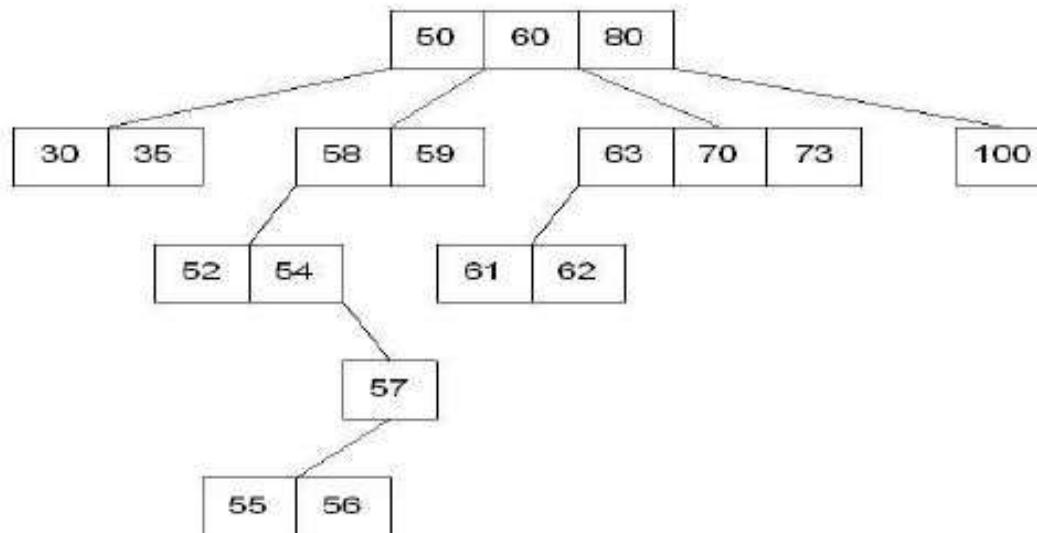
# Introduction to M-Way Search Trees

- A **multiway tree** is a tree that can have more than two children. A **multiway tree of order m** (or an **m-way tree**) is one in which a tree can have m children.
- As with the other trees that have been studied, the nodes in an m-way tree will be made up of key fields, in this case m-1 key fields, and pointers to children.
- Multiday tree of order 5



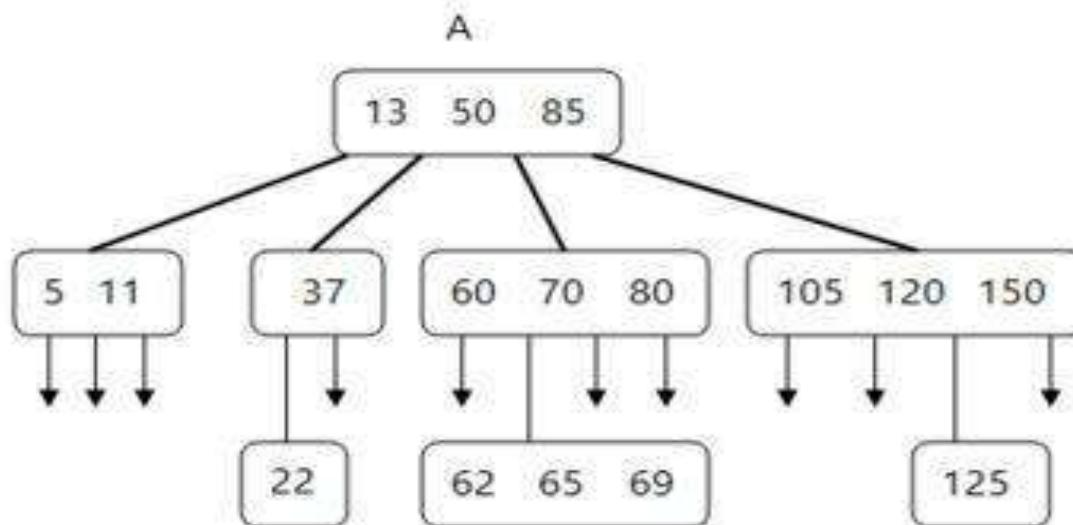
# Properties of M-way Search Trees

- m-way search tree is a m-way tree in which:
  - i. Each node has m children and m-1 key fields
  - ii. The keys in each node are in ascending order.
  - iii. The keys in the first i children are smaller than the ith key
  - iv. The keys in the last m-i children are larger than the ith key
- 4-way search tree



# Multiway Trees

- Multiway Search Trees allow nodes to store multiple child nodes (greater than 2).
- These differ from binary search trees which can only have a maximum of 2 nodes.



# Multiway Trees

- **Characteristics**

- Nodes may carry multiple keys.
- Each node may have **N** number of children
- Each node maintains **N-1** search keys
- The tree maintains all leaves at the same level

# Multiway Trees

- **Operations**

- Search: A path is traced starting at the root. The nodes are traversed and a pointer is positioned on the key value being searched. If the key is not found, it returns a **search miss**. If the key is found, it returns a **search hit**.
- Insert: The pointer searches to make sure a key does not exist. It then creates a link adding the key to the appropriate node.

# Multiway Trees

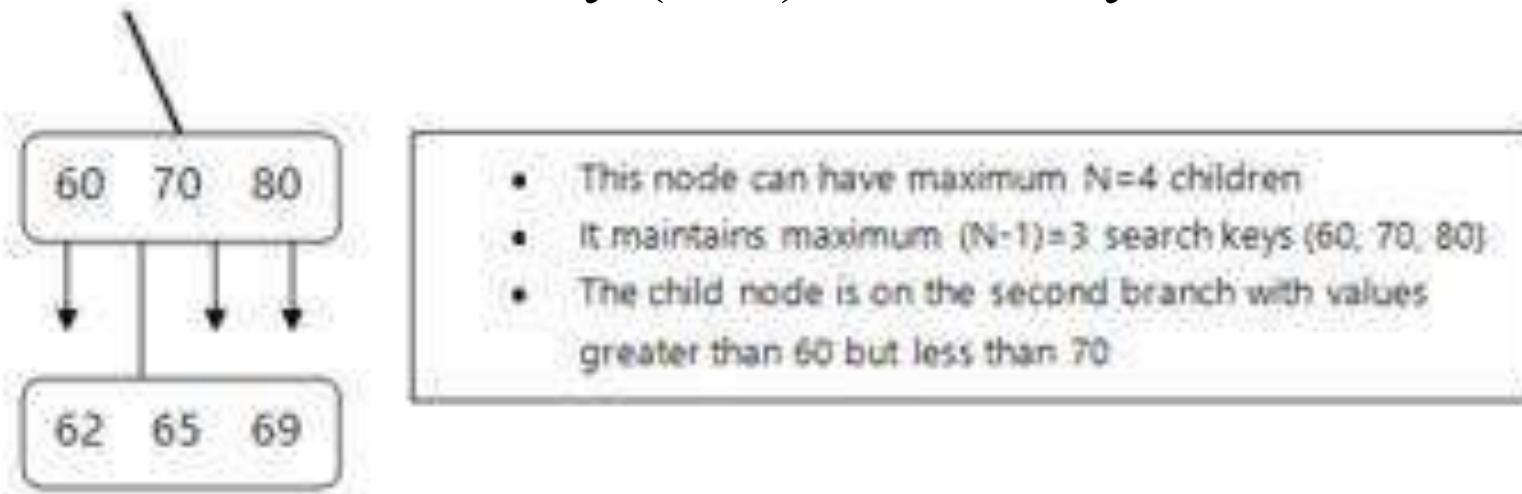
- **2-3-4 Trees**

- **2-3-4 trees** are a type of Multiway search tree.  
Each node can hold a maximum of 3 search keys and can hold 2, 3 or 4 child nodes.
- All leaves are maintained at the same level. 2-3-4 trees are **self-balancing** structures, meaning they rearrange themselves if the structure goes off balance after an insert or delete operation.

# Multiway Trees

- **2-3-4 Trees Characteristics**

- 2-3-4 trees can carry multiple child nodes.
- Each node maintains  $N$  child nodes where  $N$  is equal to 2, 3 or 4 child nodes.
- Each node can carry  $(N-1)$  search keys.



# Multiway

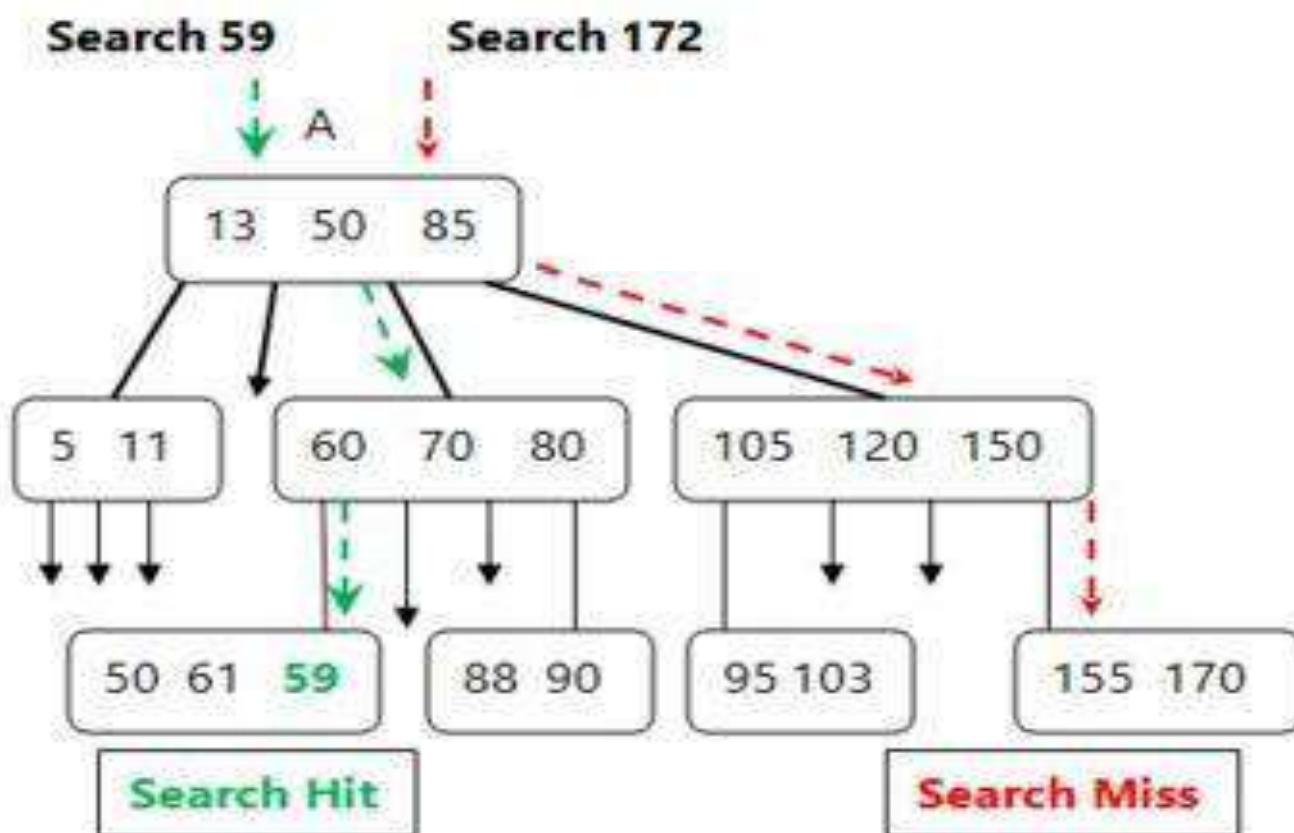
## Trees

- **2-3-4 Trees Operations**

- Search: With 2-3-4 trees, searches commence at the root and traverse each node until right node is found.
- A sequential search is done within the node to locate the correct key value. If the value is found, it returns a **search hit**. If the value is not found, it returns a **search miss**.
- For example, in Figure 3, we search for key 59 and key 172.

# Multiway Trees

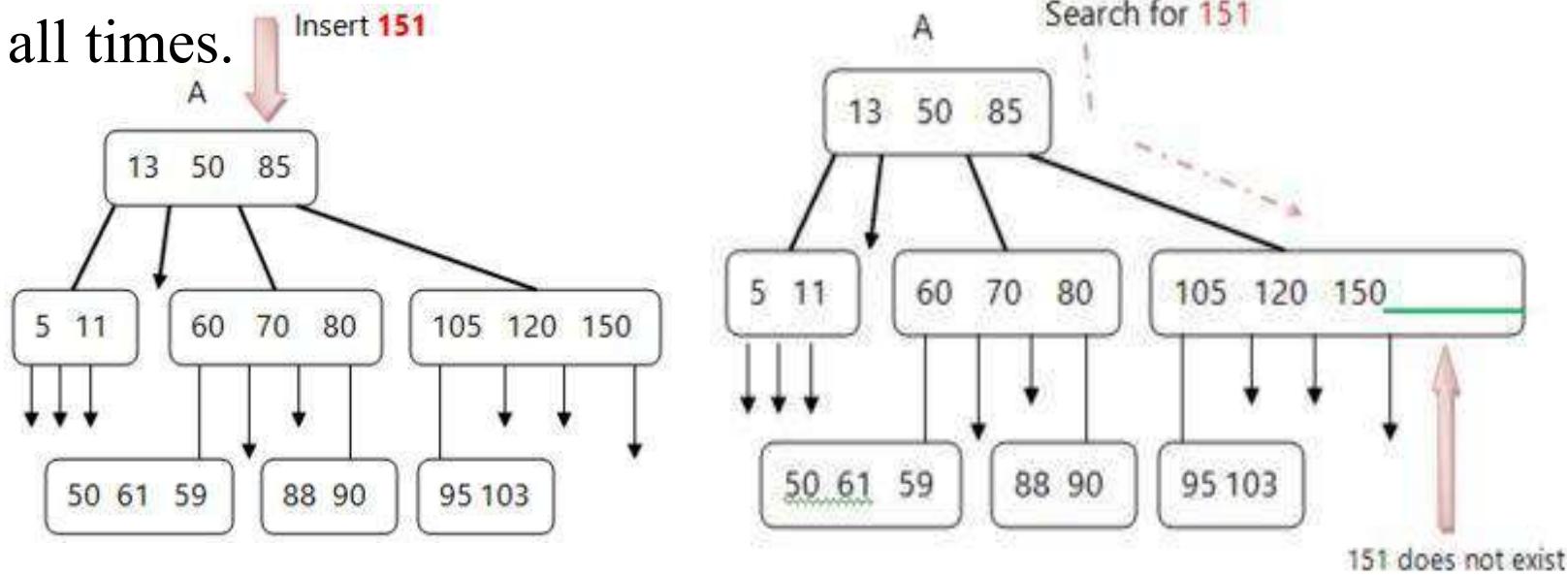
- 2-3-4 Trees Operations



# Multiway Trees

- 2-3-4 Trees Operations

- Insert: The tree is first searched to ensure that the key value does not exist.
- If it doesn't, a link is created in the appropriate node and the search key is inserted.
- Note that 2-3-4 tree characteristics must be maintained at all times.



# Multiway

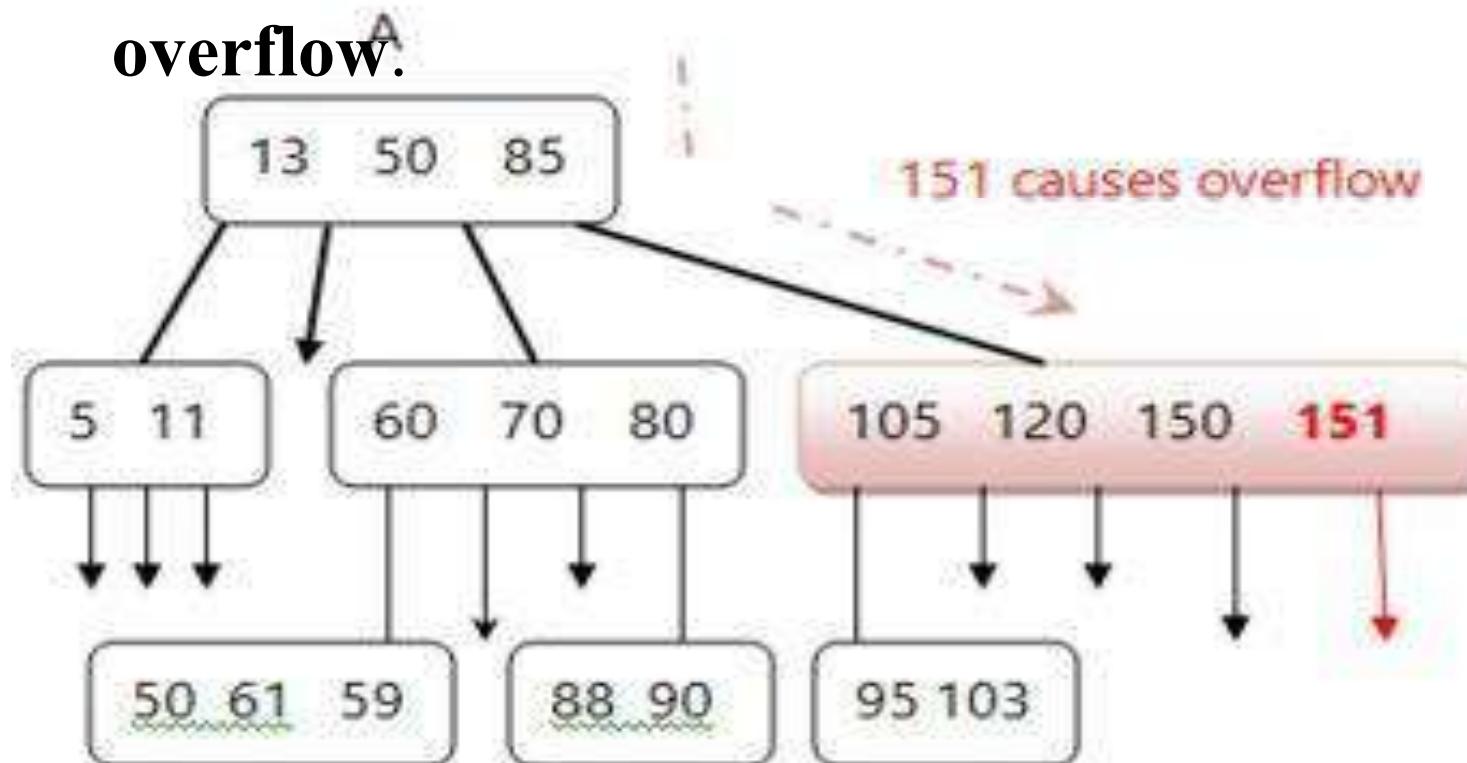
## • 2-3-4 Trees Operations

- an insert is achieved because there is a search miss on that key value.
- Key 151 does not exist and can therefore be added.
- a link is created and 151 is inserted in the appropriate node.
- This, however, results in a violation of the 2-3-4 tree rule that a node can carry no more than  $N$  or 4 child nodes and  $(N-1)$  or 3 key values. This violation is referred to as an **overflow**.

# Multiway

## Trees Operations

- 2-3-4 Trees
  - This violation is referred to as an **overflow**.



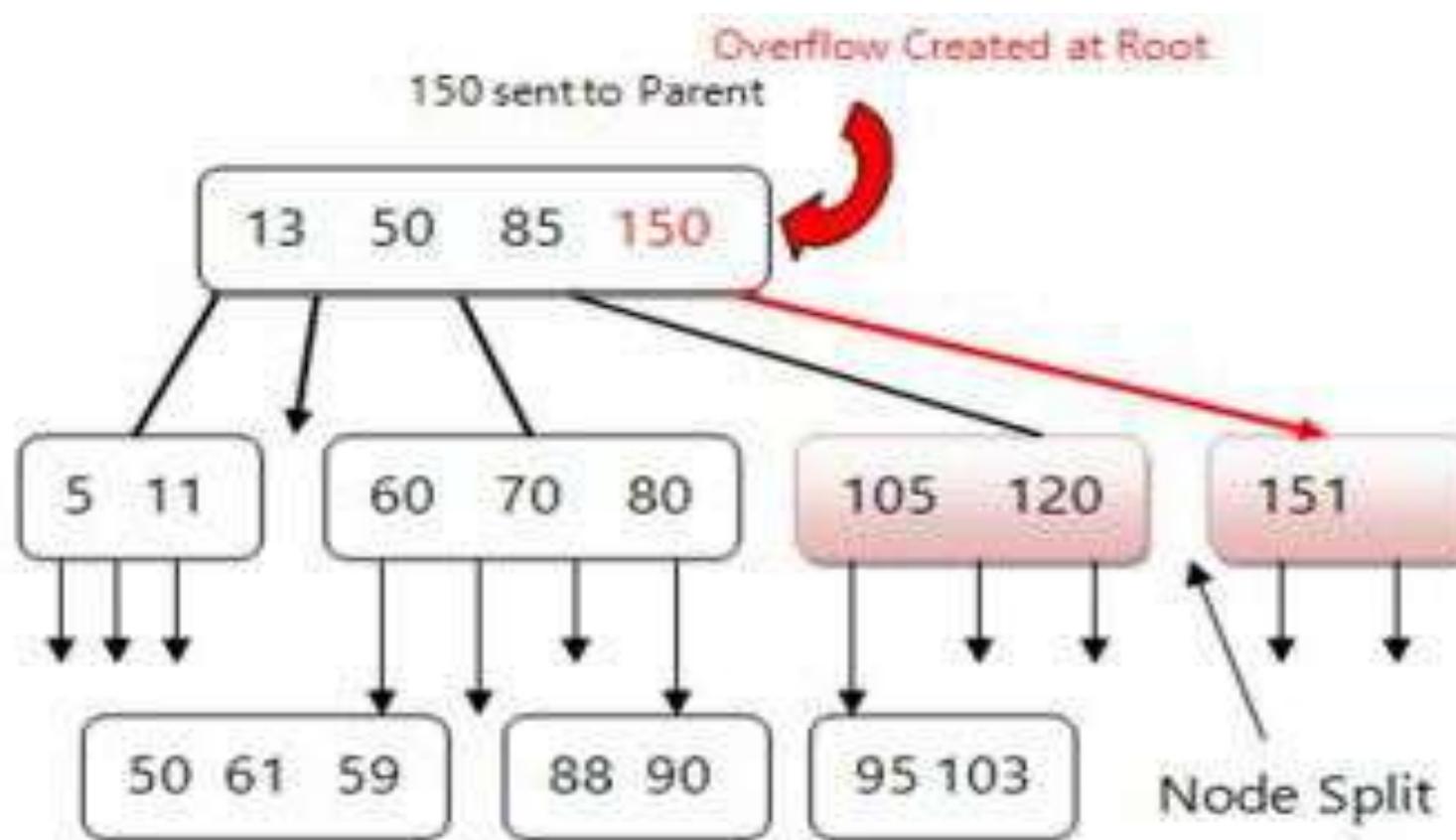
# Multiway

## • 2-3-4 Trees Operations

- To resolve the problem and re-balance the tree, the node with the overflow is split and key value 150 is sent to the parent node, which in this case, is the root.
- The original node is no longer in overflow as it has been split, but the root node is now in overflow because the key 150 has been inserted

# Multiway Trees

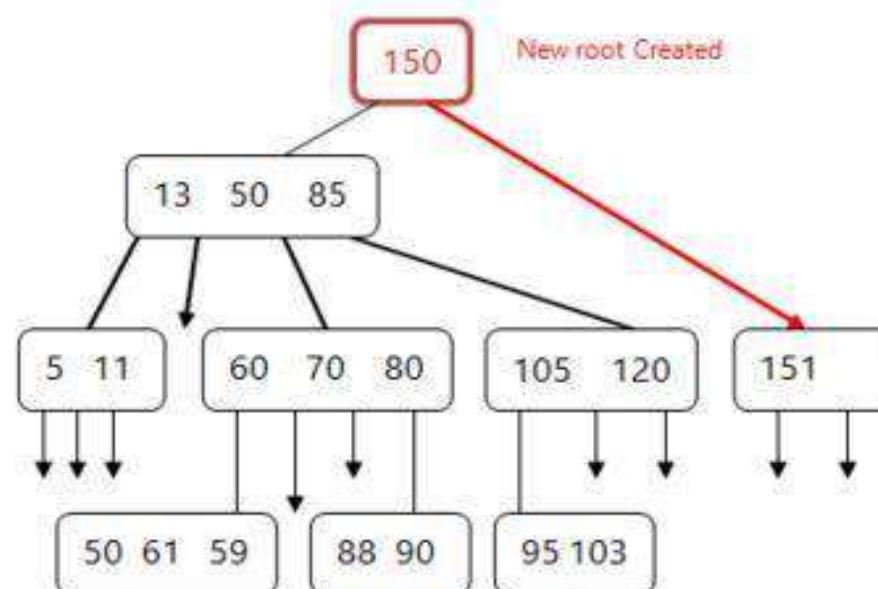
- 2-3-4 Trees Operations



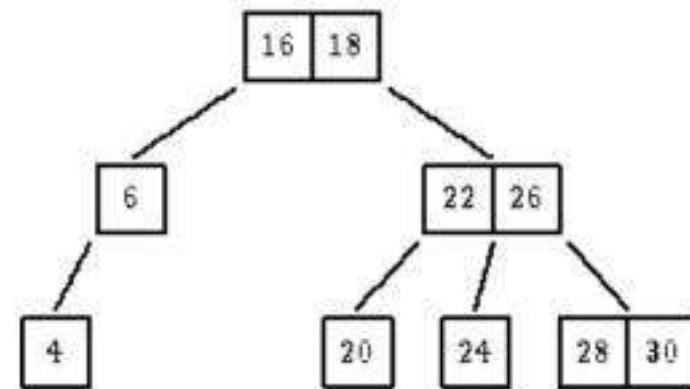
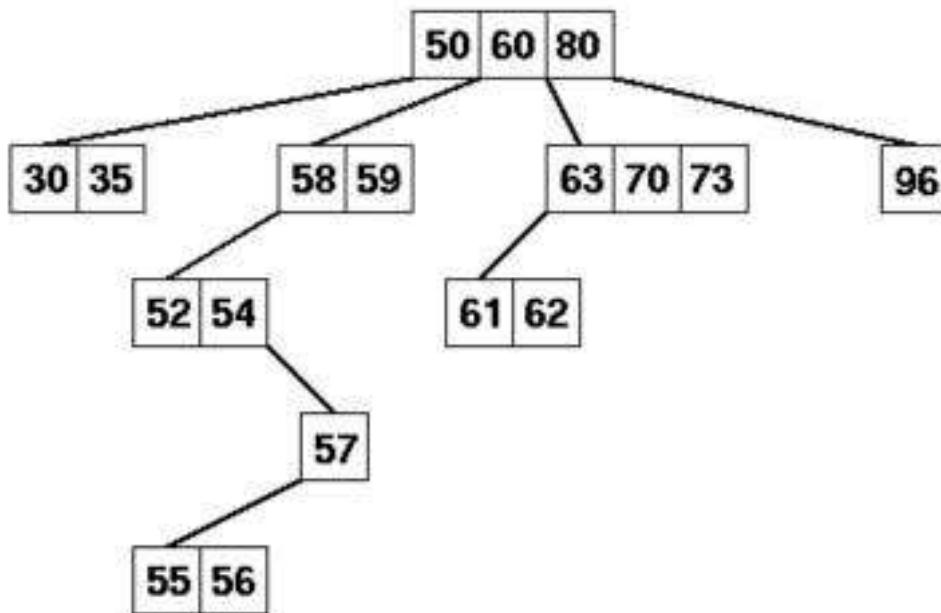
# Multiway

## Trees • 2-3-4 Trees Operations

- To fix this, the root node needs to have a single key with all other nodes emanating from it.
- key 150 is used to create a new root node and the tree is corrected.



## Examples of Multi-way Trees



- Note: In a multiway tree:
  - The leaf nodes need not be at the same level.
  - A non-leaf node with  $n$  keys may contain less than  $n + 1$  non-empty subtrees.

## What is a B-Tree?

- A B-tree of order  $m$  (or branching factor  $m$ ), where  $m \geq 3$ , is either an empty tree or a multiway search tree with the following properties:
  - The root is either a leaf or it has at least  $two$  non-empty subtrees and at most  $m$  non-empty subtrees.
  - Each non-leaf node, other than the root, has at least  $\lceil m/2 \rceil$  non-empty subtrees and at most  $m$  non-empty subtrees. (Note:  $\lceil x \rceil$  is the lowest integer  $> x$  ).
  - The number of keys in each non-leaf node is one less than the number of non-empty subtrees for that node.
  - All leaf nodes are at the same level; that is the tree is perfectly balanced.
- Note: The minimum number of no-empty subtrees in a non-leaf node of a B-tree is  $\lceil m/2 \rceil$  to ensure that the height of the B-tree is low

## What is a B-tree? (cont'd)

For a non-empty B-tree of order  $m$ :

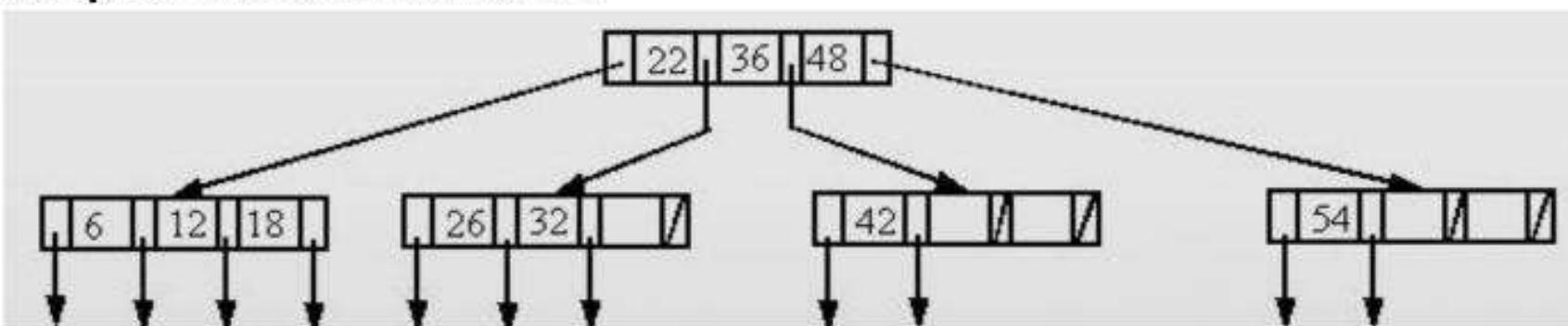
This will be zero, if the tree is empty

	Root node	Non-root node
Minimum number of keys	1	$\lceil m/2 \rceil - 1$
Minimum number of non-empty subtrees	2	$\lceil m/2 \rceil$
Maximum number of keys	$m - 1$	$m - 1$
Maximum number of non-empty subtrees	$m$	$m$

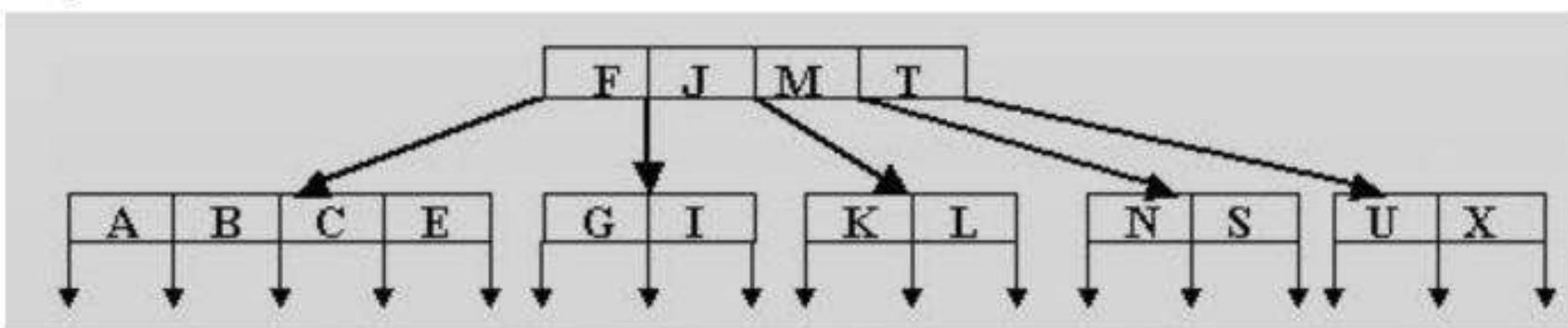
These will be zero if the node is a leaf

## B-Tree Examples

### Example1: A B-tree of order 4



### Example2: A B-tree of order 5



### Note:

- The data references are not shown.
- The leaf references are to empty subtrees
- The representation of example2 is a simplification in which unused key positions are not shown
- We may simplify the representation further by omitting the leaf references

## More on Why B-Trees

- B-trees are suitable for representing huge tables residing in secondary memory because:
  1. With a large branching factor  $m$ , the height of a B-tree is low resulting in fewer disk accesses.  
**Note: As  $m$  increases the amount of computation at each node increases; however this cost is negligible compared to hard-drive accesses.**
  2. The branching factor can be chosen such that a node corresponds to a block of secondary memory.
  3. The most common data structure used for database indices is the B+ tree which is a variation of the B-tree. An **index** is any data structure that takes as input a property (e.g. a value for a specific field), called the search key, and **quickly** finds all records with that property.

## Comparing B-Trees with AVL Trees

- The height  $h$  of a B-tree of order  $m$ , with a total of  $n$  keys, satisfies the inequality:

$$h \leq 1 + \log_{\lceil m/2 \rceil} ((n+1)/2)$$

- If  $m = 300$  and  $n = 16,000,000$  then  $h \approx 4$ .
- Thus, in the worst case finding a key in such a B-tree requires 3 disk accesses (assuming the root node is always in main memory ).
- The average number of comparisons for an AVL tree with  $n$  keys is  $\log n + 0.25$  where  $n$  is large.
- If  $n = 16,000,000$  the average number of comparisons is 24.
- Thus, in the average case, finding a key in such an AVL tree requires 24 disk accesses.

## Searching a B-Tree

- Searching **KEY**:
  - Start from the root
  - If root or an internal node is reached:
    - Search **KEY** among the keys in that node
      - linear search or binary search
      - If found, return the corresponding record
    - If **KEY < smallest key**, follow the leftmost child reference down
    - If **KEY > largest key**, follow the rightmost child reference down
    - If  $K_i < KEY < K_j$ , follow the child reference between  $K_i$  and  $K_j$
  - If a leaf is reached:
    - Search **KEY** among the keys stored in that leaf
      - linear search or binary search
    - If found, return the corresponding record; otherwise report not found

# Insertion into a B -Tree

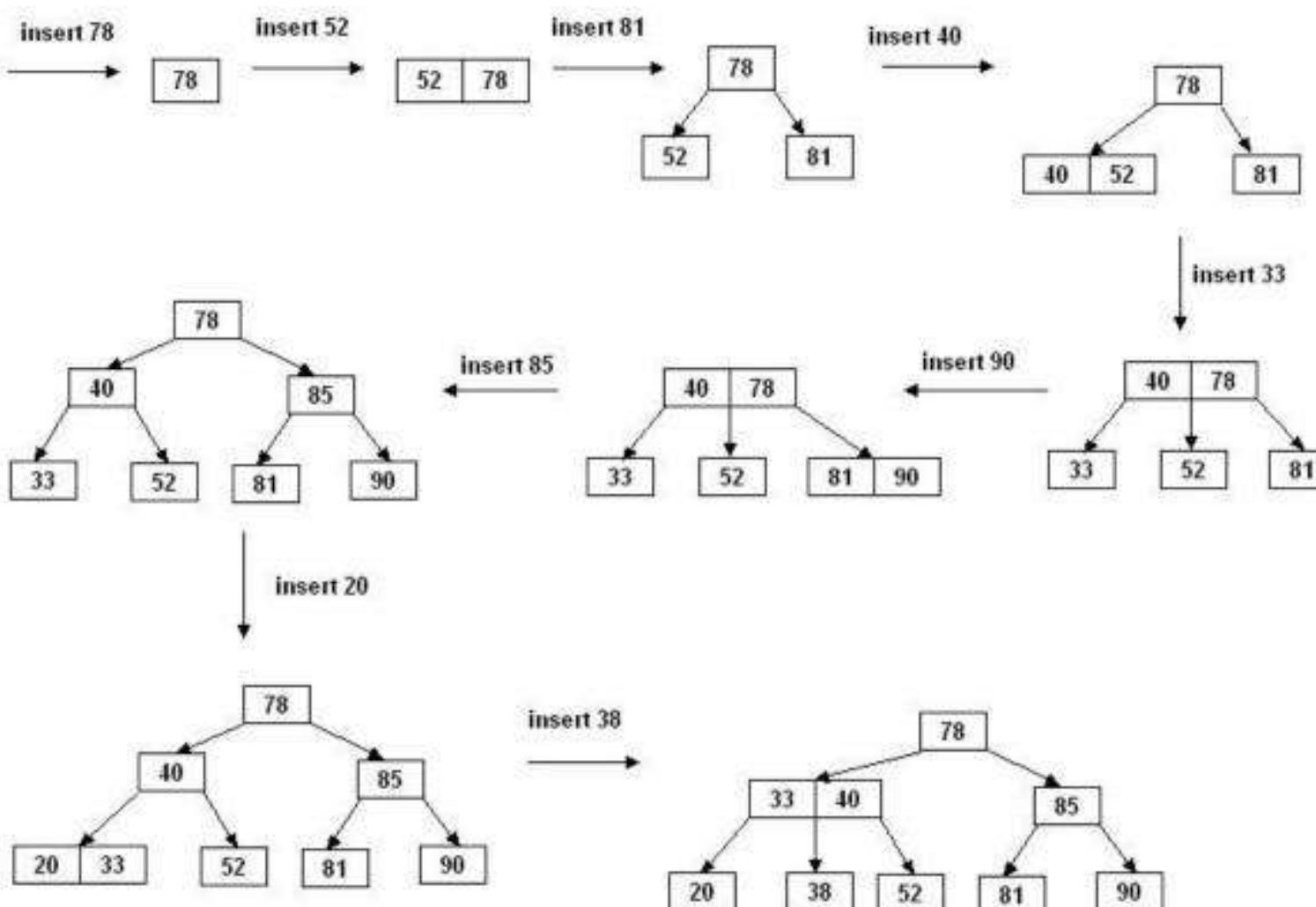
- The condition that all leaves must be on the same level forces a characteristic behavior of B-trees, namely that B-trees are not allowed to grow at the their leaves; instead they are forced to grow at the root.
- When inserting into a B-tree, a value is inserted directly into a leaf. This leads to three common situations that can occur:
  - i. A key is placed into a leaf that still has room.
  - ii. The leaf in which a key is to be placed is full.
  - iii. The root of the B-tree is full.

## Insertion in B-Trees

- **OVERFLOW CONDITION:**  
**A root-node or a non-root node of a B-tree of order  $m$  overflows if, after a key insertion, it contains  $m$  keys.**
- **Insertion algorithm:**  
**If a node overflows, split it into two, propagate the "middle" key to the parent of the node. If the parent overflows the process propagates upward. If the node has no parent, create a new root node.**
- **Note:**
  - **Insertion of a key always starts at a leaf node.**
  - **A key is inserted with its associated data**

# Insertion in B-Trees (cont'd)

- **Insertion in a B-tree of odd order**
- Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3

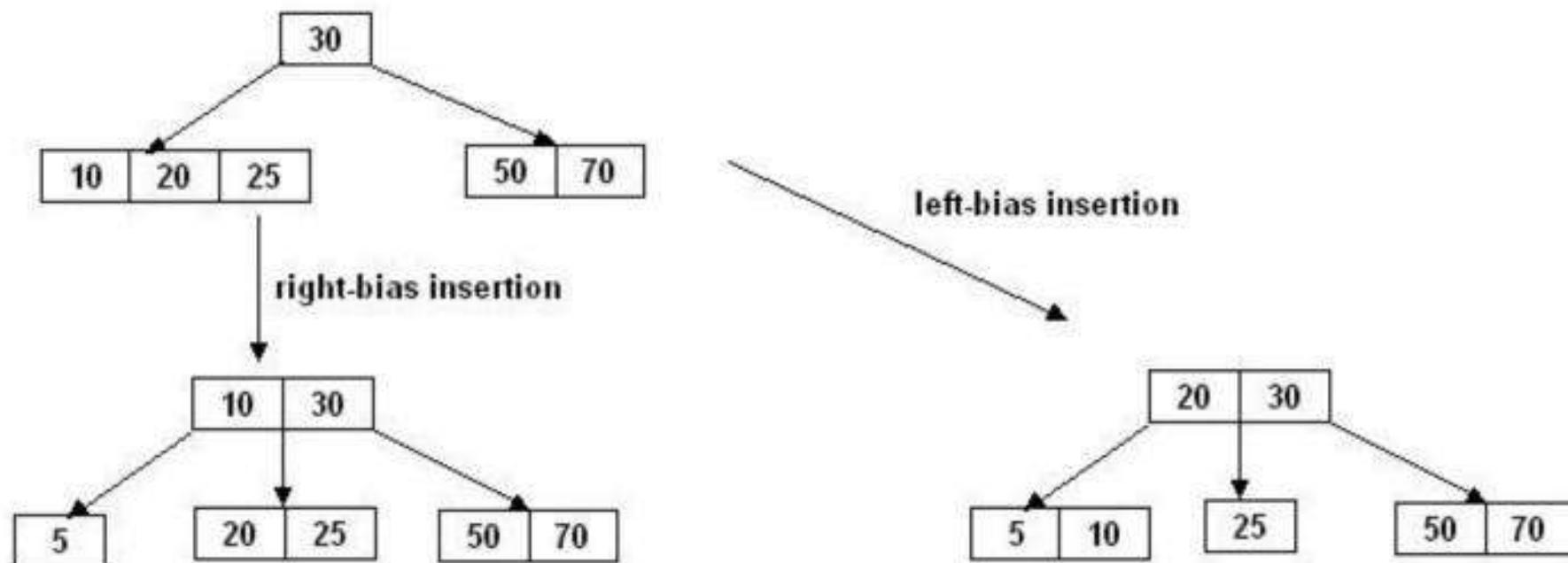


## Insertion in B-Trees (cont'd)

- **Insertion in a B-tree of even order**

At each node the insertion can be done in two different ways:

- **right-bias:** The node is split such that its right subtree has more keys than the left subtree.
- **left-bias:** The node is split such that its left subtree has more keys than the right subtree.
- **Example:** Insert the key **5** in the following B-tree of order **4**:



# B-Tree Insertion Algorithm

```
do{
    if (m is odd) {
        split currentNode into two siblings such that the right sibling rs has m/2 right-most keys,
        and the left sibling ls has m/2 left-most keys;
        Let w be the middle key of the splinted node;
    }
    else {      // m is even
        split currentNode into two siblings by any of the following methods:
        • right-bias: the right sibling rs has m/2 right-most keys, and the left sibling ls has (m-1)/2 left-most keys.
        • left-bias: the right sibling rs has (m-1)/2 right-most keys, and the left sibling ls has m/2 left-most keys.
        let w be the "middle" key of the splinted node;
    }
}
```

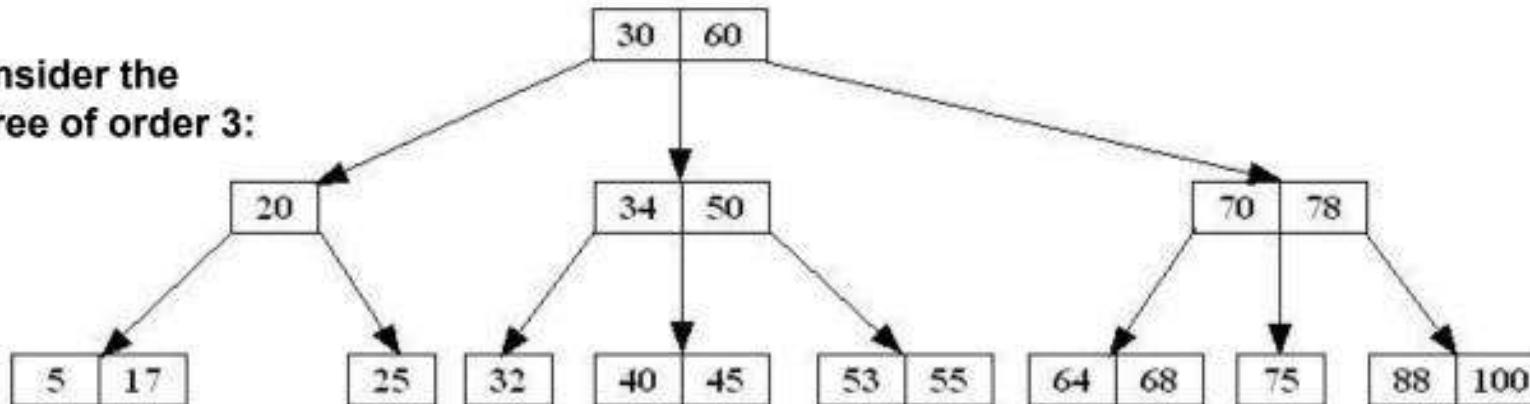
# Deleting from a B -Tree

- The deletion process will basically be a reversal of the insertion process - rather than splitting nodes, it's possible that nodes will be merged so that B-tree properties, namely the requirement that a node must be at least half full, can be maintained.
- There are two main cases to be considered:
  - i. Deletion from a leaf
  - ii. Deletion from a non-leaf

## Deletion in B-Trees

- Like insertion, deletion must be on a leaf node. If the key to be deleted is not in a leaf, swap it with either its successor or predecessor (each will be in a leaf).
- The successor of a key  $k$  is the smallest key greater than  $k$ .
- The predecessor of a key  $k$  is the largest key smaller than  $k$ .
- IN A B-TREE THE SUCCESSOR AND PREDECESSOR, IF ANY, OF ANY KEY IS IN A LEAF NODE

**Example:** Consider the following B-tree of order 3:



key	predecessor	successor
20	17	25
30	25	32
34	32	40
50	45	53
60	55	64
70	68	75
78	75	88

## Deletion in B-Trees (cont'd)

- **UNDERFLOW CONDITION**
- A non-root node of a B-tree of order  $m$  underflows if, after a key deletion, it contains  $\lceil m / 2 \rceil - 2$  keys
- The root node does not underflow. If it contains only one key and this key is deleted, the tree becomes empty.

## Deletion in B-Trees (cont'd)

- **Deletion algorithm:**

If a node underflows, **rotate** the appropriate key from the **adjacent** right- or left-sibling if the sibling contains at least  $\lceil m / 2 \rceil$  keys; otherwise perform a **merging**.

⇒ A key rotation must always be attempted before a merging

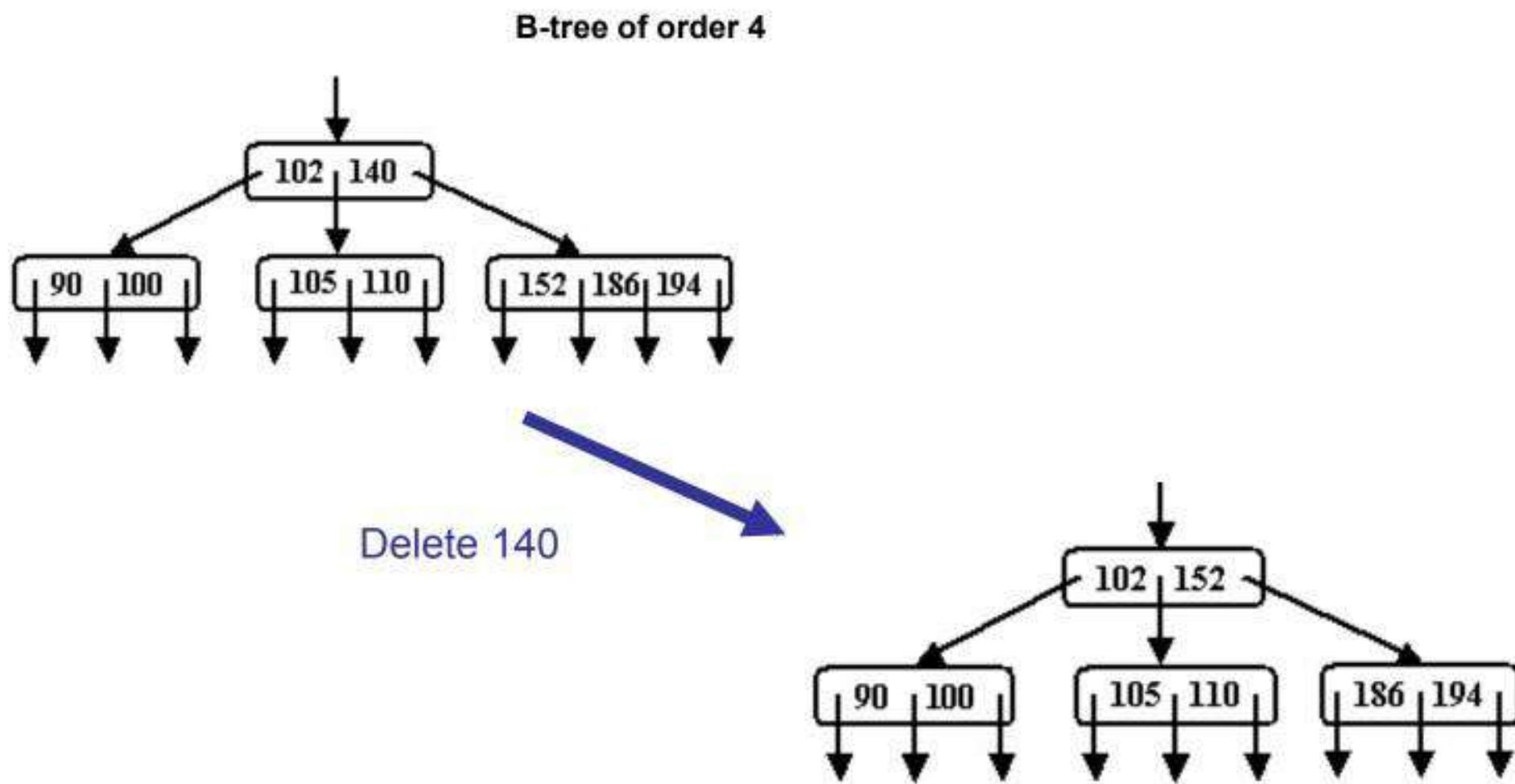
- There are **five** deletion cases:

1. The leaf does not underflow.
2. The leaf underflows and the adjacent right sibling has at least  $\lceil m / 2 \rceil$  keys.  
perform a **left key-rotation**
3. The leaf underflows and the adjacent left sibling has at least  $\lceil m / 2 \rceil$  keys.  
perform a **right key-rotation**
4. The leaf underflows and each of the adjacent right sibling and the adjacent left sibling has at least  $\lceil m / 2 \rceil$  keys.  
perform either a **left** or a **right key-rotation**
5. The leaf underflows and each adjacent sibling has  $\lceil m / 2 \rceil - 1$  keys.  
perform a **merging**

## Deletion in B-Trees (cont'd)

Case1: The leaf does not underflow.

Example:

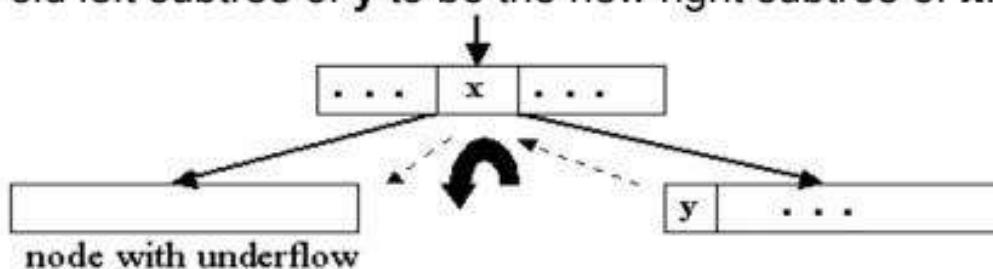


## Deletion in B-Trees (cont'd)

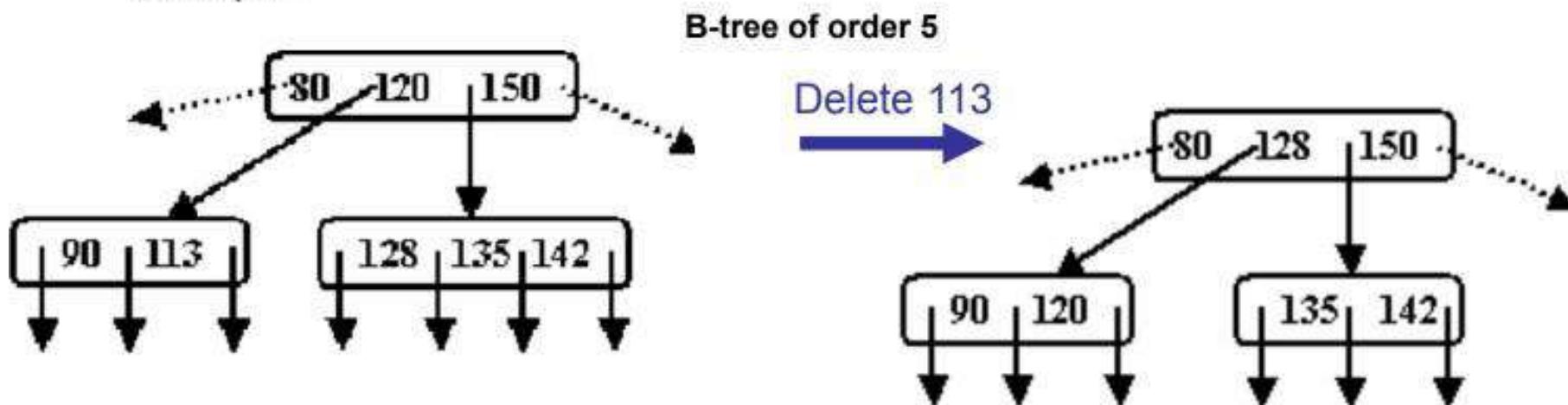
Case2: The leaf underflows and the adjacent right sibling has at least  $\lceil m / 2 \rceil$  keys.

Perform a left key-rotation:

1. Move the parent key  $x$  that separates the siblings to the node with underflow
2. Move  $y$ , the minimum key in the right sibling, to where the key  $x$  was
3. Make the old left subtree of  $y$  to be the new right subtree of  $x$ .



Example:

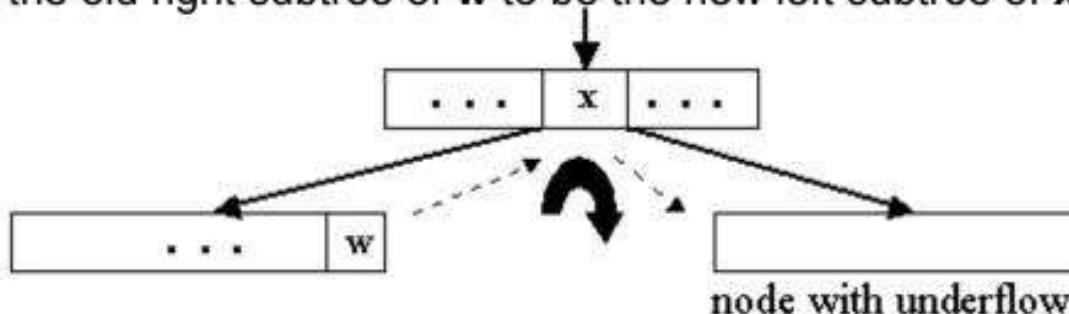


## Deletion in B-Trees (cont'd)

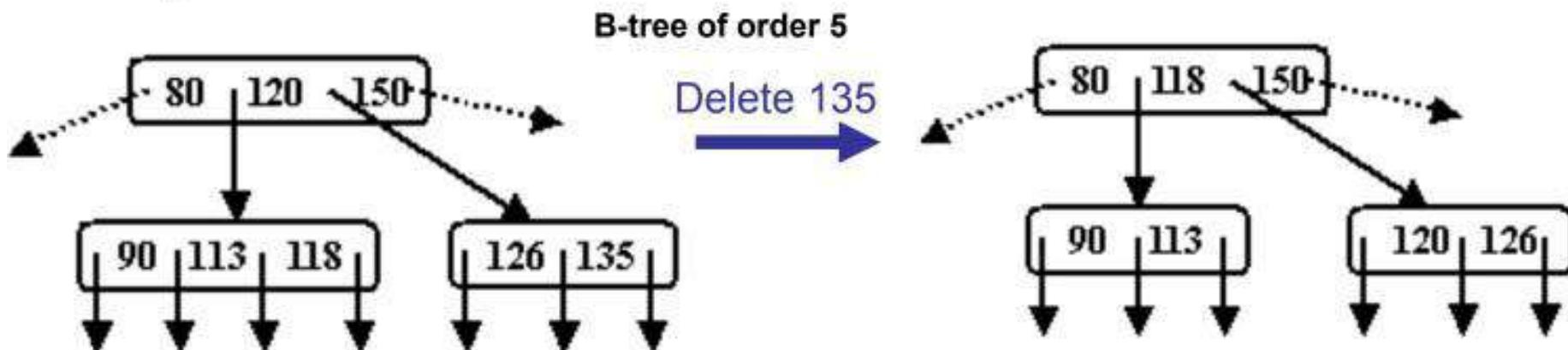
Case 3: The leaf underflows and the adjacent left sibling has at least  $\lceil m / 2 \rceil$  keys.

Perform a right key-rotation:

1. Move the parent key  $x$  that separates the siblings to the node with underflow
2. Move  $w$ , the maximum key in the left sibling, to where the key  $x$  was
3. Make the old right subtree of  $w$  to be the new left subtree of  $x$



Example:

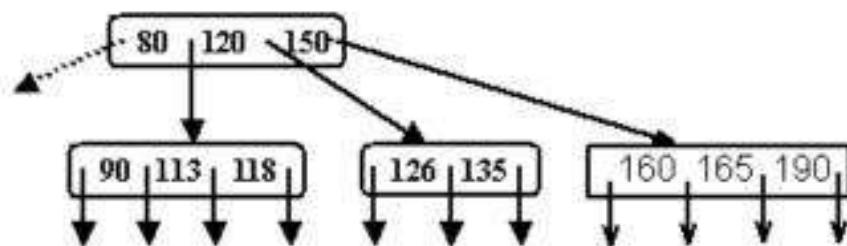


## Deletion in B-Trees (cont'd)

Case 4: The leaf underflows and each of the adjacent right sibling and the adjacent left sibling has at least  $\lceil m / 2 \rceil$  keys.

Example:

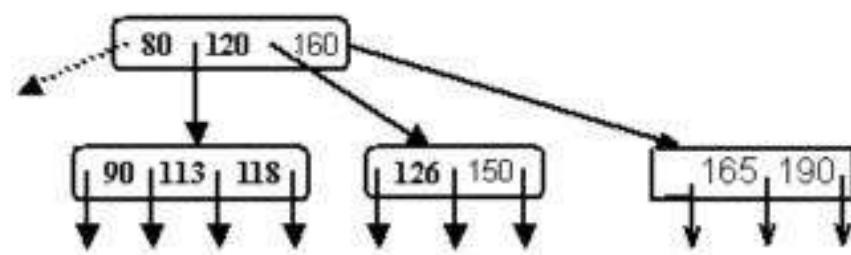
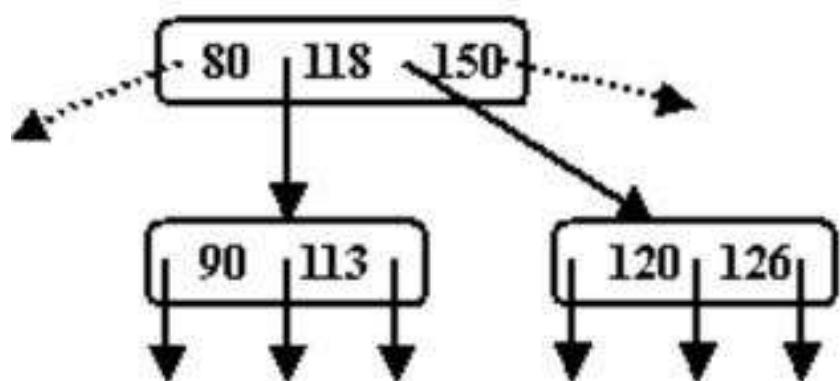
B-tree of order 5



Delete 135

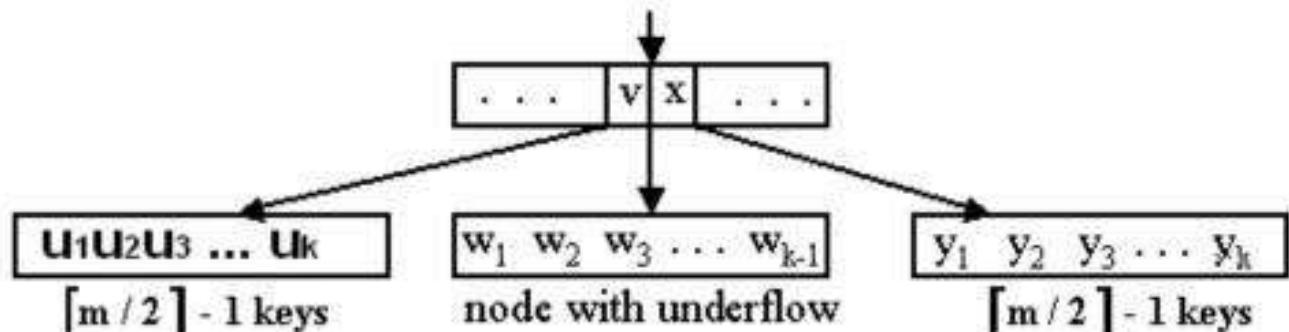
right rotation

left rotation

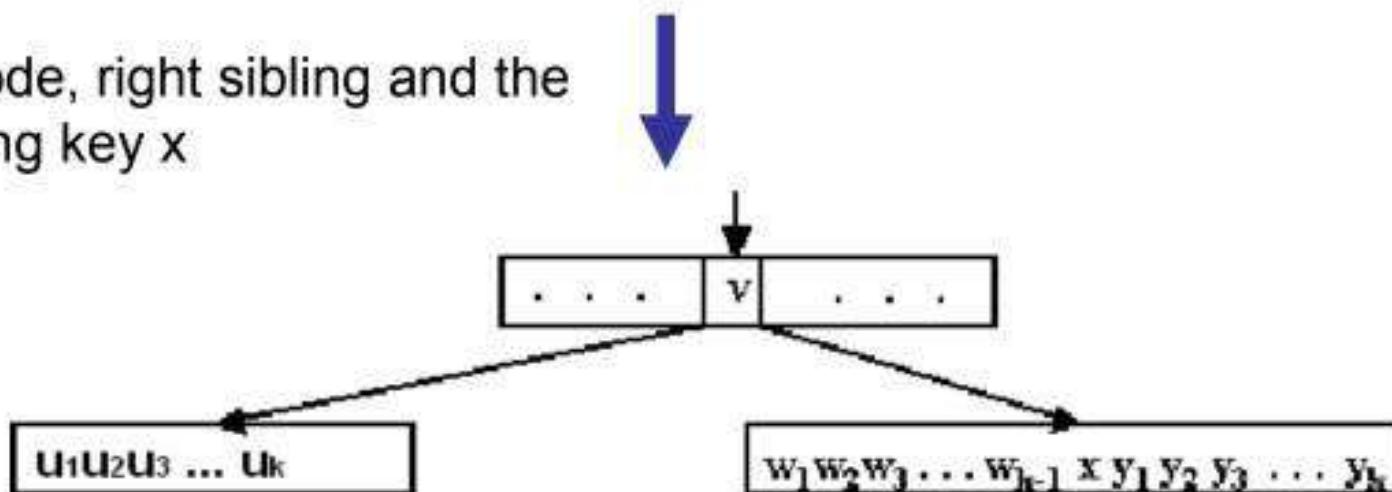


## Deletion in B-Trees (cont'd)

Case 5: The leaf underflows and each adjacent sibling has  $\lceil m / 2 \rceil - 1$  keys.



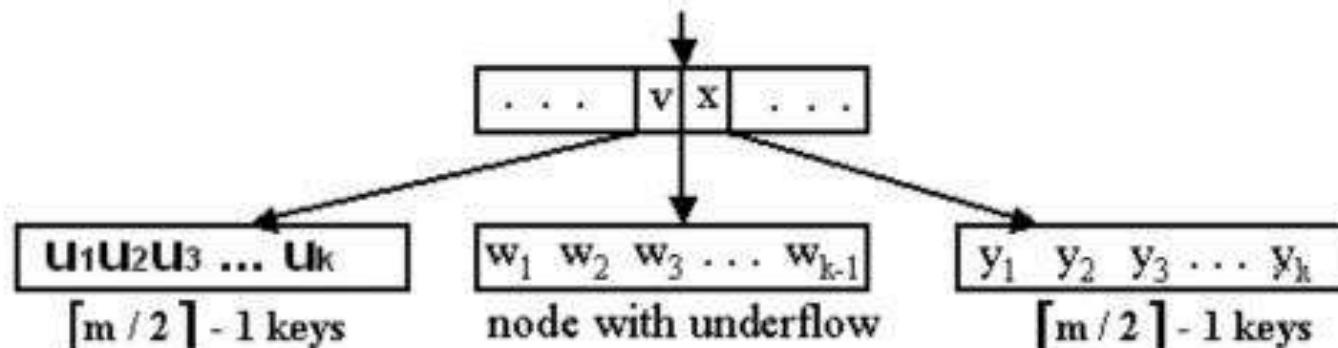
merge node, right sibling and the separating key x



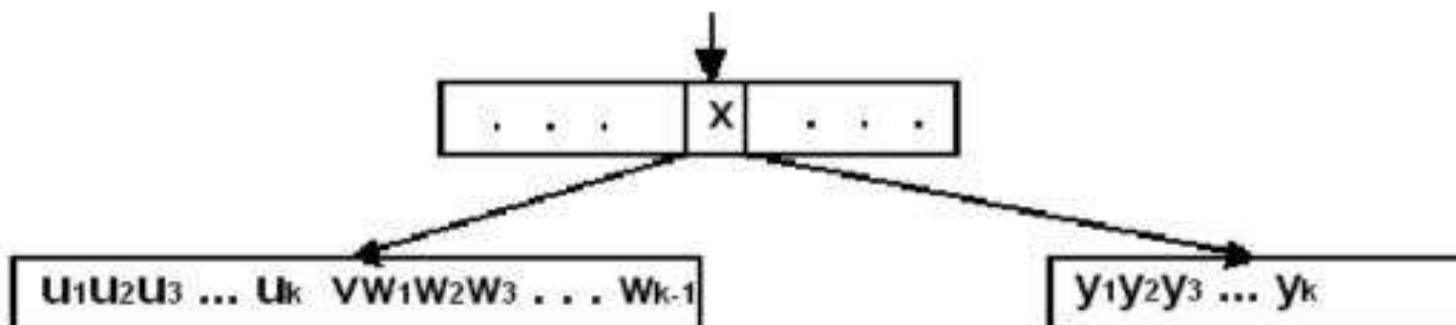
If the parent of the merged node underflows, the merging process propagates upward. In the limit, a root with one key is deleted and the height decreases by one.

## Deletion in B-Trees (cont'd)

**Note:** The merging could also be done by using the left sibling instead of the right sibling.

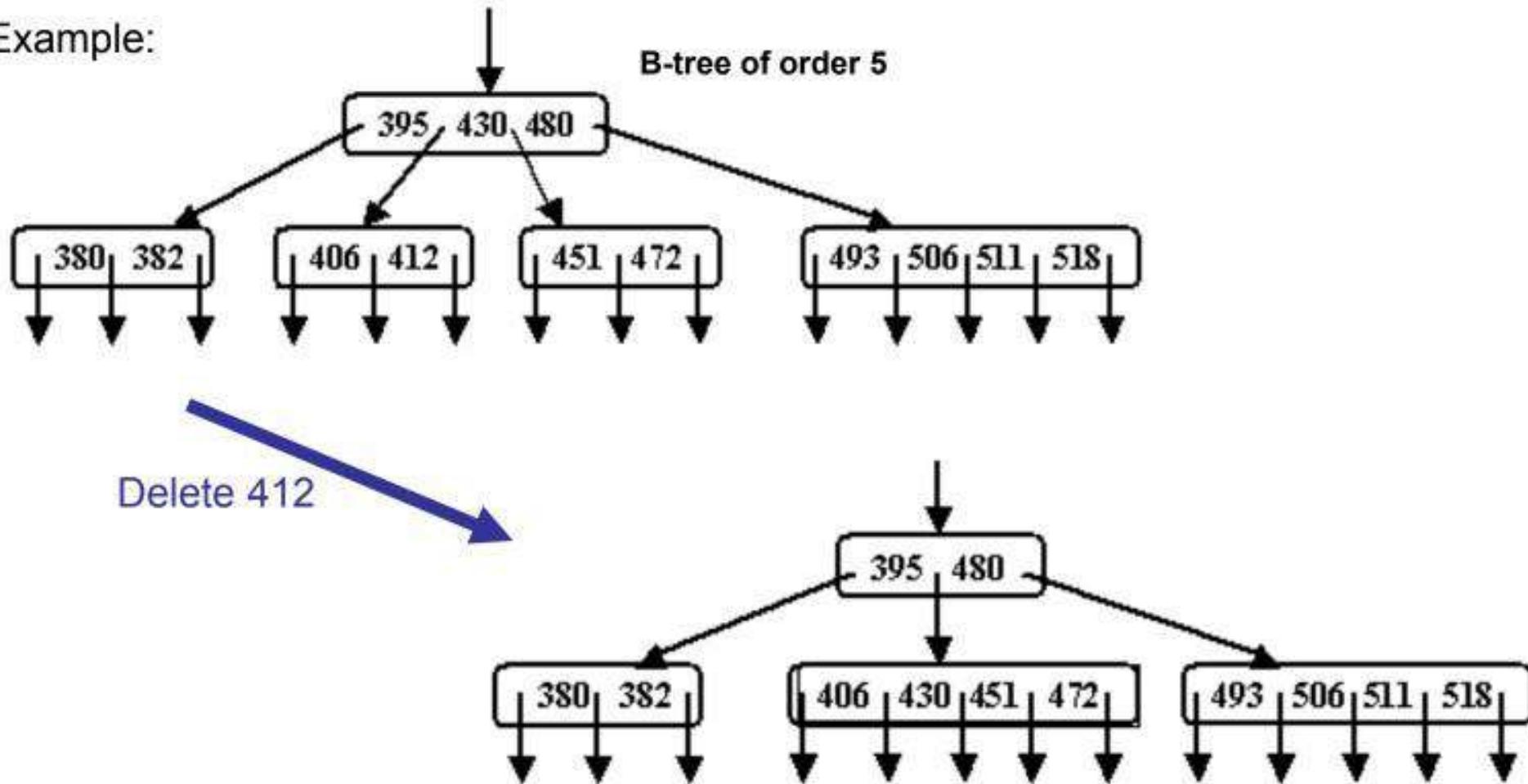


merge node, left sibling and the separating key v



## Deletion in B-Trees (cont'd)

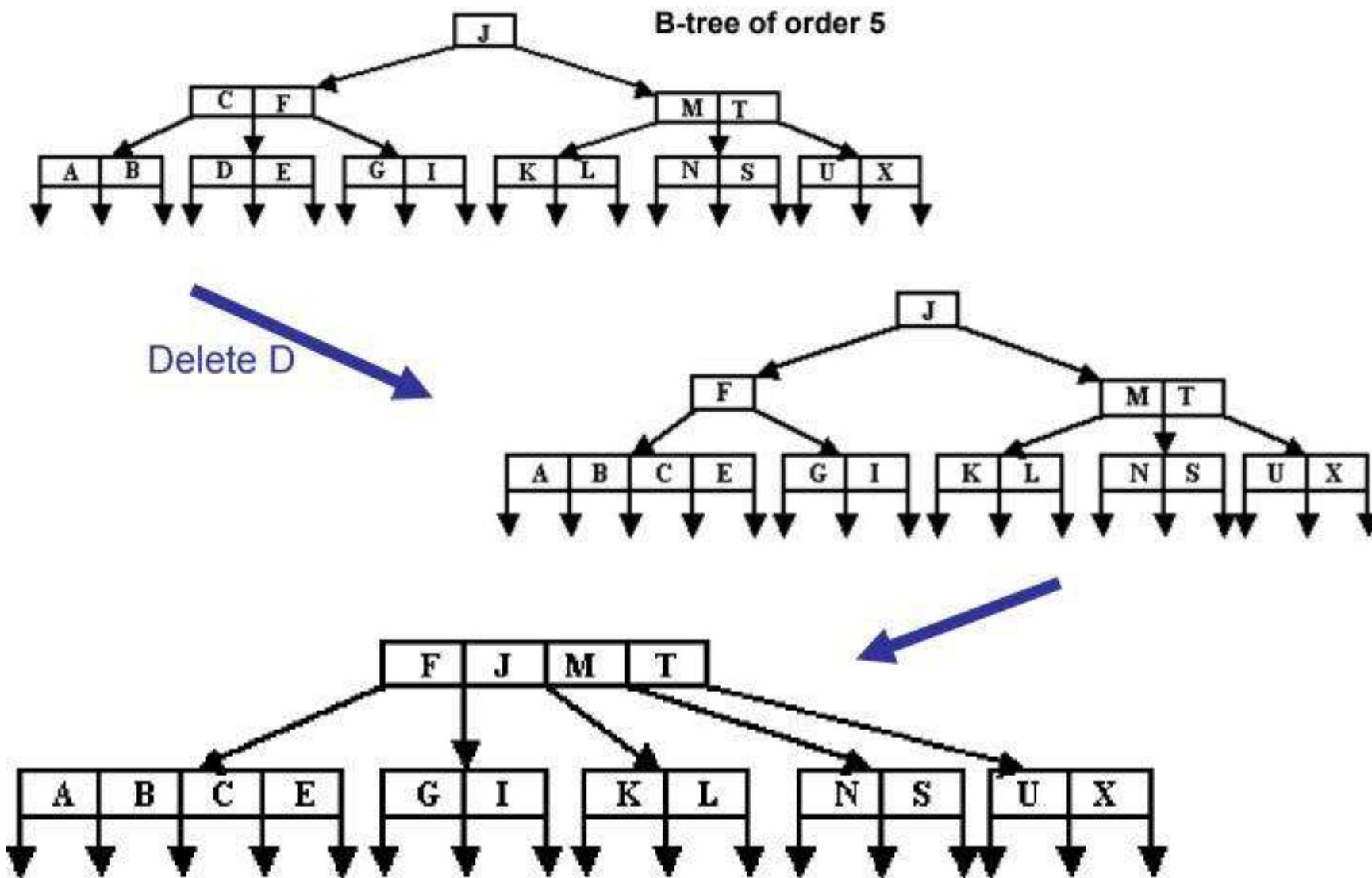
Example:



The parent of the merged node does not underflow. The merging process does not propagate upward.

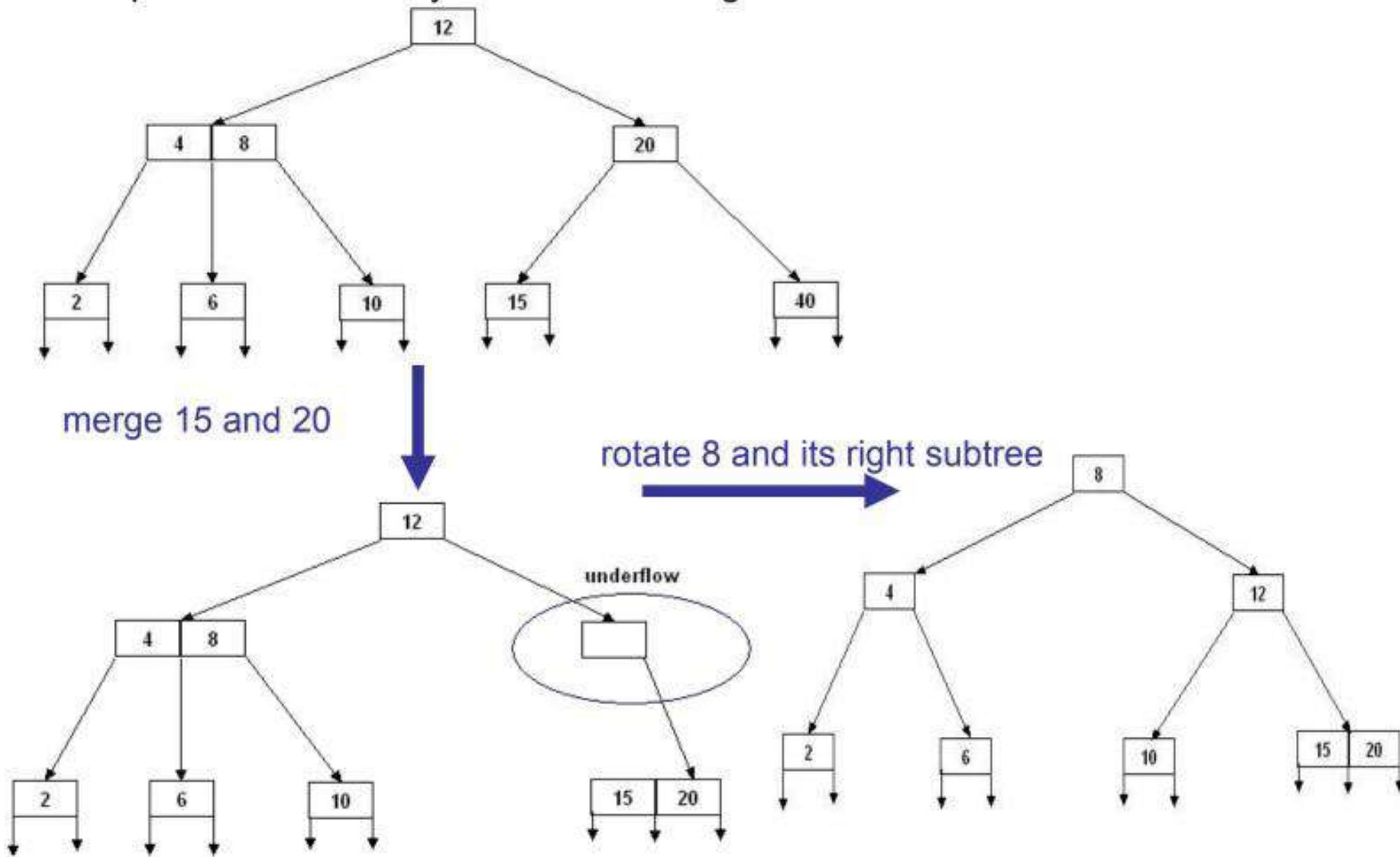
## Deletion in B-Trees (cont'd)

Example:



## Deletion : Special Case, involves rotation and merging

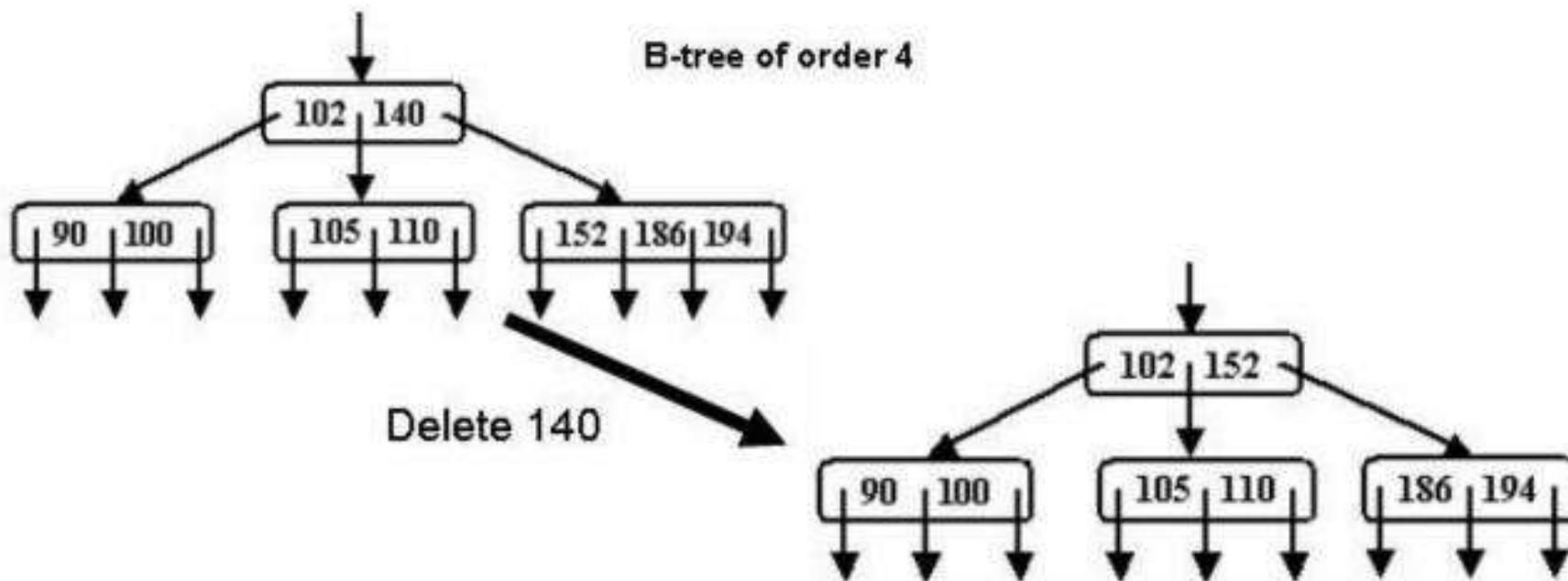
Example: Delete the key **40** in the following B-tree of order 3:



## Deletion of a non-leaf node

Deletion of a non-leaf key can always be done in two different ways: by first swapping the key with its successor or predecessor. The resulting trees may be similar or they may be different.

Example: Delete the key 140 in the following B-tree of order 4:



# B+

- The drawback of B-tree ~~Tree~~ indexing is that it stores the data pointer corresponding to a particular key value, along with that key value in the node of a B-tree.
- This technique greatly reduces the number of entries that can be packed into a node of a B-tree, thereby contributing to the increase in the number of levels in the B-tree, hence increasing the search time of a record.
- B+ tree eliminates the above drawback by **storing data pointers only at the leaf nodes of the tree.**
- Thus, the structure of the leaf nodes of a B+ tree is quite different from the structure of the internal nodes of the B tree.
- Since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them.

# Why Use B+ Tree?

1. B+ Trees are the best choice for storage systems with sluggish data access because they **minimize I/O operations** while facilitating **efficient disc access**.
2. B+ Trees are a good choice for database systems and applications needing **quick data retrieval** because of their **balanced structure**, which guarantees predictable performance for a variety of activities and facilitates effective range-based queries.

# B+

- B+ tree is an N-ary tree with a variable but often large number of children per node.
- B+ tree consists of a root, internal nodes and leaves.
- The root may be either a leaf or a node with two or more children.
- B+ tree can be viewed as a B-tree in which each node contains only keys (not key–value pairs), and to which **an additional level is added at the bottom with linked leaves**.
- The B+ Tree consists of two types of nodes:
  - internal nodes
  - leaf nodes

# B+

The leaf nodes of the B+ tree are linked together to provide ordered access to the search field to the records. Internal nodes of a B+ tree are used to guide the search. Some search field values from the leaf nodes are repeated in the internal nodes of the B+ tree.

Parameters	B+ Tree	B Tree
Structure	Separate leaf nodes for data storage and internal nodes for indexing	Nodes store both keys and data values
Leaf Nodes	Leaf nodes form a linked list for efficient range-based queries	Leaf nodes do not form a linked list
Order	Higher order (more keys)	Lower order (fewer keys)
Key Duplication	Typically allows key duplication in leaf nodes	Usually does not allow key duplication
Disk Access	Better disk access due to sequential reads in a linked list structure	More disk I/O due to non-sequential reads in internal nodes
Applications	Database systems, file systems, where range queries are common	In-memory data structures, databases, general-purpose use
are	Better performance for range queries and bulk data retrieval	Balanced performance for search, insert, and delete operations
Memory Usage	Requires more memory for internal nodes	Requires less memory as keys and values are stored in the same node

# B+ Tree

## Properties:

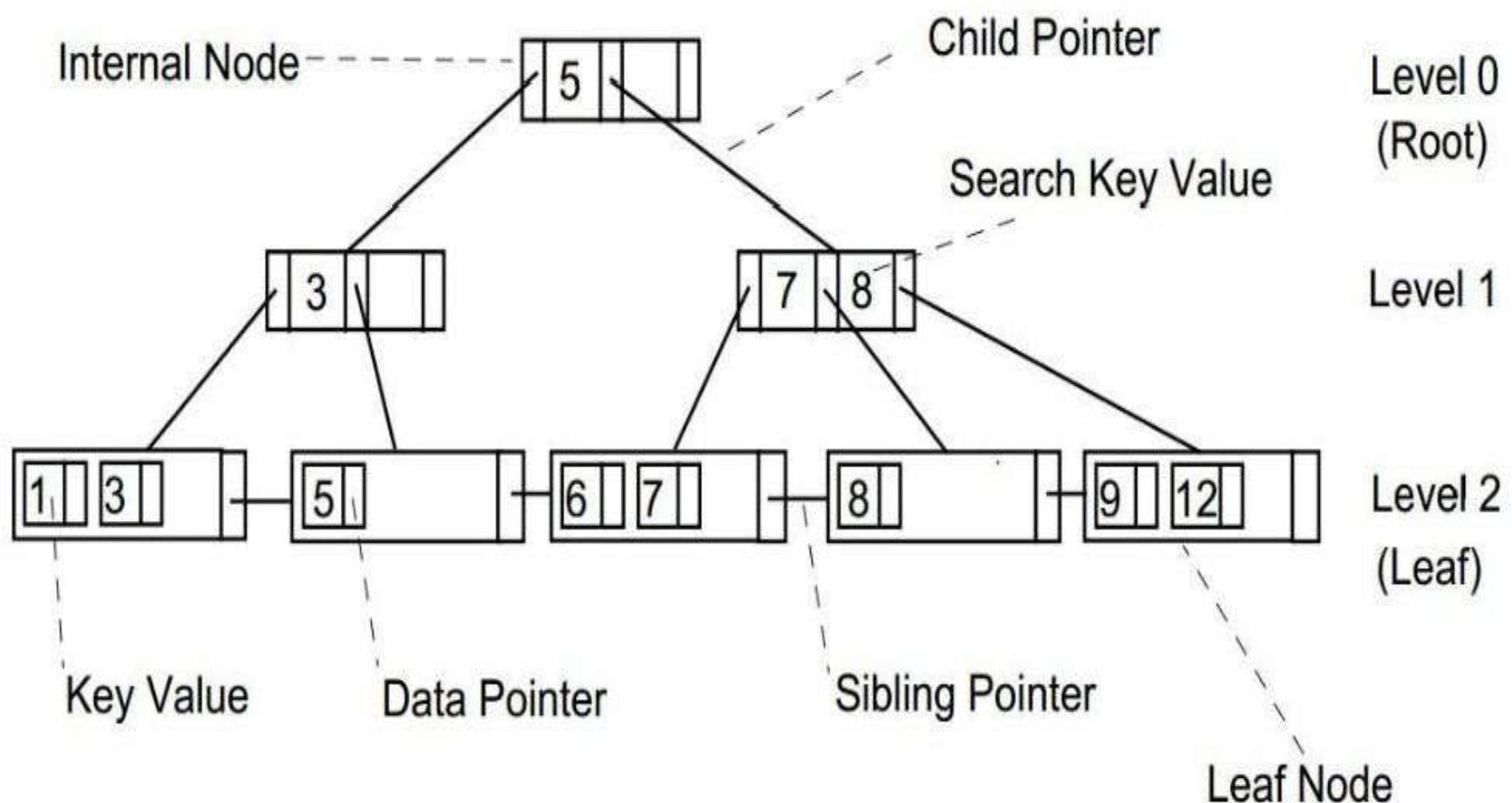
- Internal nodes point to other nodes in the tree.
- **Leaf nodes point to data in the database using data pointers.**  
Leaf nodes also contain an additional pointer, called the **sibling pointer**, which is used to improve the efficiency of certain types of search.
- All the nodes in a B+ Tree must be at least half full except the root node which may contain a minimum of two entries.

# B+ Tree

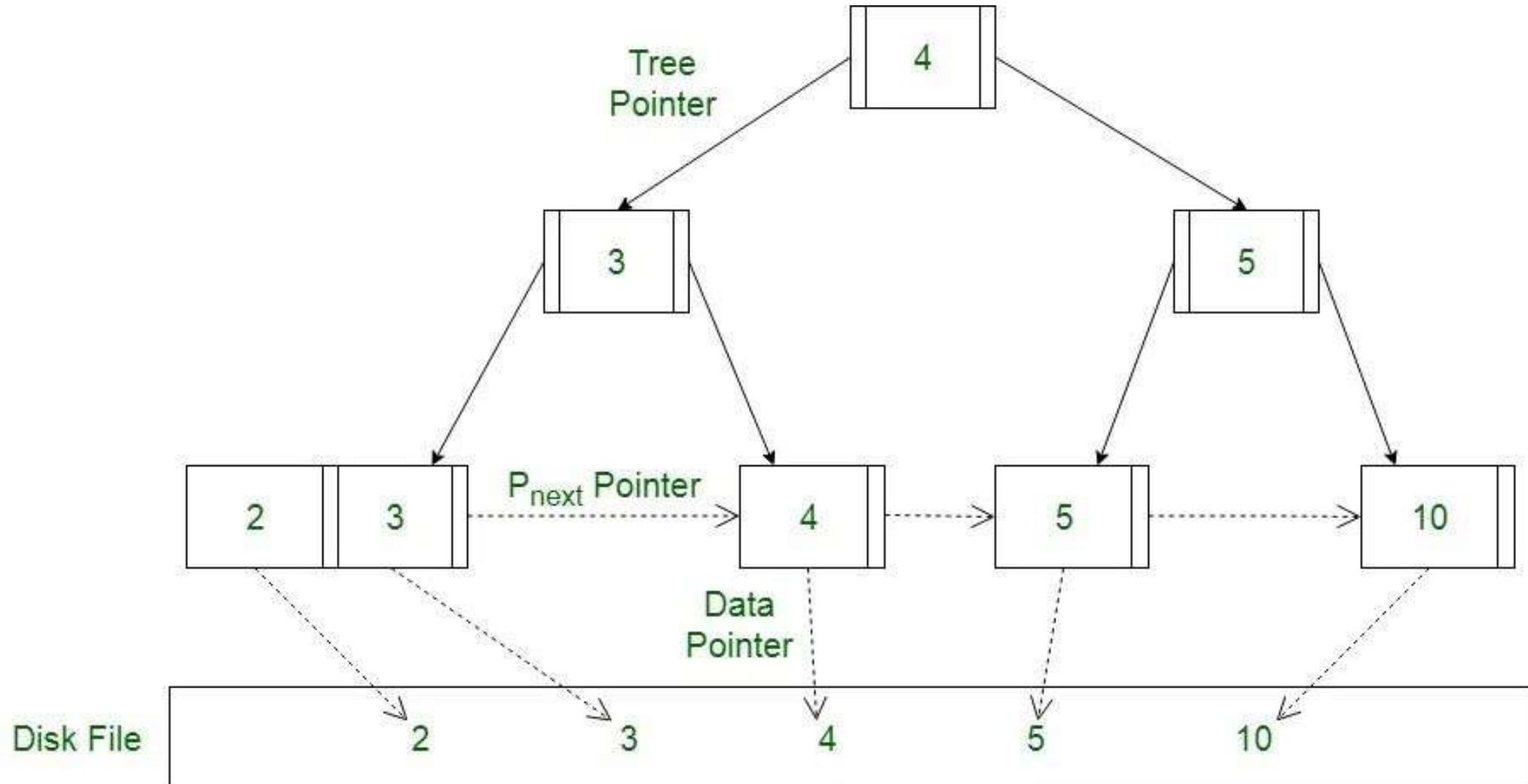
## Properties:

- Searching for a value in the B+ Tree always starts at the root node and moves downwards until it reaches a leaf node.
- Both internal and leaf nodes contain key values that are used to guide the search for entries in the index.
- The B+ Tree is called a **balanced tree** because every path from the root node to a leaf node is the same length. A balanced tree means that **all searches for individual values require the same number of nodes to be read from the disc**.

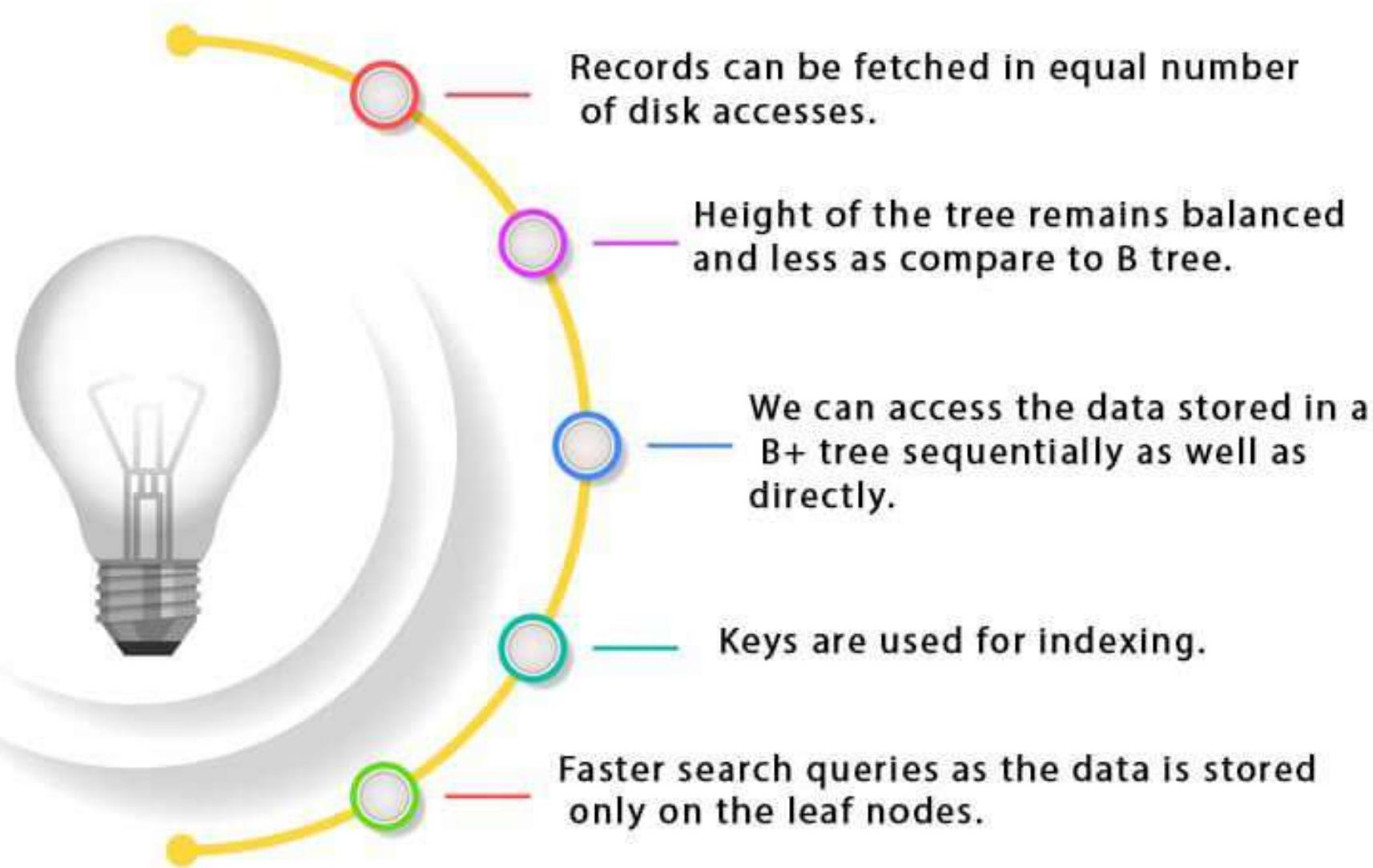
# B+



Using the  $P_{next}$  pointer it is viable to traverse all the leaf nodes, just like a linked list, thereby achieving **ordered access** to the records stored in the disk.



# Advantages of B+ Tree



# B+

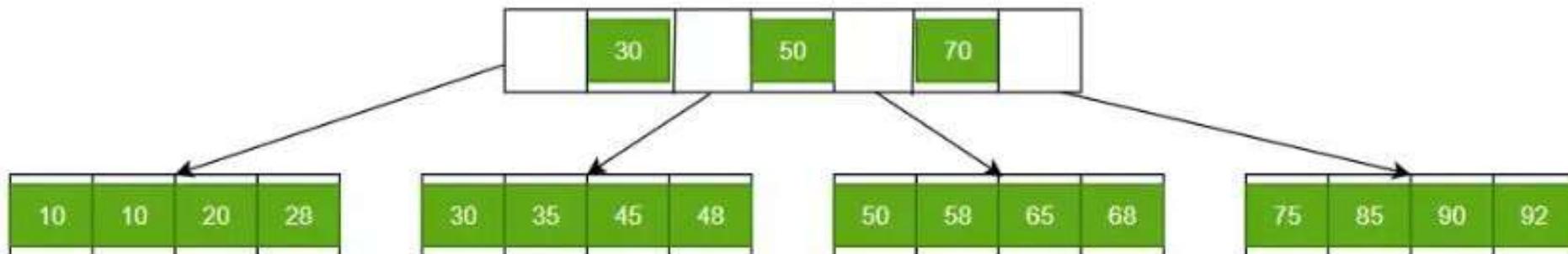
# Tree

- **Basic operations associated with B+ Tree:**

- **Searching a node in a B+ Tree**

- Perform a binary search on the records in the current node.
    - If a record with the search key is found, then return that record.
    - If the current node is a leaf node and the key is not found, then report an unsuccessful search.
    - Otherwise, follow the proper branch and repeat the process.

# Searching a Record in B+ Trees



*Searching Record in B+ Trees*

Let us suppose we have to find 58 in the B+ Tree. We will start by fetching from the root node then we will move to the leaf node, which might contain a record of 58. In the image given above, we will get 58 between 50 and 70. Therefore, we will we are getting a leaf node in the third leaf node and get 58 there. If we are unable to find that node, we will return that 'record not founded' message.

# B+

## Tree

- **Insertion of node in a B+ Tree:**

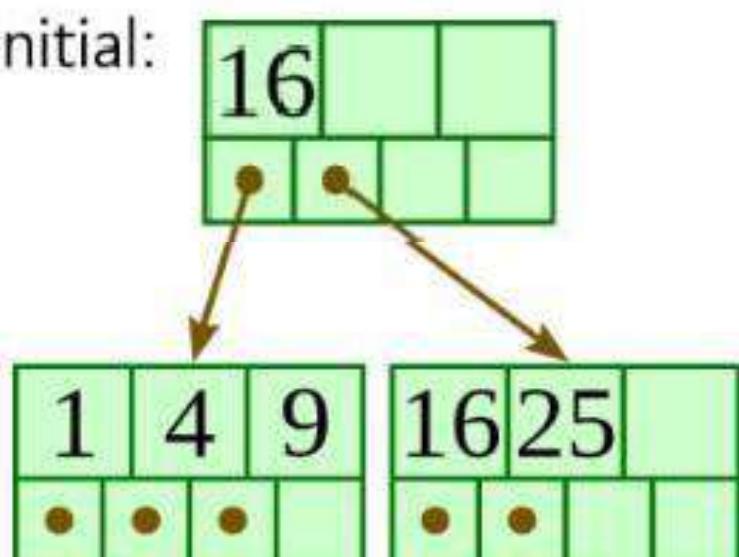
- Allocate new leaf and move half the buckets elements to the new bucket.
- Insert the new leaf's smallest key and address into the parent.
- If the parent is full, split it too.
- Add the middle key to the parent node.
- Repeat until a parent is found that need not split.
- If the root splits, create a new root which has one key and two pointers. (That is, the value that gets pushed to the new root gets removed from the original node)

# Insertion in B+ Trees

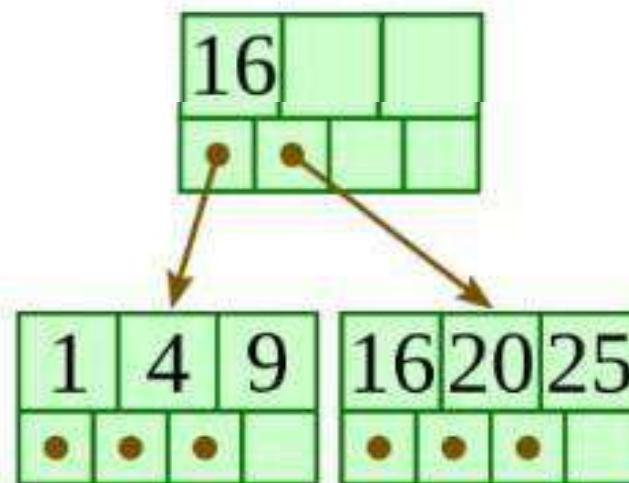
Insertion in B+ Trees is done via the following steps.

- Every element in the tree has to be inserted into a leaf node. Therefore, it is necessary to go to a proper leaf node.
- Insert the key into the leaf node in increasing order if there is no overflow.

Initial:

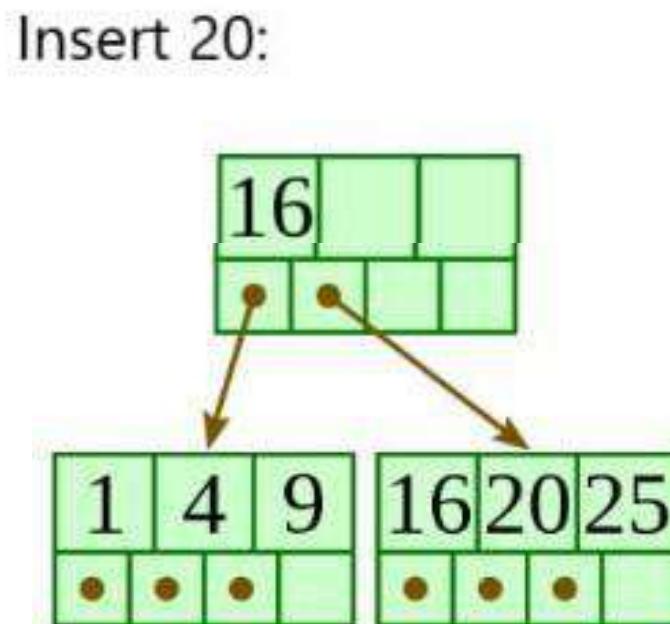
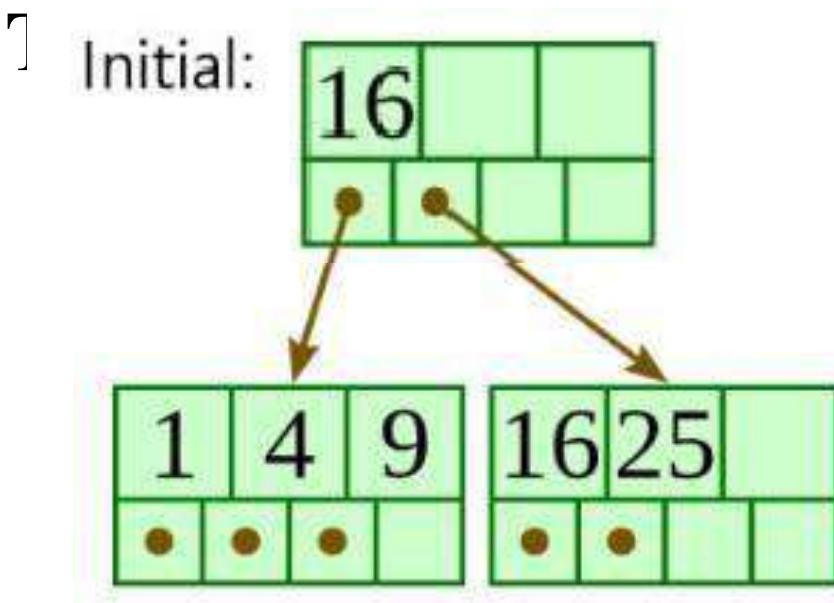


Insert 20:



# B+ Tree

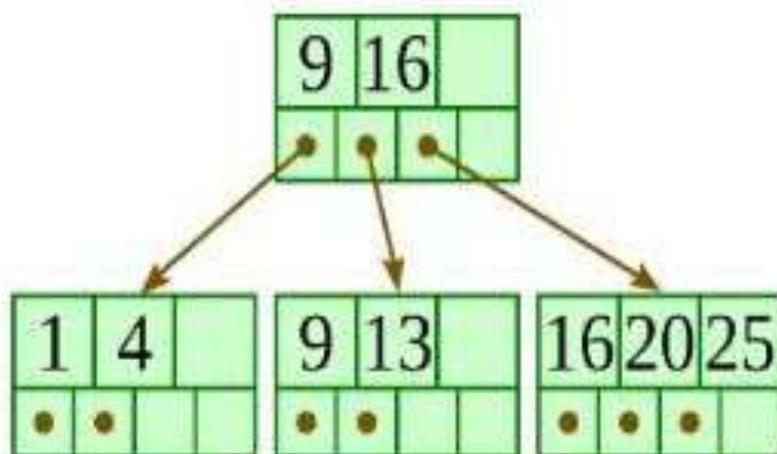
- Insertion of node in a B+



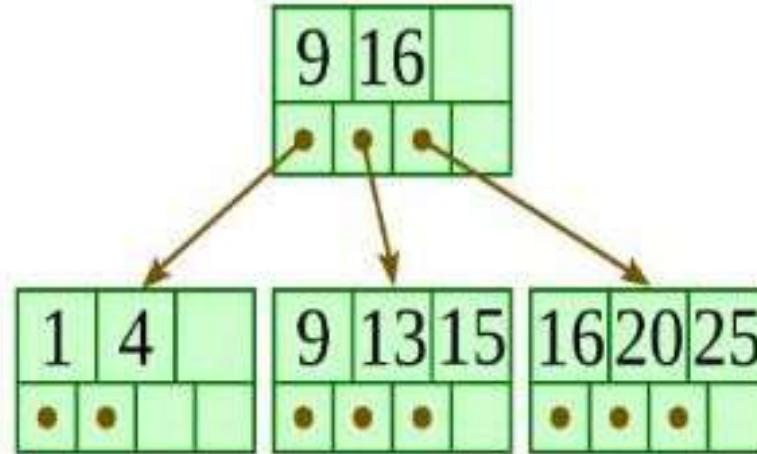
# B+ Tree

- **Insertion of node in a B+**

Insert 13:



Insert 15:

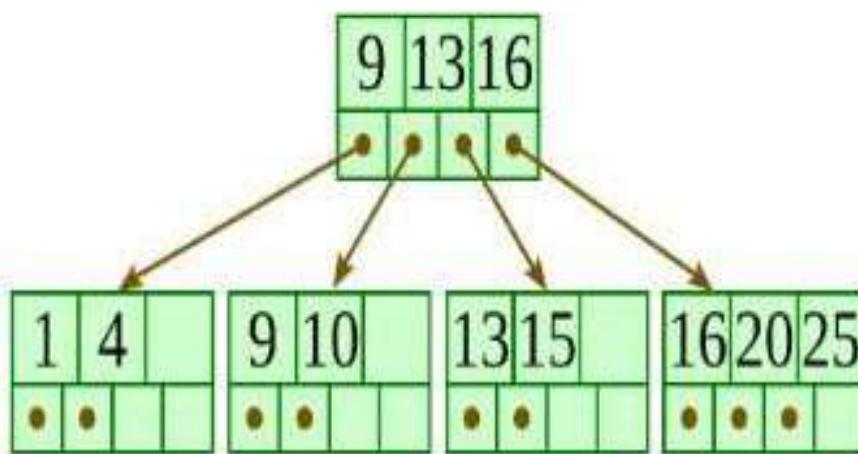


# B+ Tree

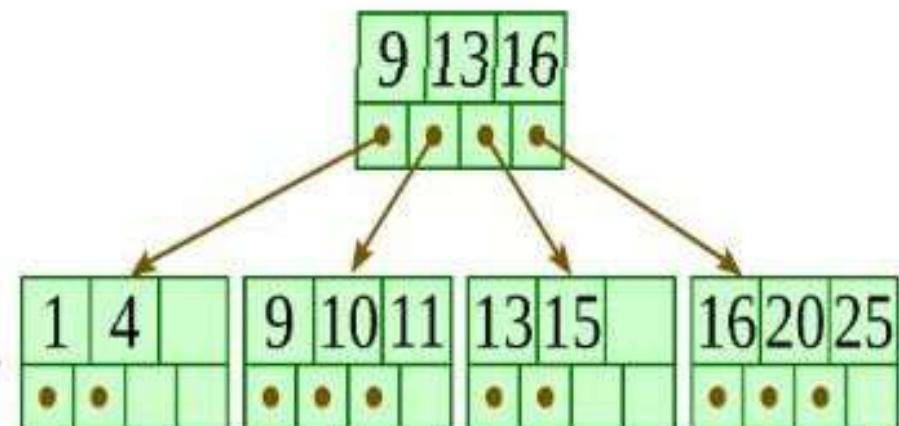
- **Insertion of node in a B+**

■

Insert 10:



Insert 11:



## **Deletion in B+Trees**

Deletion in B+ Trees is just not deletion but it is a combined process of Searching, Deletion, and Balancing. In the last step of the Deletion Process, it is mandatory to balance the B+ Trees, otherwise, it fails in the property of B+ Trees.

# B+

- **Deletion of a node in a B+ Tree:**

- Descend to the leaf where the key exists.
- Remove the required key and associated reference from the node.
- If the node still has enough keys and references to
- If the node has too few keys to satisfy the invariants, but its next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. Repair the keys in the level above to represent that these nodes now have a different “split point” between them; this involves simply changing a key in the levels above, without deletion or insertion.

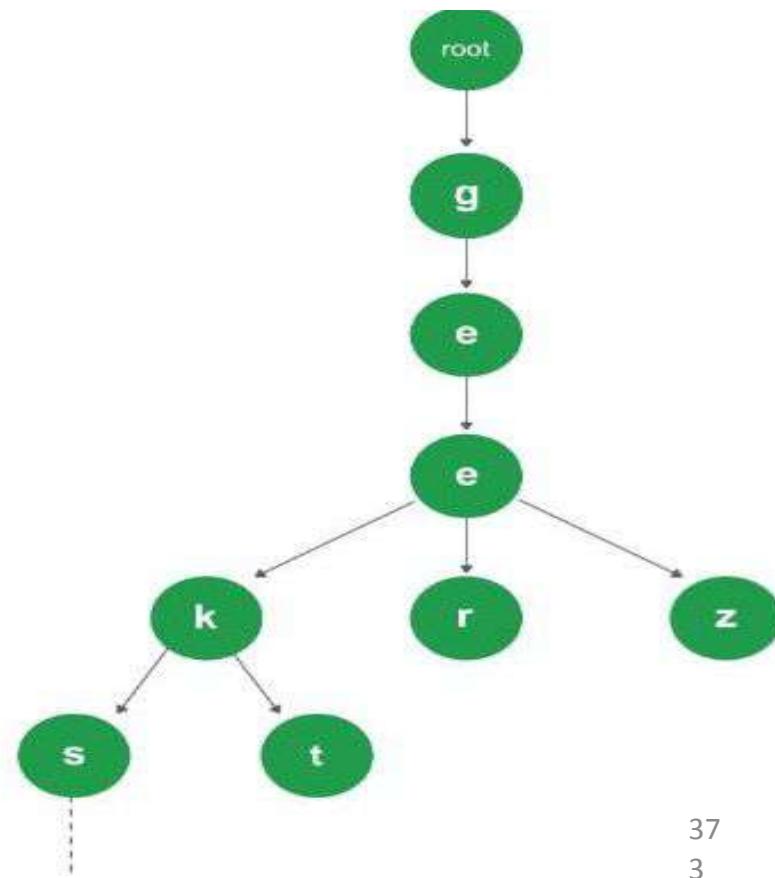
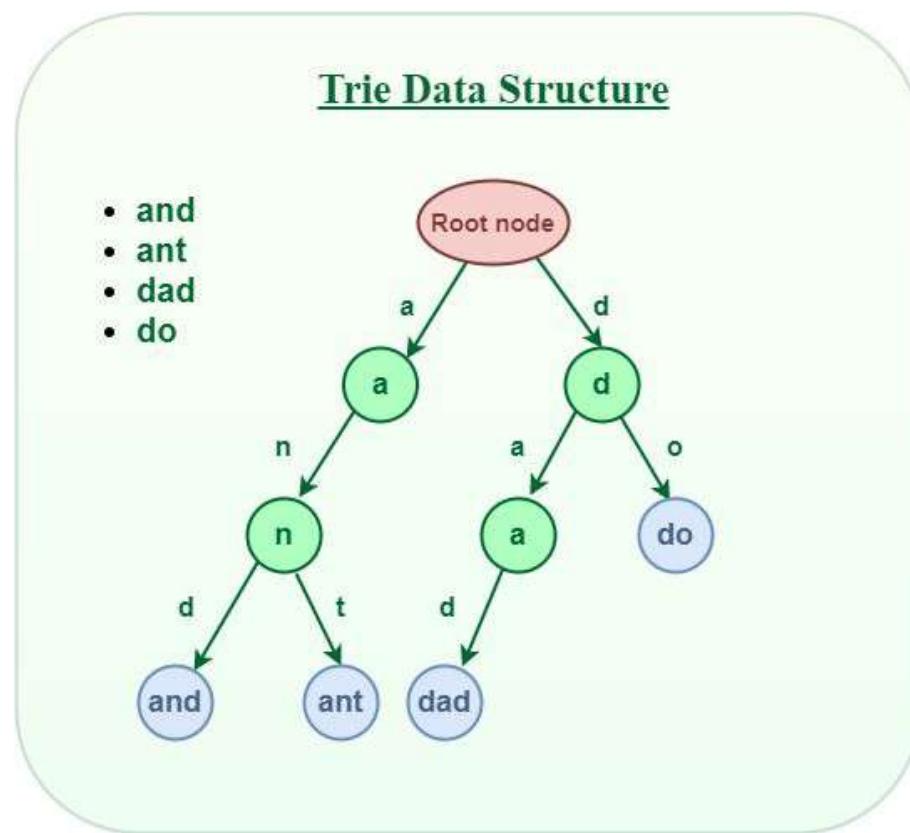
# B+

## • Deletion of a node in a Tree:

- If the node has too few keys to satisfy the invariant, and the next oldest or next youngest sibling is at the minimum for the invariant, then merge the node with its sibling; if the node is a non-leaf, we will need to incorporate the “split key” from the parent into our merging.
- In either case, we will need to repeat the removal algorithm on the parent node to remove the “split key” that previously separated these merged nodes — unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).

# Trie

- A trie or keyword tree, is a data structure that **stores strings as data items** that can be organized in a visual graph.



# Why Trie Data Structure?

- **Searching trees** in general favor keys which are of **fixed size** since this leads to efficient storage management.
- However in case of applications which are **retrieval based** and which call for keys **varying length**, tries **provide better options**.
- Tries are also called as **Lexicographic Search trees**.

# Tries

- All the search trees are used to store the collection of numerical values but they are not suitable for storing the collection of words or strings.
- Trie is a data structure which is used to **store the collection of strings** and makes searching of a pattern in words more easy.
- The term ***trie*** came from the word **retrieval**. Trie data structure makes retrieval of a string from the collection of strings more easily and faster.
- In computer science, Trie is also called as **Prefix/Radix Tree** and some times **Digital Tree**.

<https://www.geeksforgeeks.org/introduction-to-trie-data-structure-and-algorithm-tutorials/>

# Properties of a tries

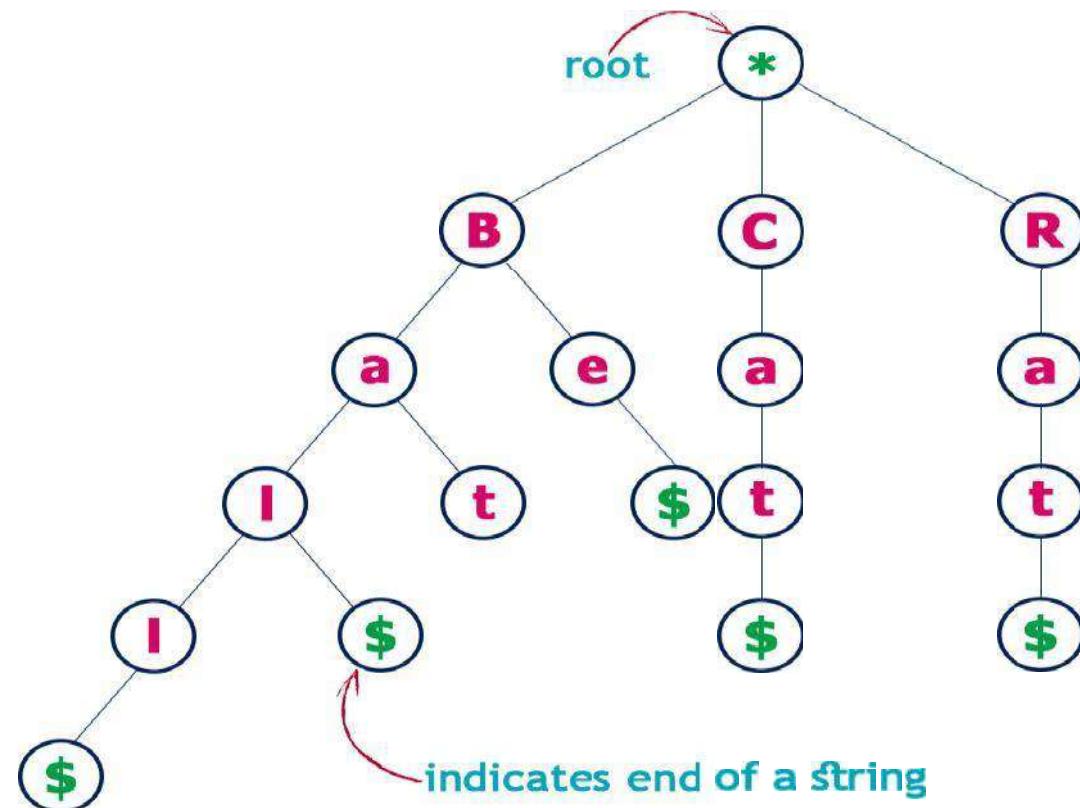
- A multi-way tree.
- Each node has from 1 to n children.
  - There is one root node in each Trie.
  - Each **node** of a Trie represents a **string** and each **edge** represents a **character**.
- Each leaf nodes corresponds to the stored string, which is a concatenation of characters on a path from the root to this node.
  - Each path from the root to any node represents a word or string.
- Trie data structure can contain any number of characters including alphabets, numbers, and special characters.

# Tries

Consider the following list of strings to construct Trie

Cat, Bat, Ball, Rat, Cap & Be

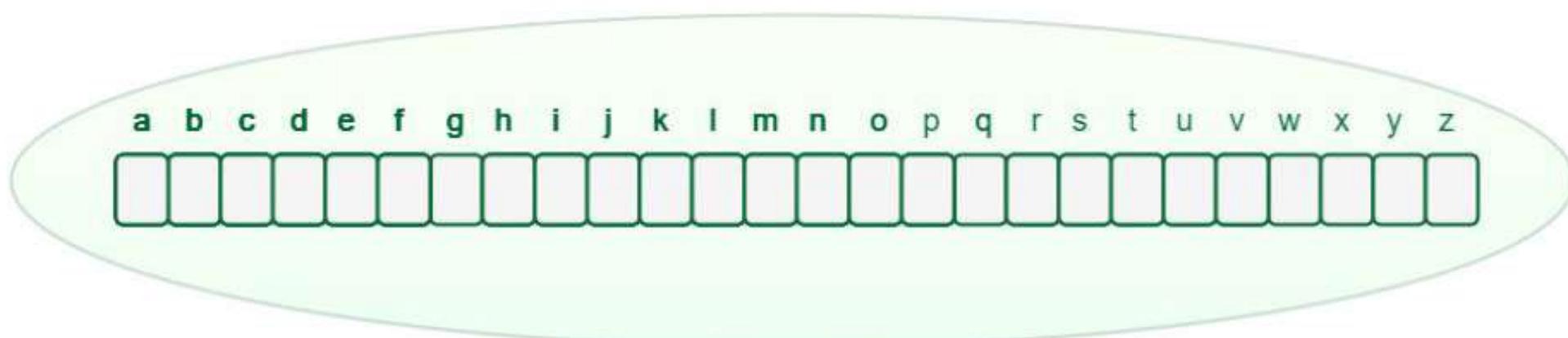
Trie follows some property that If two strings have a **common prefix** then they will have the **same ancestor in the trie**. A trie can be used to sort a collection of strings alphabetically as well as search whether a string with a given prefix is present in the trie or not.



## How does Trie Data Structure work?

We already know that the Trie data structure can contain any number of characters including **alphabets**, **numbers**, and **special characters**. But for this article, we will discuss strings with characters **a-z**. Therefore, only 26 pointers need for every node, where the **0th** index represents 'a' and the **25th** index represents 'z' characters.

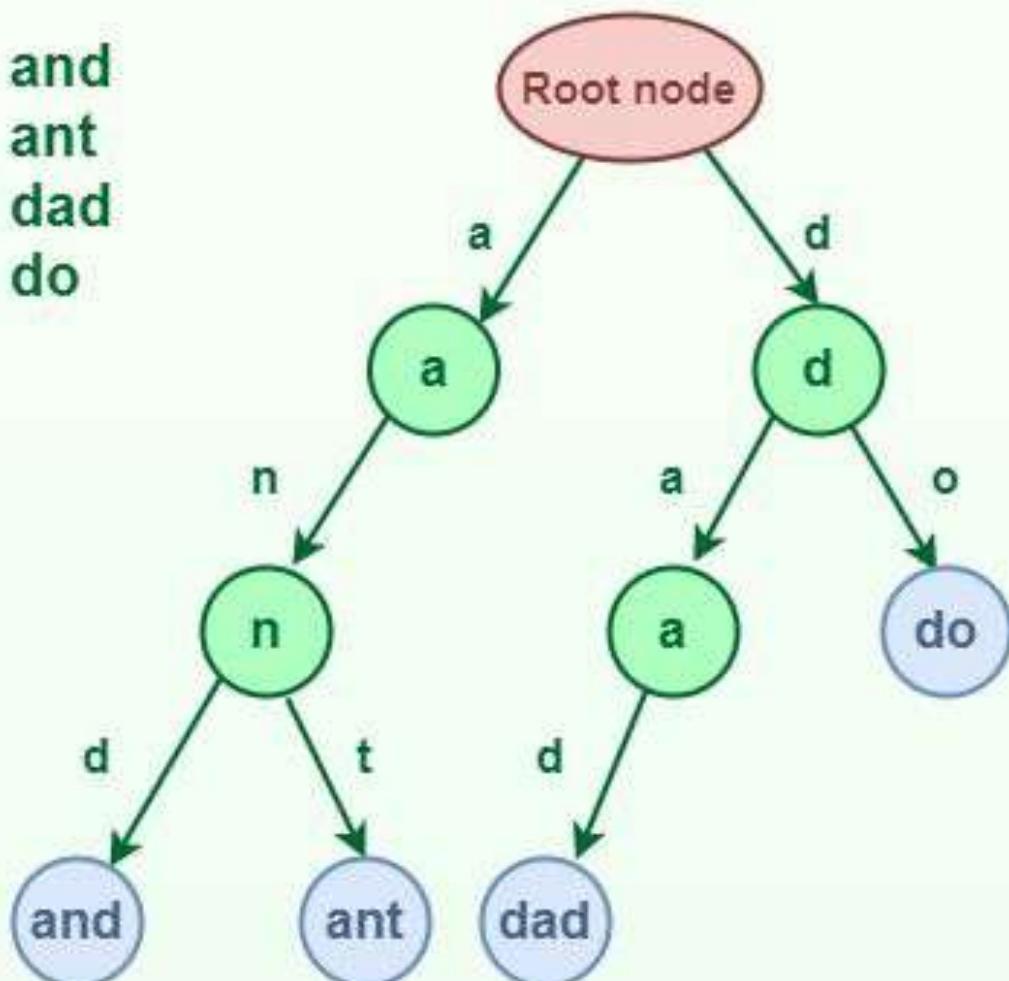
Any **lowercase English word** can start with **a-z**, then the next letter of the word could be **a-z**, the third letter of the word again could be **a-z**, and so on. So for storing a word, we need to take an array (container) of size **26** and initially, all the characters are empty as there are no words and it will look as shown below.



*An array of pointers inside every Trie node*

## Trie Data Structure

- and
- ant
- dad
- do



Let's see how a word "**and**" and "**ant**" is stored in the Trie data structure:

1. Store "**and**" in Trie data structure:

- The word "**and**" starts with "**a**", So we will mark the position "**a**" as filled in the Trie node, which represents the use of "**a**".
- After placing the first character, for the second character again there are **26 possibilities**, So from "**a**", again there is an array of size **26**, for storing the 2nd character.
- The second character is "**n**", So from "**a**", we will move to "**n**" and mark "**n**" in the **2nd** array as used.
- After "**n**", the 3rd character is "**d**", So mark the position "**d**" as used in the respective array.

2. Store "**ant**" in the Trie data structure:

- The word "**ant**" starts with "**a**" and the position of "**a**" in the root node has already been filled. So, no need to fill it again, just move to the node '**a**' in Trie.
- For the second character '**n**' we can observe that the position of '**n**' in the '**a**' node has already been filled. So, no need to fill it again, just move to node '**n**' in Trie.
- For the last character '**t**' of the word, The position for '**t**' in the '**n**' node is not filled. So, filled the position of '**t**' in '**n**' node and move to '**t**' node.

After storing the word "**and**" and "**ant**" the Trie will look like this:

Let us try to Insert “and” & “ant” in this Trie:

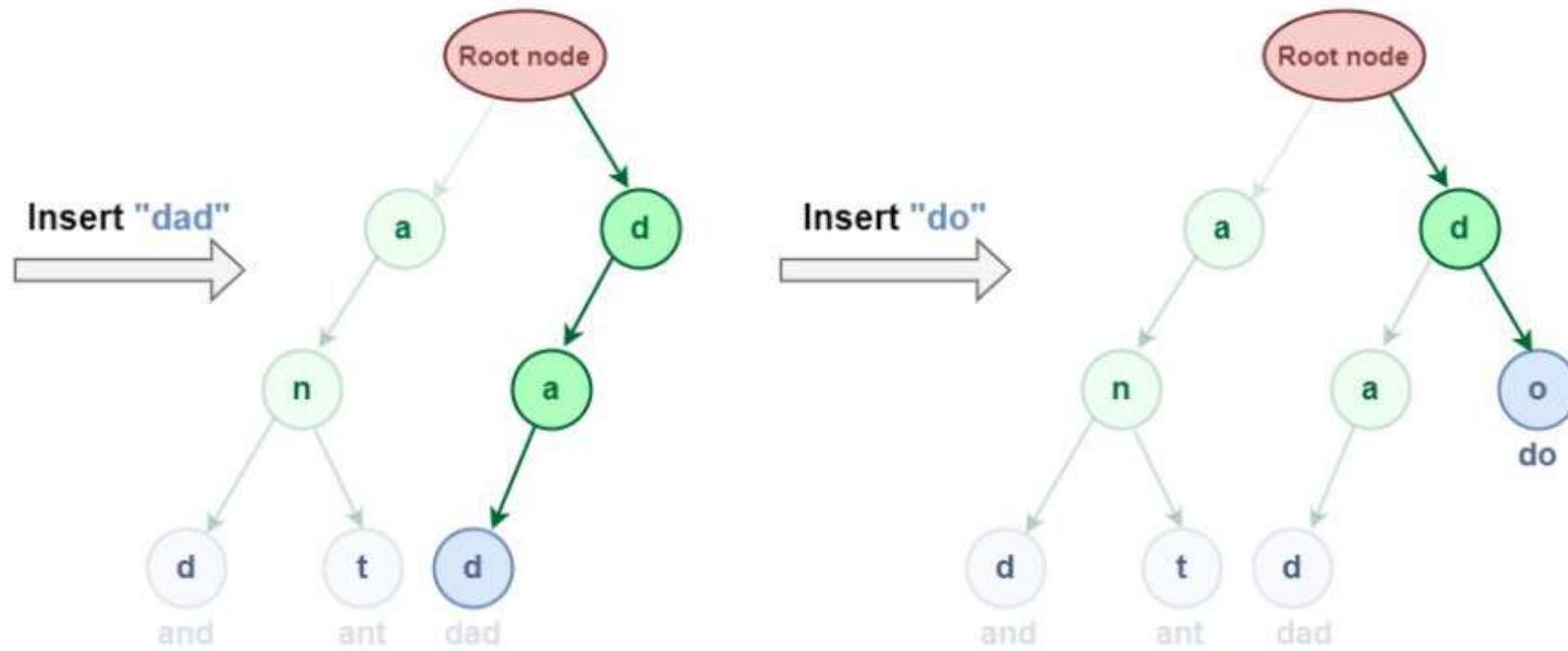


Insert “and” & “ant”

*Insert “and” & “ant”*

From the above representation of insertion, we can see that the word “and” & “ant” have shared some common node (i.e “an”) this is because of the property of the Trie data structure that If two strings have a common prefix then they will have the same ancestor in the trie.

Now let us try to Insert “dad” & “do”:



## 2. Searching in Trie Data Structure:

Search operation in Trie is performed in a similar way as the insertion operation but the only difference is that whenever we find that the array of pointers in **curr node** does not point to the **current character** of the **word** then return false instead of creating a new node for that current character of the word.

This operation is used to search whether a string is present in the Trie data structure or not. There are two search approaches in the Trie data structure.

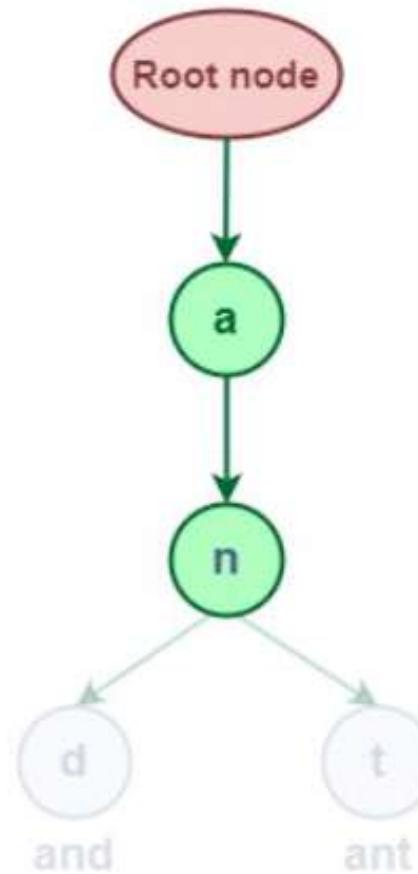
1. Find whether the given word exists in Trie.
2. Find whether any word that starts with the given prefix exists in Trie.

There is a similar search pattern in both approaches. The first step in searching a given word in Trie is to convert the word to characters and then compare every character with the trie node from the root node. If the current character is present in the node, move forward to its children. Repeat this process until all characters are found.

## 2.1 Searching Prefix in Trie Data Structure:

Search for the prefix “an” in the Trie Data Structure.

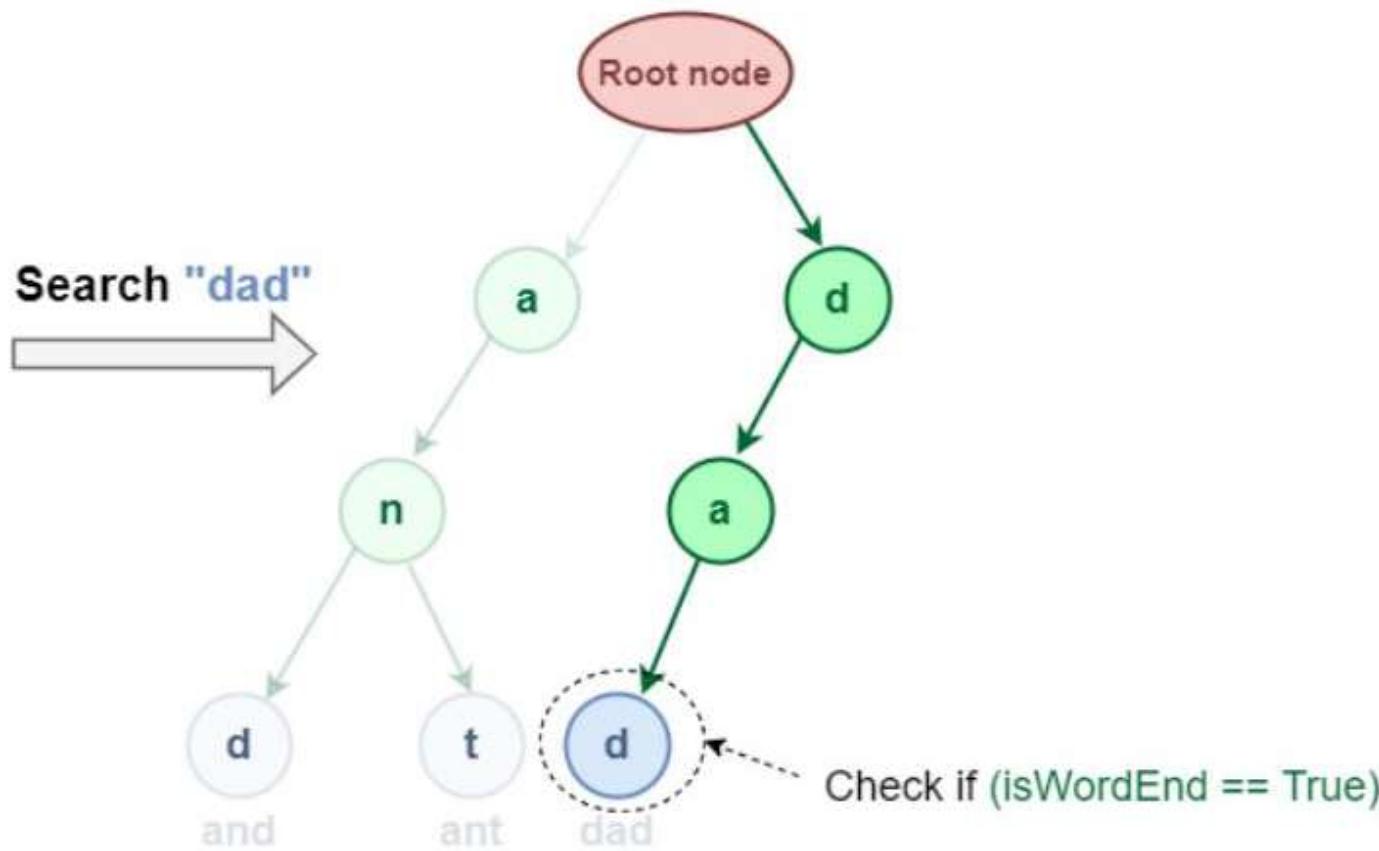
### Search for prefix “an” in Trie



*Search for the prefix “an” in Trie*

## 2.2 Searching Complete word in Trie Data Structure:

It is similar to prefix search but additionally, we have to check if the word is ending at the last character of the word or not.



Search "dad" in the Trie data structure

# TRIES IN AUTO COMPLETE

- Since a trie is a tree-like data structure in which each node contains an array of pointers, one pointer for each character in the alphabet.
  - Starting at the root node, we can trace a word by following pointers corresponding to the letters in the target word.
  - Starting from the root node, you can check if a word exists in the trie easily by following pointers corresponding to the letters in the target word.
-

# AUTO COMPLETE

- Auto-complete functionality is used widely over the internet and mobile apps. A lot of websites and apps try to complete your input as soon as you start typing.
- All the descendants of a node have a common prefix of the string associated with that node.

# AUTO COMPLETE IN GOOGLE SEARCH



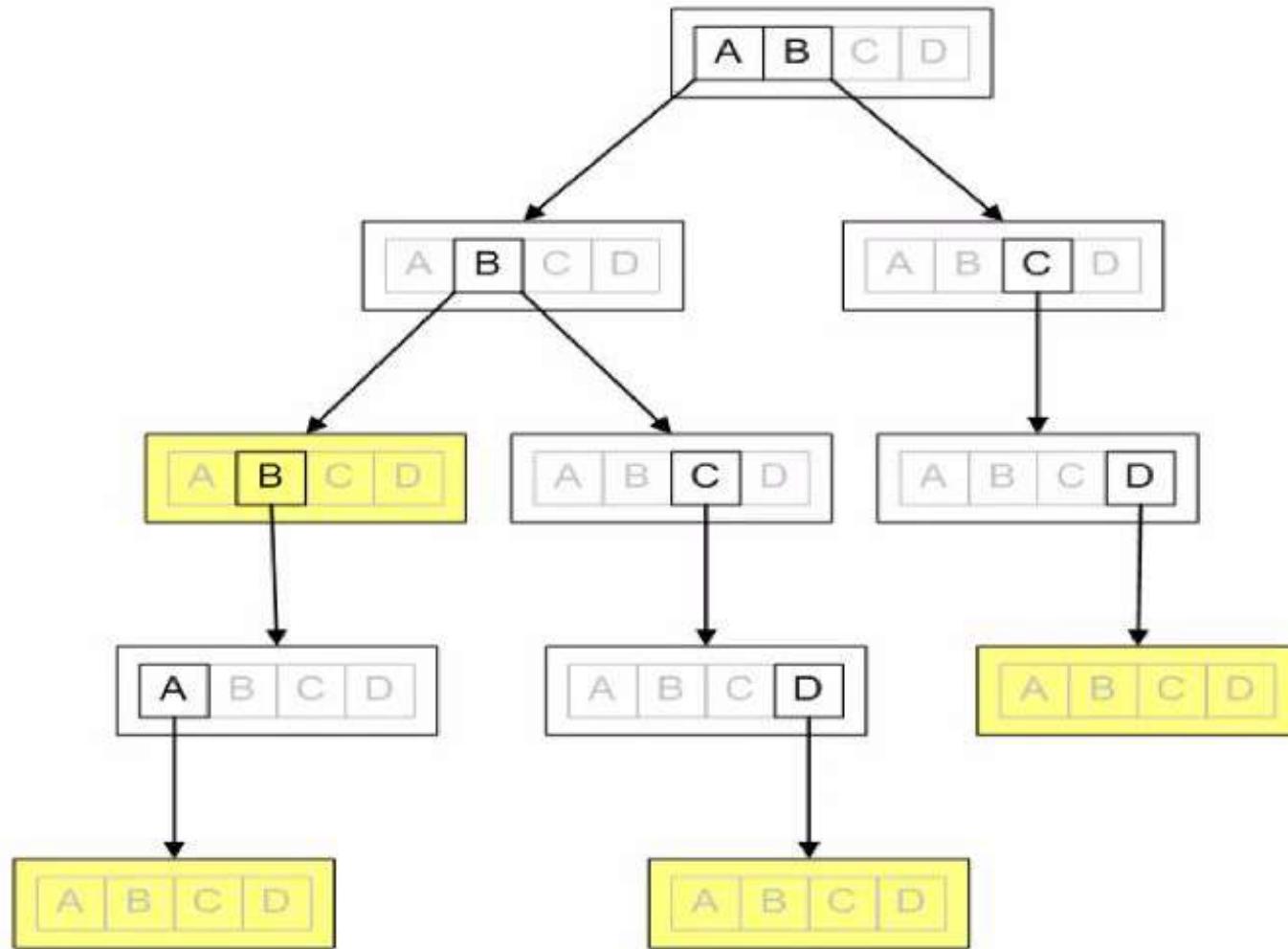
data  
dataone  
datawind  
data mining  
database management system  
data.bsnl.in  
data warehousing  
data structures  
data entry jobs in chennai  
database  
data flow diagram

background image

Privacy About Google

# WHY TRIES IN AUTO COMPLETE

- Implementing auto complete using a trie is easy.
- We simply trace pointers to get to a node that represents the string the user entered. By exploring the trie from that node down, we can enumerate all strings that complete user's input.



# AUTOMATIC COMMAND COMPLETION

- When using an operating system such as Unix or DOS, we type in system commands to accomplish certain tasks. For example, the Unix and DOS command `cd` may be used to change the current directory.

# Commands that have the prefix “ps”

- ps2ascii    ps2pdf    psbook    psmandup    psselect
- ps2epsi    ps2pk    pscal    psmerge    pstopnm
- ps2frag    ps2ps    psidtopgm    psnup    pstops
- ps2gif    psbb    pslatex    psresize    pstruct
- Figure 10 Commands that begin with "ps"

We can simplify the task of typing in commands by providing a command completion facility which automatically types in the command suffix once the user has typed in a long enough prefix to uniquely identify the command. For instance, once the letters `psi` have been entered, we know that the command must be `psidtopgm` because there is only one command that has the prefix `psi`. In this case, we replace the need to type in a 9 character command name by the need to type in just the first 3 characters of the command!

# LONGEST PREFIX MATCHING

- Longest prefix match (also called Maximum prefix length match) refers to an algorithm used by routers in Internet Protocol (IP) networking to select an entry from a routing table .
- Because each entry in a routing table may specify a network, one destination address may match more than one routing table entry. The most specific table entry — the one with the highest subnet mask — is called the longest prefix match. It is called this because it is also the entry where the largest number of leading address bits in the table entry match those of the destination address.

For example, consider this IPv4 routing table (CIDR notation is used):

192.168.20.16/28

192.168.0.0/16

When the address 192.168.20.19 needs to be looked up, both entries in the routing table "match". That is, both entries contain the looked up address. In this case, the longest prefix of the candidate routes is 192.168.20.16/28, since its subnet mask (/28) is higher than the other entry's mask (/16), making the route more specific.

- A network browser keeps a history of the URLs of sites that you have visited. By organizing this history as a trie, the user need only type the prefix of a previously used URL and the browser can complete the URL.

## Speed Dial - Opera

File Edit View Bookmarks Widgets Tools Help

X Trie - Wikipedia, the free encyclopedia X Hyphenation algorithm... X Gmail: Email from Google... X Speed Dial

Bookmarks > Amazon.com

★ <http://www.ask.com/?o=101483&l=dis> Ask.com

Bookmarks > Ask.com

★ <http://www.kelkoo.com/> Kelkoo

Bookmarks > Kelkoo

W <http://en.wikipedia.org/wiki/Trie> Trie - Wikipedia, the free encyclopedia

study can be found at <http://www.nedprod.com/programs/portable/nedtries/> [ edit ] Compressing tries When the trie is mostly static, i.e. all insertions or deletions o...

http://www.brothersoft.com/mp3-cutter-132871.html Download Free MP3 Cutter, MP3 Cutter 1.9 Download

Download Google Chrome [www.google.com/](http://www.google.com/) Chrome Searching is fast and easy with Google's web browser. Mobile Recharge Offer [www.Paytm.com](http://www.Paytm.com) Get Free Co...

http://www.brothersoft.com/mp3-cutter-download-132871.html Free MP3 Cutter Download

seconds. Download KeyShot Now! [www.keyshot.com](http://www.keyshot.com) Free MP3 Converter Convert all audio songs to MP3 free Download now! [www.ultimatumz.com](http://www.ultimatumz.com) Graphic Desig...

http://www.google.co.in/search?q=mp3+cutter+freeware&hl=en&gbv=2&gs\_sm=3&gs\_upl=32199|57362|0|58173|36|0|16|16|0|670|1809|5-3|3|0&oq=

mp3 cutter freeware - Google Search

-1 Audio Layer 3 (MP3) file. [www.free-mp3-cutter.org/](http://www.free-mp3-cutter.org/) - Cached - Similar Mp3 cutter Free Download Mp3 cutter Free Download, Mp3 cutter Software Collection ...

http://www.google.co.in/search?hl=en&gbv=2&gs\_sm=3&gs\_upl=32199|57362|0|58173|36|0|16|16|0|670|1809|5-3|3|0&oq=

download free mp3 cutter for pc - Google Search

fragment your big audio ... [www.toggle.com/lv/group/view/kl39886/Power\\_MP3\\_Cutter.htm](http://www.toggle.com/lv/group/view/kl39886/Power_MP3_Cutter.htm) - Cached - Similar All Free MP3 Cutter 2.4.3 Download 15 Feb 2012 ...

http://www.google.co.in/search?hl=en&gbv=2&gs\_sm=3&gs\_upl=32199|57362|0|58173|36|0|16|16|0|670|1809|5-3|3|0&oq=

download free mp3 cutter and joiner - Google Search

Joiner (Size:2.62 MB) ... [www.audiotoolsfactory.com/mp3cutter/mp3cutter.htm](http://www.audiotoolsfactory.com/mp3cutter/mp3cutter.htm) - Cached - Similar Power MP3 Cutter Joiner - SagaSoft Power MP3 Cutter Joiner, m...

http://www.google.co.in/search?q=download+mp3+cutter+software+free&hl=en&gbv=2&gs\_sm=3&gs\_upl=32199|57362|0|58173|36|0|16|16|0|670|1809|5-3|3|0&oq=

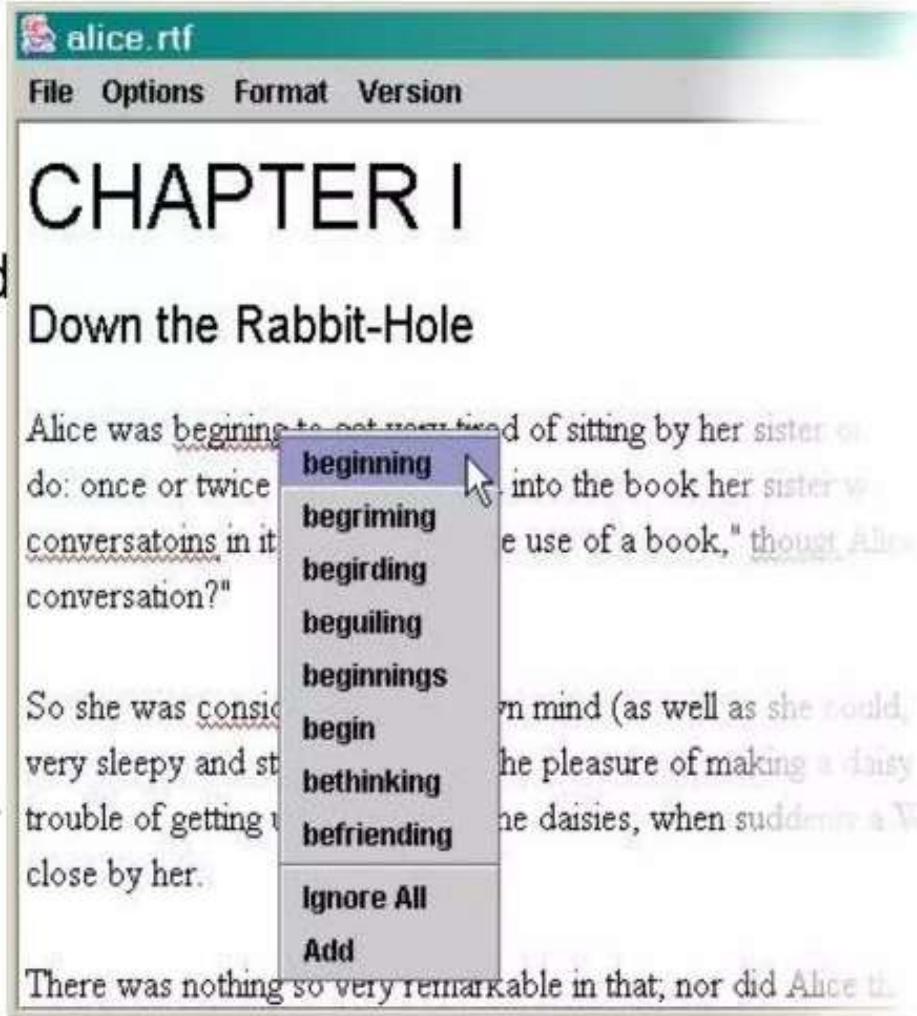
download mp3 cutter software free - Google Search

& editing operations in ... [www.free-mp3-cutter.com/](http://www.free-mp3-cutter.com/) - Cached - Similar Free mp3 cutter software Free Download Free mp3 cutter software Free Download, Free ...

http://www.free-mp3-cutter.org/ Free MP3 Cutter

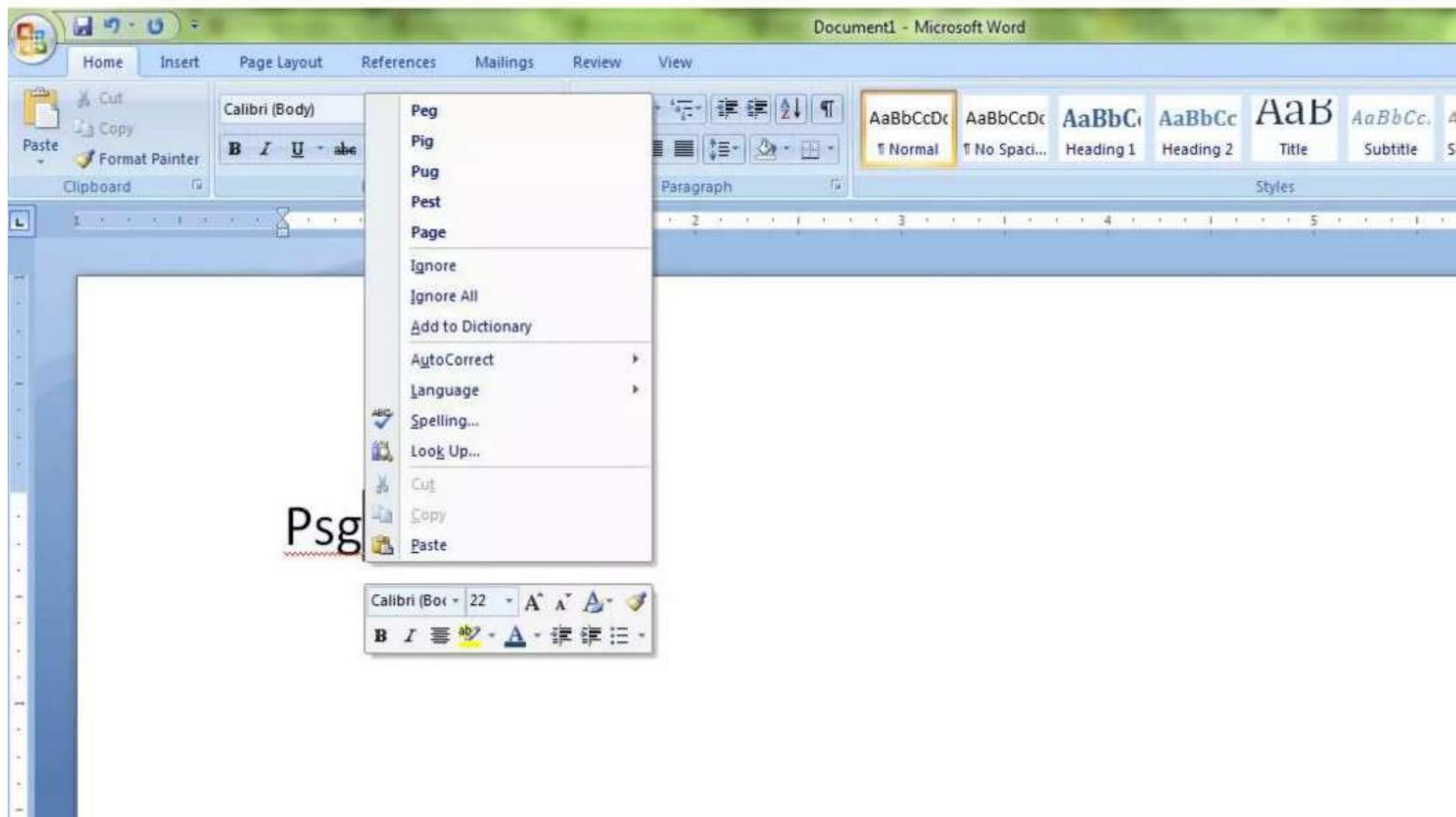
# SPELL CHECKERS

- Spell checkers are ubiquitous. Word processors have spell checkers, as do browser-based e-mail clients. They all work the same way: a dictionary is stored in some data structure, then each word of input is submitted to a search in the data structure, and those that fail are flagged as spelling errors



## SPELL CHECKERS

- There are many appropriate data structures to store the word list, including a sorted array accessed via binary search, a hash table, or a bloom filter. In this exercise you are challenged to store the word list character-by-character in a trie.



## Disadvantages of Trie data structure:

- The main disadvantage of the trie is that it takes a lot of memory to store all the strings. For each node, we have too many node pointers which are equal to the no of characters in the worst case.

# Tree Applications

- Trees are used to store simple as well as complex data. Here simple means an integer value, character value and complex data means a structure or a record.
- Trees are often used for implementing other types of data structures like hash tables, sets, and maps.
- A self-balancing tree, Red-black tree is used in kernel scheduling, to preempt massively multiprocessor computer operating system use.
- Another variation of tree, B-trees are prominently used to store tree structures on disc. They are used to index a large number of records.
- B-trees are also used for secondary indexes in databases, where the index facilitates a select operation to answer some range criteria.
- Trees are an important data structure used for compiler construction.
- Trees are also used in database design.
- Trees are used in file system directories.
- Trees are also widely used for information storage and retrieval in symbol tables.

# Tree Applications

- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- Heap is a kind of tree that is used for heap sort.
- A modified version of a tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write.

# Applications

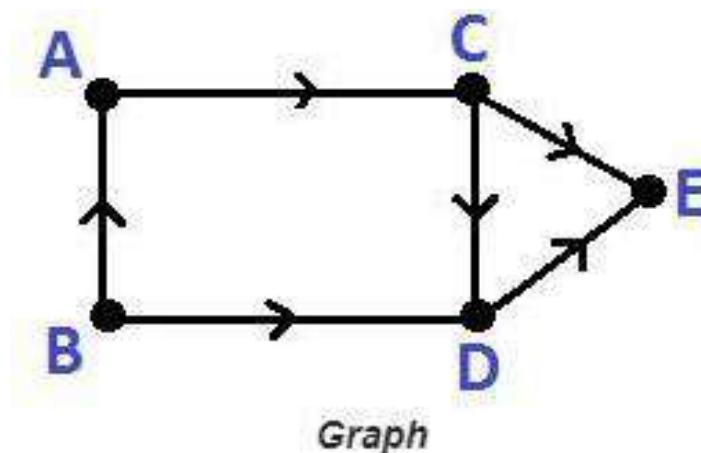
- **Spanning trees:** It is the shortest path tree used in the routers to direct the packets to the destination.
- **Binary Search Tree:** It is a type of tree data structure that helps in maintaining a sorted stream of data.
- Full Binary tree
- Complete Binary tree
- Skewed Binary tree
- Strictly Binary tree
- Extended Binary tree
- **Storing hierarchical data:** Tree data structures are used to store the hierarchical data, which means data is arranged in the form of order.
- **Syntax tree:** The syntax tree represents the structure of the program's source code, which is used in compilers.
- **Trie:** It is a fast and efficient way for dynamic spell checking. It is also used for locating specific keys from within a set.
- **Heap:** It is also a tree data structure that can be represented in a form of an array. It is used to implement priority queues.

# Tree Applications

- Directory structure of a file storage
- Structure of an arithmetic expressions
- Used in almost every 3D video game to determine what objects need to be rendered.
- Used in almost every high-bandwidth router for storing router-tables.
- Used in compression algorithms, such as those used by the .jpeg and .mp3 file formats
- Game trees

- **Graphs-**

- A graph stores a collection of items in a **non-linear fashion**.
- Graphs are made up of a finite set of nodes also known as **vertices** and lines that connect them also known as **edges**.
- These are useful for representing real-life systems such as computer networks.



# Types of data structures

- *The different types of Graphs are :*
  - Directed Graph
  - Non-directed Graph
  - Connected Graph
  - Non-connected Graph
  - Simple Graph
  - Multi-Graph

# Graph Basics

- Graphs are collections of nodes connected by edges –  $G = (V,E)$  where  $V$  is a set of nodes and  $E$  a set of edges.
- Graphs are useful in a number of applications including
  - Shortest path problems
  - Maximum flow problems
- Graphs unlike trees are more general for they can have connected components.

# Graph Types

- **Directed Graphs:** A directed graph edges allow travel in one direction.
- **Undirected Graphs:** An undirected graph edges allow travel in either direction.

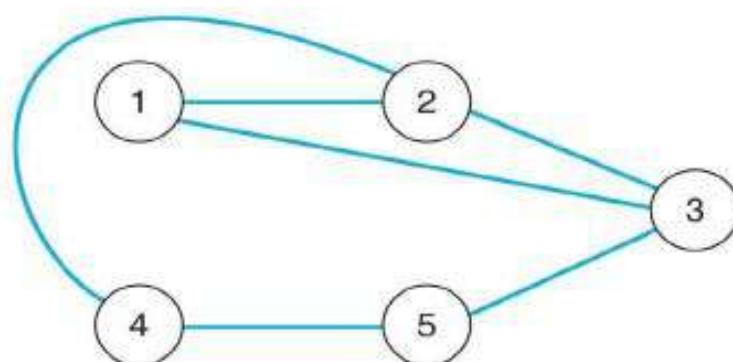


FIGURE 8.1A

The graph  $G = ([1, 2, 3, 4, 5], \{(1, 2), (1, 3), (2, 3), (2, 4), (3, 5), (4, 5)\})$

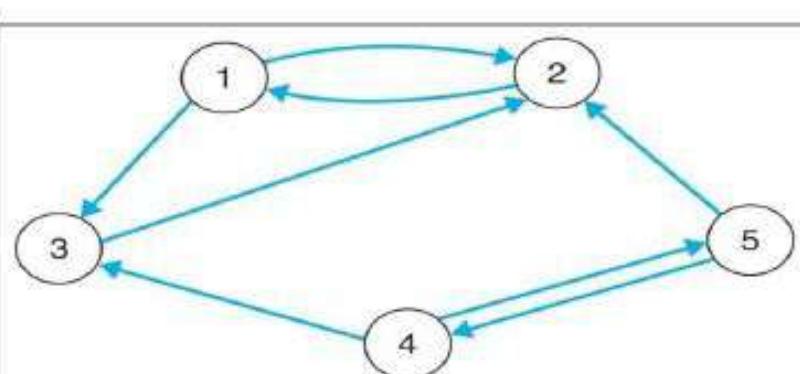


FIGURE 8.1B

The directed graph  $G = ([1, 2, 3, 4, 5], \{(1, 2), (1, 3), (2, 1), (2, 2), (3, 2), (3, 4), (4, 3), (4, 5), (5, 2), (5, 4)\})$

# Graph Terminology

- A graph is an ordered pair  $G=(V,E)$  with a set of vertices or nodes and the edges that connect them.
- A subgraph of a graph has a subset of the vertices and edges.
- The edges indicate how we can move through the graph.
- A path is a subset of  $E$  that is a series of edges between two nodes.
- A graph is connected if there is at least one path between every pair of nodes.

# Graph Terminology

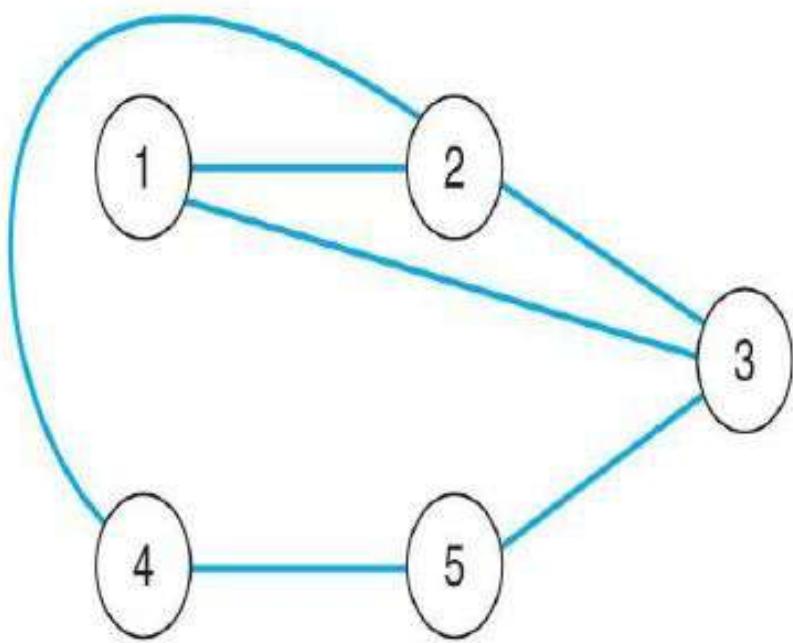
- The length of a path in a graph is the number of edges in the path.
- A complete graph is one that has an edge between every pair of nodes.
- A weighted graph is one where each edge has a cost for traveling between the nodes.
- A cycle is a path that begins and ends at the same node.
- An acyclic graph is one that has no cycles.
- An acyclic, connected graph is also called an unrooted tree

# Data Structures for Graphs

## An Adjacency Matrix

- For an undirected graph, the matrix will be symmetric along the diagonal.
- For a weighted graph, the adjacency matrix would have the weight for edges in the graph, zeros along the diagonal, and infinity ( $\infty$ ) every place else.

# Adjacency Matrix Example 1



■ FIGURE 8.1A

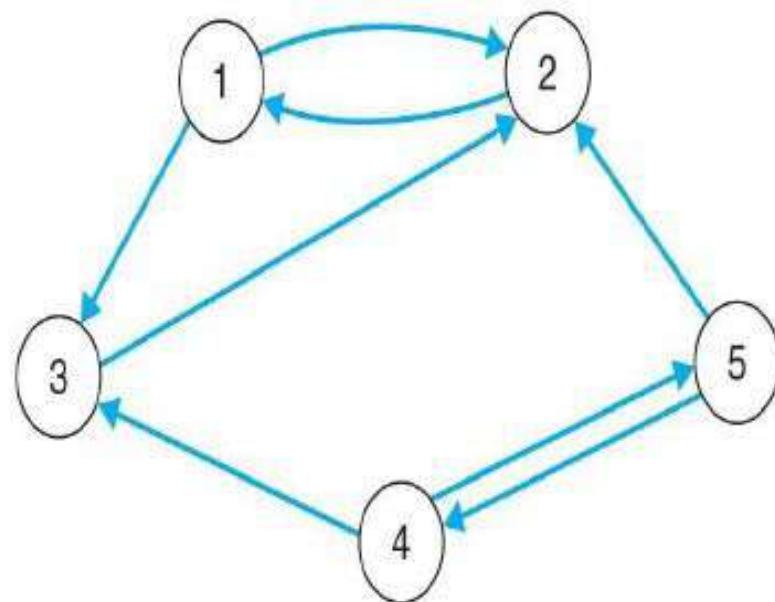
The graph  $G = (\{1, 2, 3, 4, 5\}, \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{3, 5\}, \{4, 5\}\})$

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	0
3	1	1	0	0	1
4	0	1	0	0	1
5	0	0	1	1	0

■ FIGURE 8.2A

The adjacency matrix for the graph in Fig. 8.1(a)

# Adjacency Matrix Example 2



■ FIGURE 8.1B

The directed graph  $G = ([1, 2, 3, 4, 5], \{(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5, 4)\})$

	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	0	0
3	0	1	0	0	0
4	0	0	1	0	1
5	0	1	0	1	0

■ FIGURE 8.2B

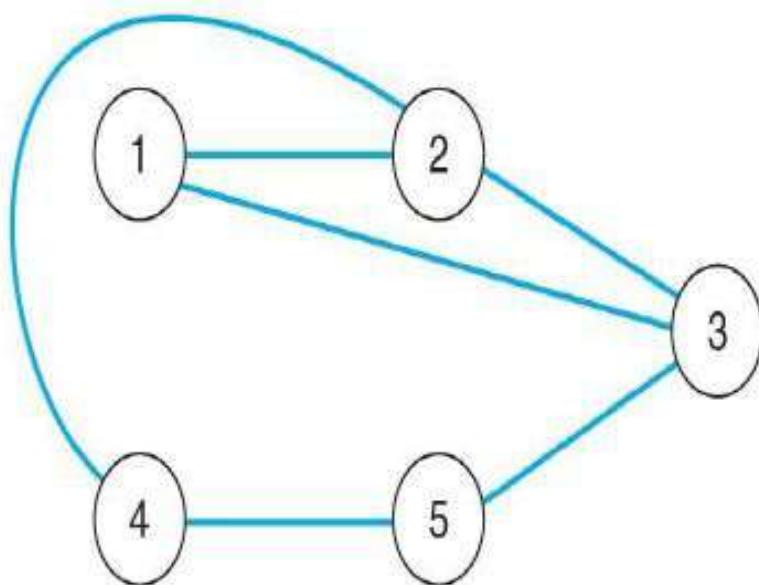
The adjacency matrix for the digraph in Fig. 8.1(b)

# Data Structures for Graphs

## An Adjacency List

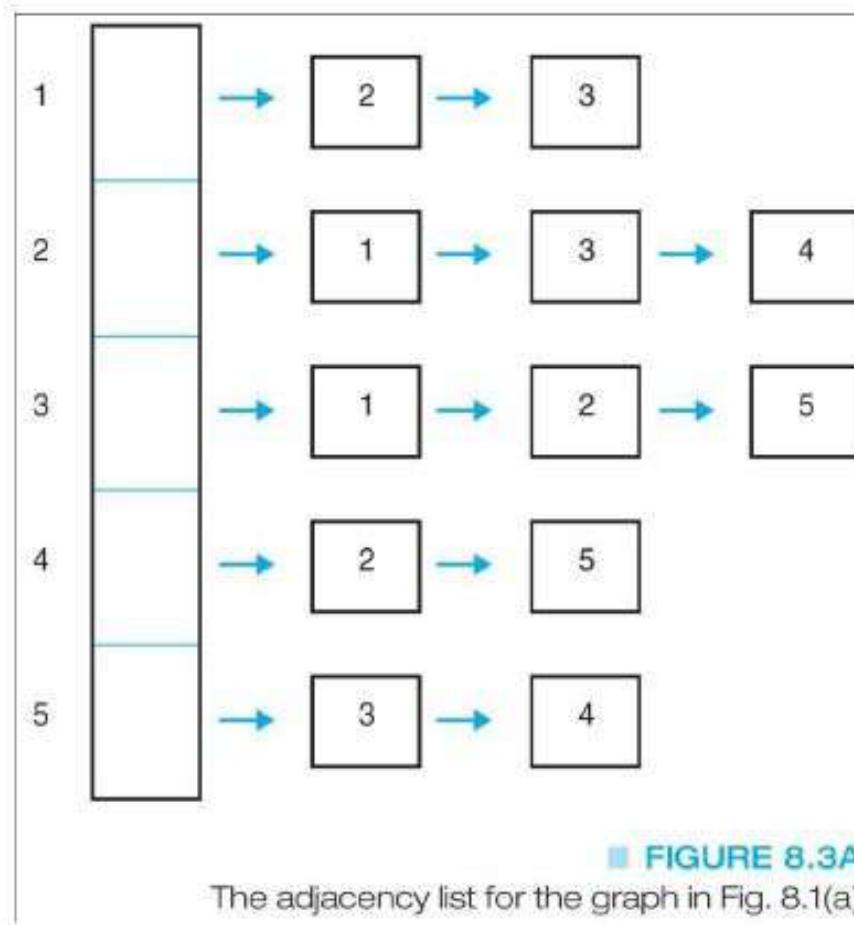
- A list of pointers, one for each node of the graph.
- These pointers are the start of a linked list of nodes that can be reached by one edge of the graph.
- For a weighted graph, this list would also include the weight for each edge.

# Adjacency List Example 1



■ FIGURE 8.1A

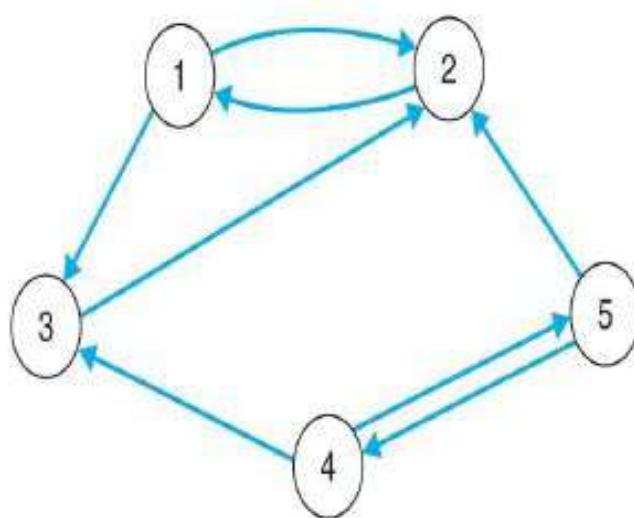
The graph  $G = (\{1, 2, 3, 4, 5\}, \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{3, 5\}, \{4, 5\}\})$



■ FIGURE 8.3A

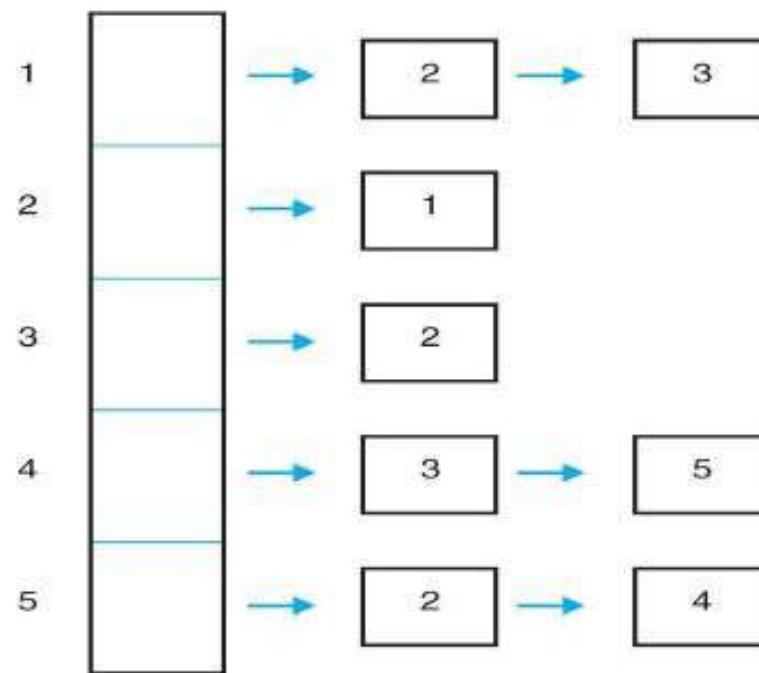
The adjacency list for the graph in Fig. 8.1(a)

# Adjacency List Example 2



■ FIGURE 8.1B

The directed graph  $G = ([1, 2, 3, 4, 5], \{(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5, 4)\})$



■ FIGURE 8.3B

The adjacency list for the graph in Fig. 8.1(b)

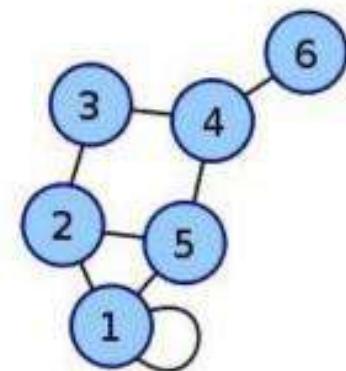
# What is a Graph?

Graphs are Generalization of Trees.

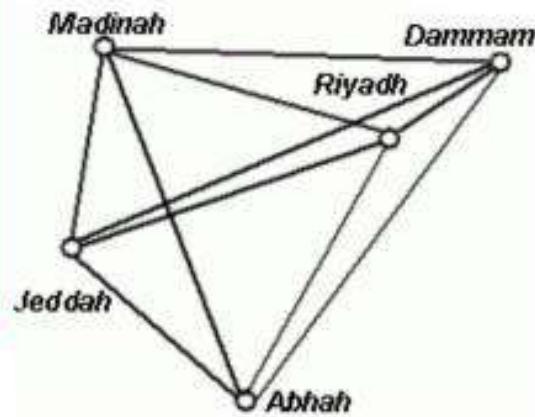
A **simple** graph  $G = (V, E)$  consists of a non-empty set  $V$ , whose members are called the vertices of  $G$ , and a set  $E$  of pairs of distinct vertices from  $V$ , called the edges of  $G$ .

A **simple** graph has *no loop* (An edge that connects a vertex to itself)

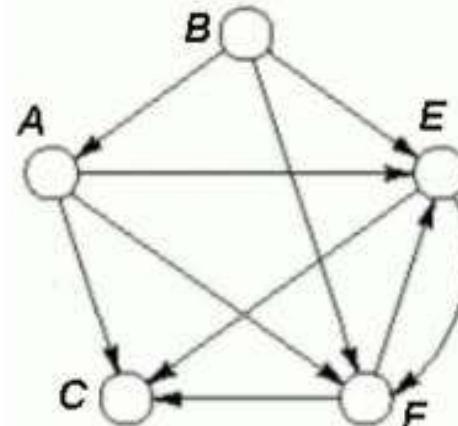
A **simple** graph has *no vertices joined by multiple edges*



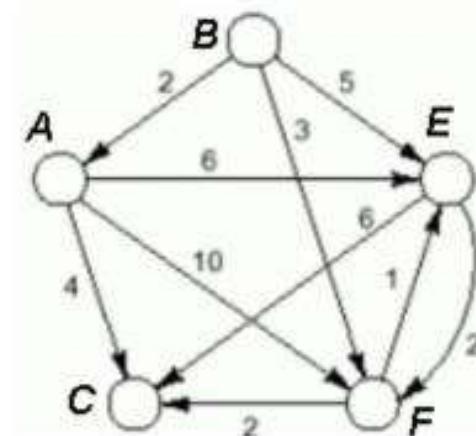
Undirected



Directed (Digraph)



Weighted



SOMAIYA

VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

Somaiya

TRUST 2

# Some Applications of Graphs

- Finding the least congested route between two phones, given connections between switching stations.
- Determining if there is a way to get from one page to another, just by following links.
- Finding the shortest path from one city to another.
- As a traveling sales-person, finding the cheapest path that passes through all the cities that the sales person must visit.
- Determining an ordering of courses so that prerequisite courses are always taken first.
- Networks: Vertices represent computers and edges represent network connections between them.
- World Wide Web: Vertices represent webpages, and edges represent hyperlinks.



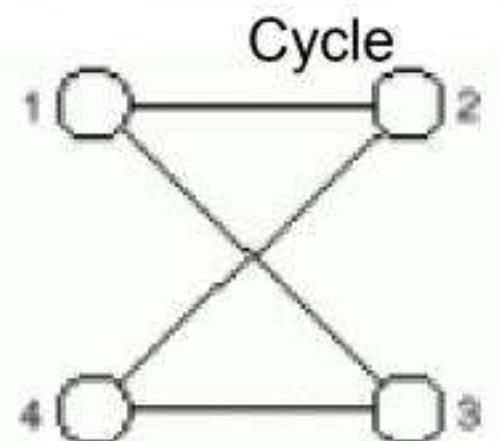
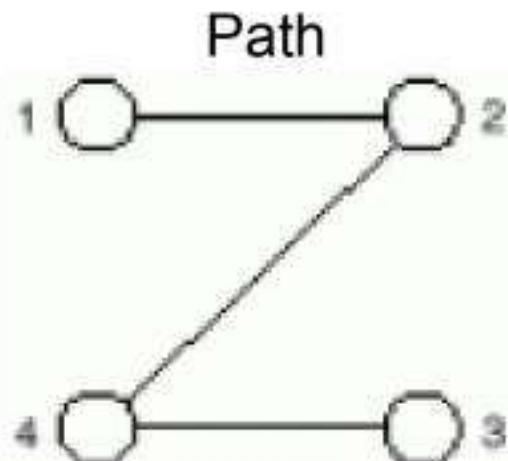
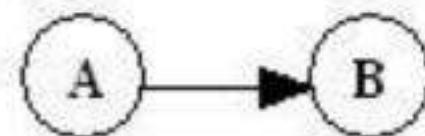
# Graphs Terminologies

**Adjacent Vertex:** A vertex **W** is adjacent to a vertex **V** if there is an edge from **V** to **W**

Example: **B** is adjacent to **A**; but **A** is not adjacent to **B**

**A path:** A path is a sequence of consecutive edges in a graph. The length of the path is the number of edges traversed.

**A cycle (or circuit):** a subset of the **edge set** that forms a **path** such that the first vertex of the path is also the last.



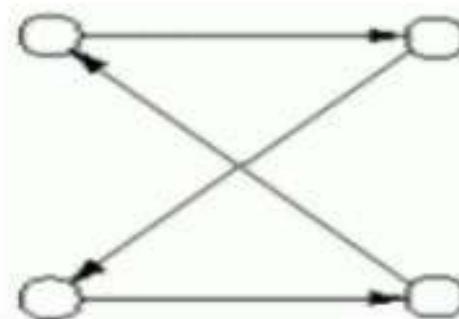
# Graphs Terminologies

- A graph containing no cycles of any length is known as an **acyclic graph**, whereas a graph containing at least one cycle is called **cyclic graph**.

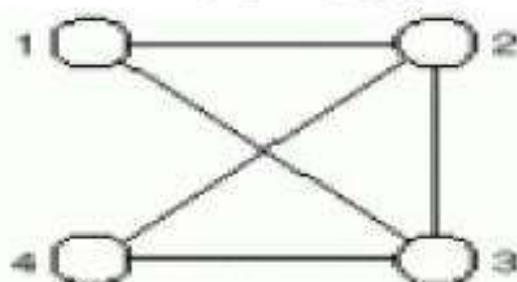
- Connected graph:** A graph is connected if there is a path from any vertex to every other vertex. Alternatively a graph is connected if there is a **path** connecting every pair of vertices.

- An undirected graph that is not connected can be divided into **connected components** (maximal disjoint connected subgraphs). For example, the disconnected graph below is made of two connected components.

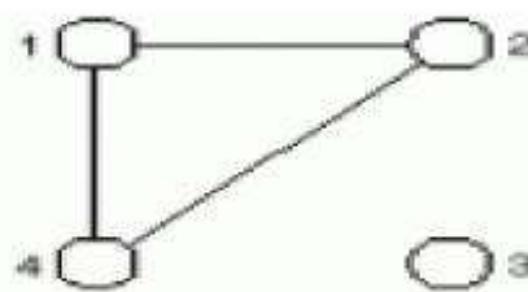
Directed Cyclic



Connected

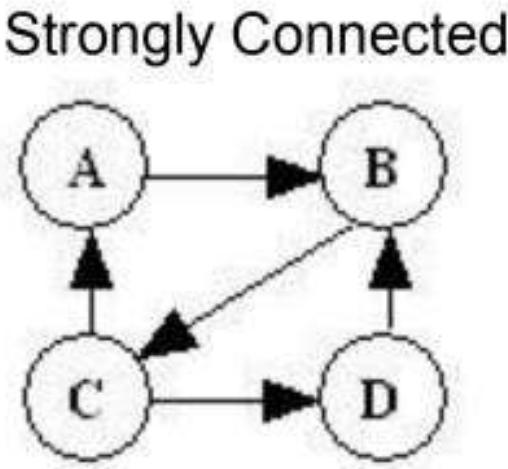


Disconnected

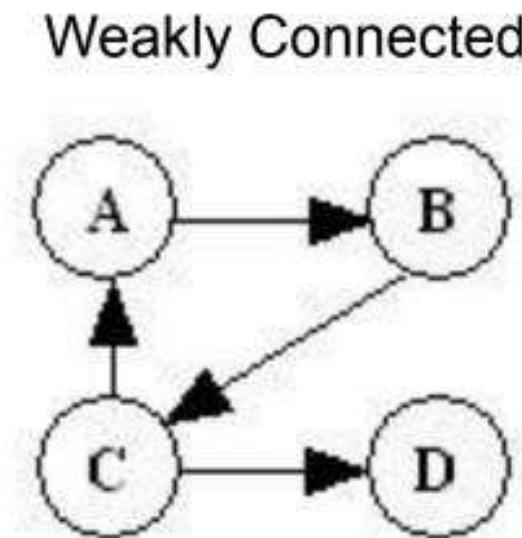


## More Graph Terminologies

- **Strongly Connected:** If connected as a digraph [There is a path from each vertex to every other vertex]

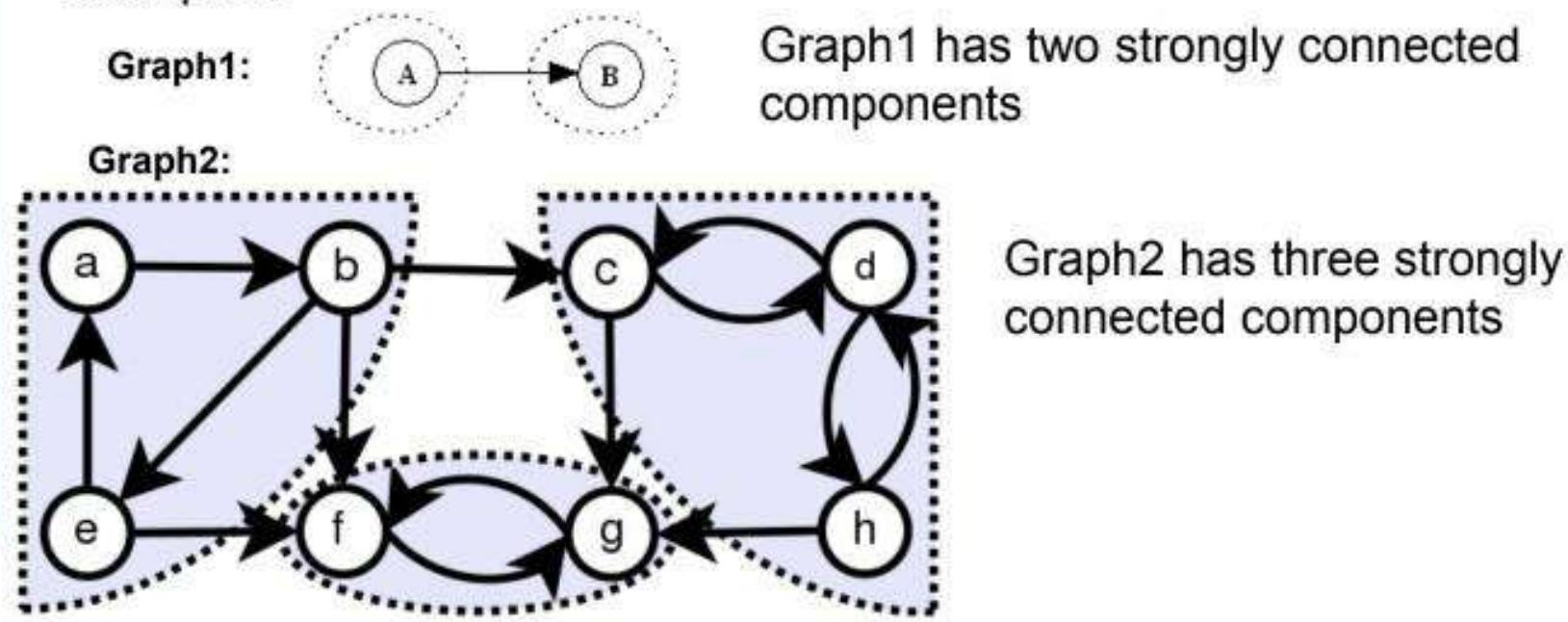


- Weakly connected: If the underlying undirected graph is connected [There is at least one vertex that does not have a path to every other vertex]



# Strongly Connected Components (SCCs)

- A disconnected or weakly connected directed graph can be divided into two or more strongly connected components. Two vertices of a directed graph are in the same strongly connected component if and only if they are reachable from each other.
- Note: If a graph is strongly connected it has only one strongly connected component.
- Examples:



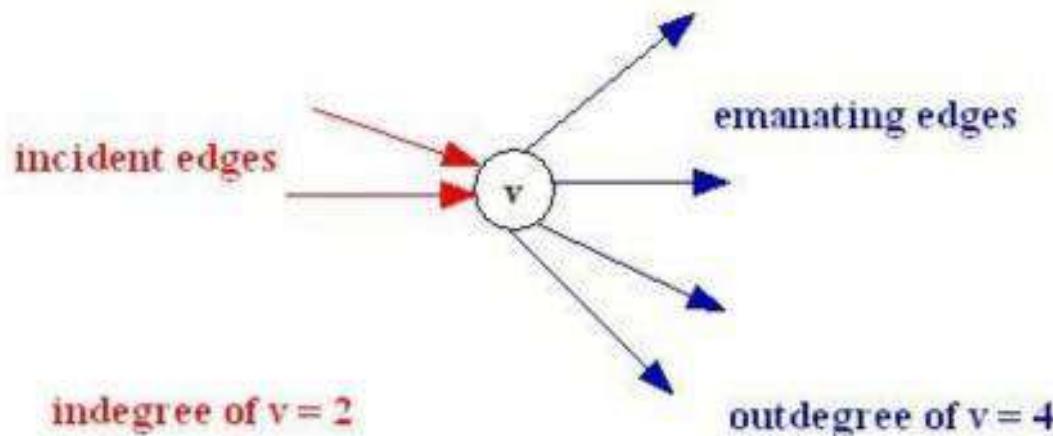
# Further Graph Terminologies

Emanating edge: an edge  $e = (v, w)$  is said to emanate from  $v$ .  
 $A(v)$  denotes the set of all edges emanating from  $v$ .

Incident edge: an edge  $e = (v, w)$  is said to be incident to  $w$ .  
 $I(w)$  denote the set of all edges incident to  $w$ .

Out-degree: number of edges emanating from  $v$ :  $|A(v)|$

In-degree: number of edges incident to  $w$ :  $|I(w)|$



# Further Graph Terminologies

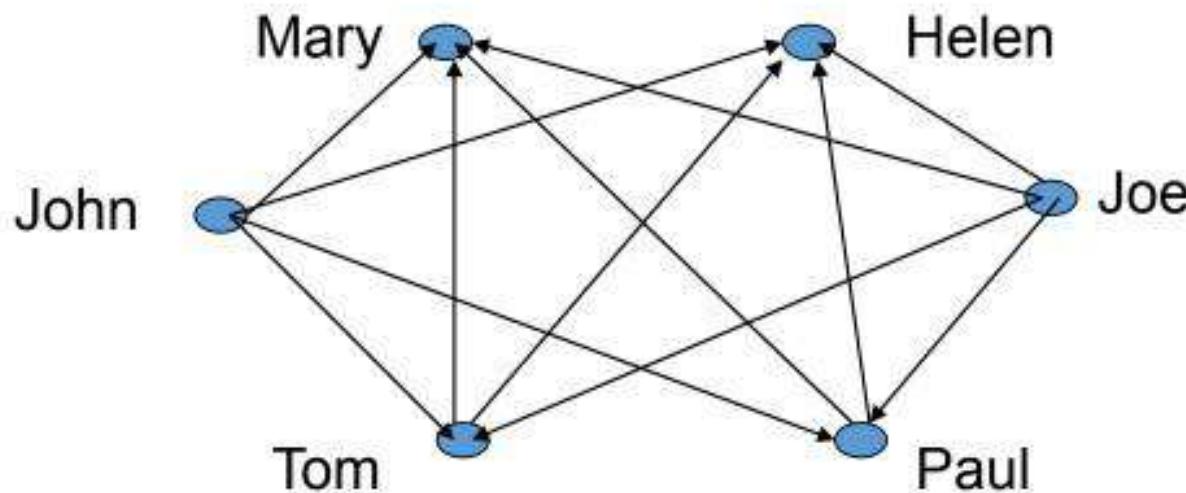
Subgraph: A graph  $G = (V_G, E_G)$  is a subgraph of  $H = (V_H, E_H)$  if  $V_G \subseteq V_H$  and  $E_G \subseteq E_H$ .

A spanning subgraph of  $G$  is a subgraph that contains all the vertices of  $G$

A spanning tree of a connected graph is a spanning subgraph that is a tree

# A “Real-life” Example of a Graph

$V$ =set of 6 people: John, Mary, Joe, Helen, Tom, and Paul, of ages 12, 15, 12, 15, 13, and 13, respectively.  
 $E = \{(x,y) \mid \text{if } x \text{ is younger than } y\}$



# Intuition Behind Graphs

The nodes represent entities (such as people, cities, computers, words, etc.)

Edges  $(x,y)$  represent relationships between entities  $x$  and  $y$ , such as:

- “ $x$  loves  $y$ ”
- “ $x$  hates  $y$ ”
- “ $x$  is a friend of  $y$ ” (note that this is not necessarily reciprocal)
- “ $x$  considers  $y$  a friend”
- “ $x$  is a child of  $y$ ”
- “ $x$  is a half-sibling of  $y$ ”
- “ $x$  is a full-sibling of  $y$ ”

In those examples, each relationship is a different graph

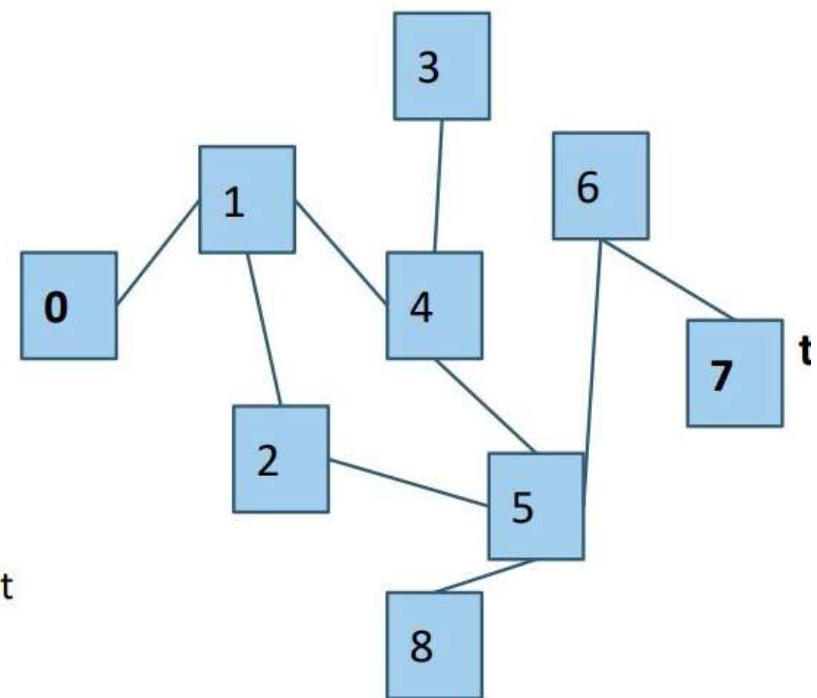
# s-t path Problem

## s-t path problem

- Given source vertex **s** and a target vertex **t**, does there exist a path between **s** and **t**?

Why does this problem matter? Some possible context:

- real life maps and trip planning – can we get from one location (vertex) to another location (vertex) given the current available roads (edges)
- family trees and checking ancestry – are two people (vertices) related by some common ancestor (edges for direct parent/child relationships)
- game states (Artificial Intelligence) can you win the game from the current vertex (think: current board position)? Are there moves (edges) you can take to get to the vertex that represents an already won game?



# Graph Representations

For vertices:

an array or a linked list can be used

For edges:

Adjacency Matrix (Two-dimensional array)

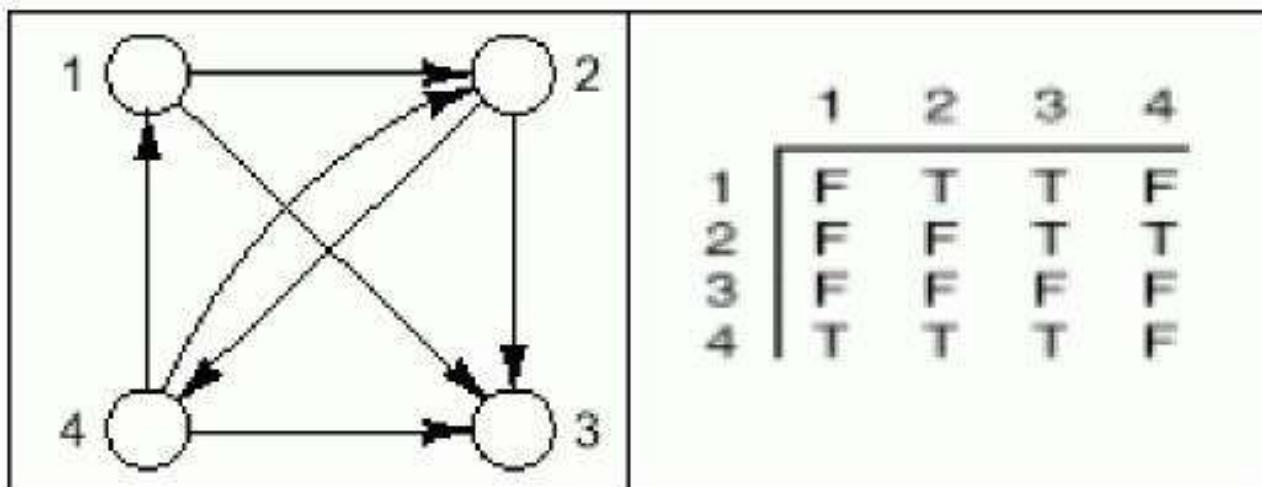
Adjacency List (One-dimensional array of linked lists)

# Adjacency Matrix Representation

Adjacency Matrix uses a 2-D array of dimension  $|V| \times |V|$  for edges. (For vertices, a 1-D array is used)

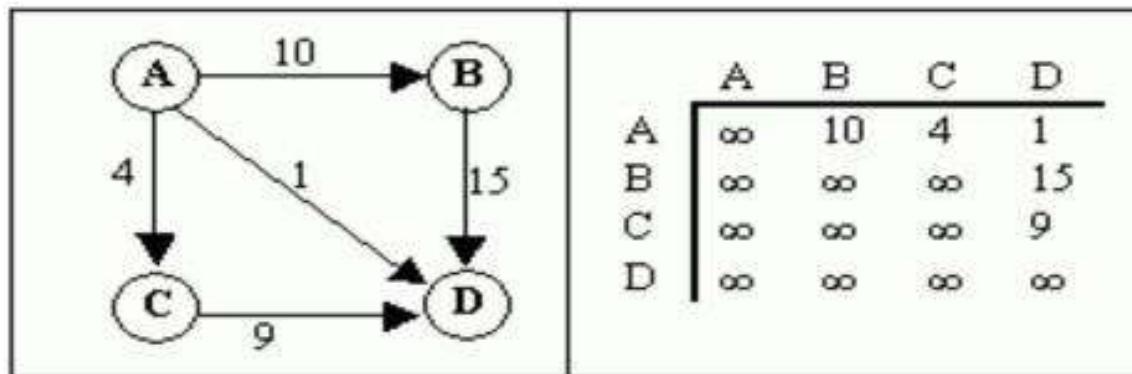
The presence or absence of an edge,  $(v, w)$  is indicated by the entry in row  $v$ , column  $w$  of the matrix.

For an unweighted graph, boolean values could be used.

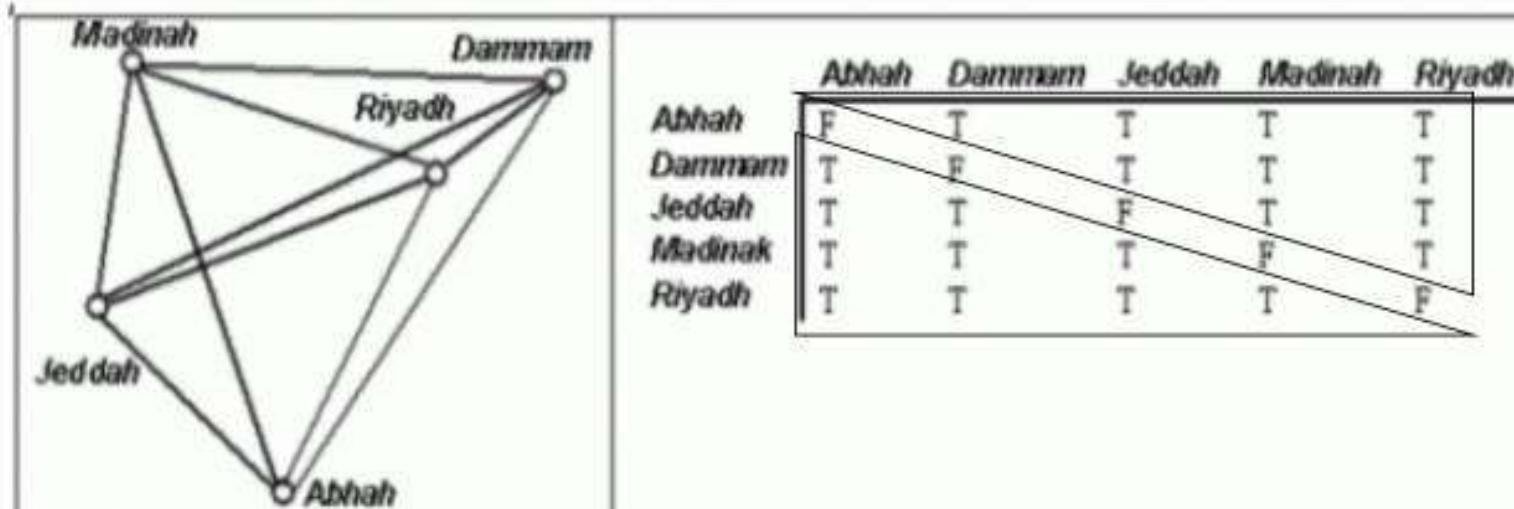


# Adjacency Matrix Representation

For a weighted graph, the actual weights are used.



For undirected graph, the adjacency matrix is always symmetric.



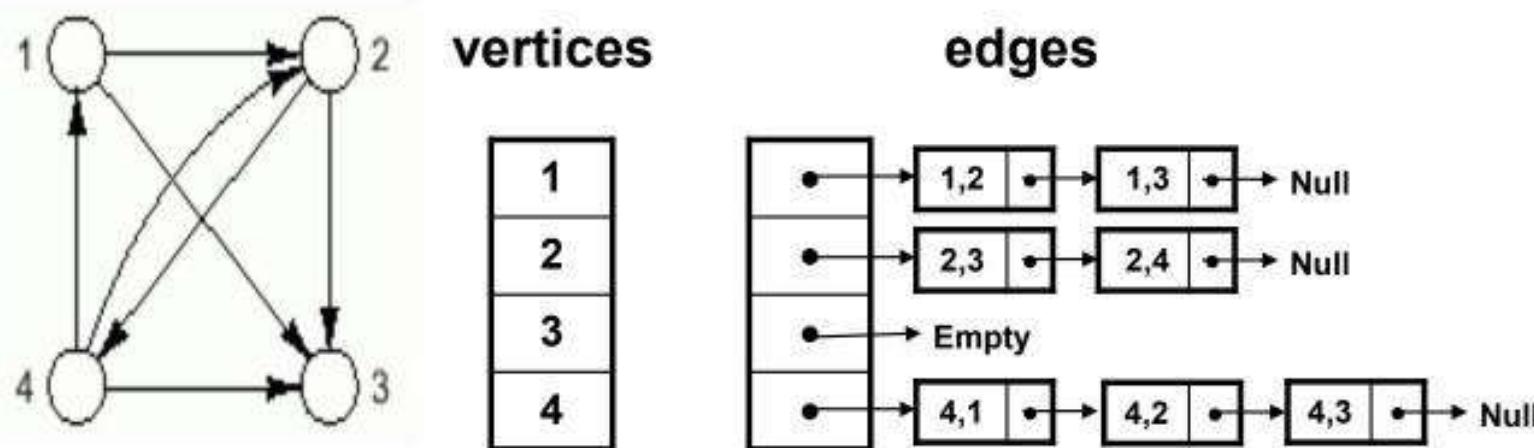
# Adjacency List Representation

This involves representing the set of vertices adjacent to each vertex as a list. Thus, generating a set of lists.

This can be implemented in different ways.

Vertices as a one dimensional array

Edges as an array of linked list (the emanating edges of vertex 1 will be in the list of the first element, and so on, ...)



# Graph Traversals

- Introduction
- Breadth-First Traversal.
- Depth-First Traversals.
- Some traversal applications:
- Review Questions.



# Graph Traversals

- Some algorithms require that every vertex of a graph be visited exactly once.
- The order in which the vertices are visited may be important, and may depend upon the particular algorithm.
- The two common traversals:
  - depth-first
  - breadth-first

# Graph Traversals:

## Depth First Search Traversal

- We follow a path through the graph until we reach a dead end.
- We then back up until we reach a node with an edge to an unvisited node.
- We take this edge and again follow it until we reach a dead end.
- This process continues until we back up to the starting node and it has no edges to unvisited nodes.

# Depth First Search Traversal Example

- Consider the following graph:

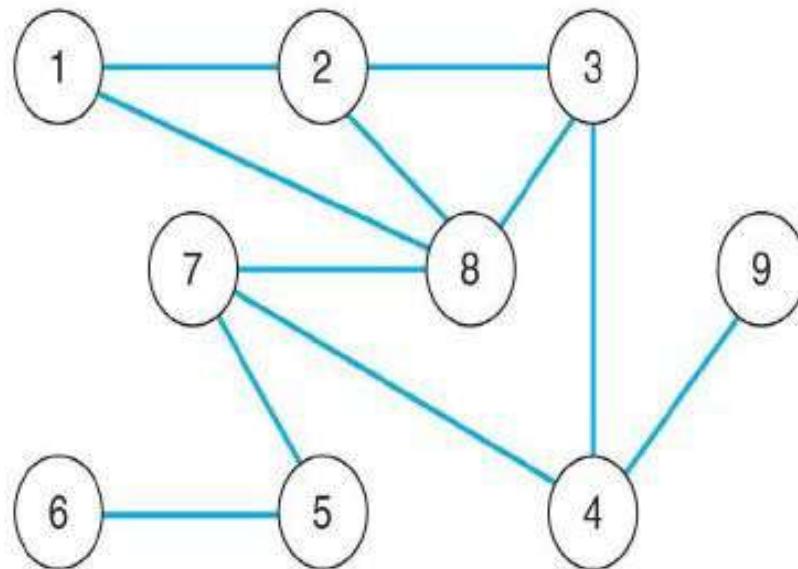


FIGURE 8.4

A graph

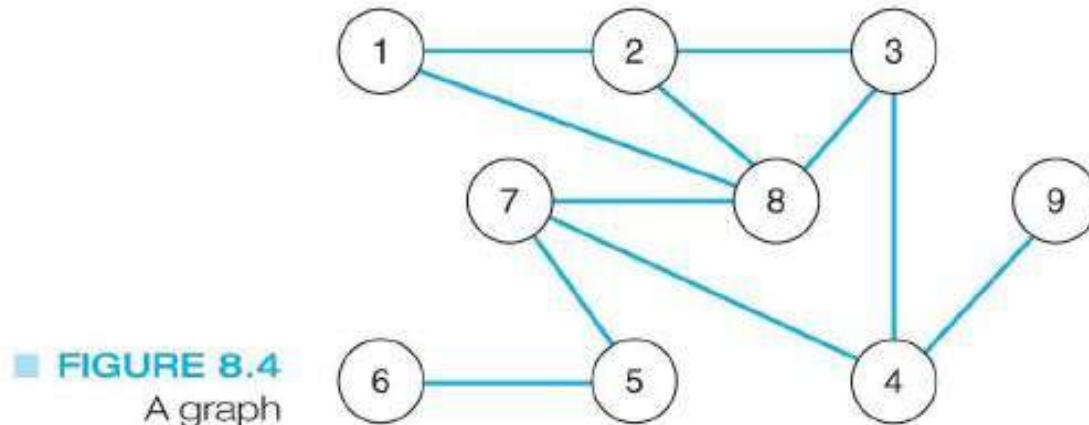
- The order of the depth-first traversal of this graph starting at node 1 would be:  
1, 2, 3, 4, 7, 5, 6, 8, 9

# Breadth First Search Traversal

- From the starting node, we follow all paths of length one.
- Then we follow paths of length two that go to unvisited nodes.
- We continue increasing the length of the paths until there are no unvisited nodes along any of the paths.

# Breadth First Search Traversal Example

- Consider the following graph:



- The order of the breadth-first traversal of this graph starting at node 1 would be: 1, 2, 8, 3, 7, 4, 5, 9, 6

# Introduction

- To traverse a graph is to systematically visit and process each node in the graph exactly once.
- There are two common graph traversal algorithms that are applicable to both directed and undirected graphs :
  - BreadthFirst Traversal (BFS)
  - DepthFirst Traversal (DFS)

## Introduction (Cont'd)

The BFS and DFS traversal of a graph  $G$  is not unique. A traversal depends both on the starting vertex, and on the order of traversing the adjacent vertices of each node.

General traversal algorithms [dfsAllVertices and bfsAllVertices] that will visit each vertex of a graph  $G$  in those algorithms that require each vertex to be visited.

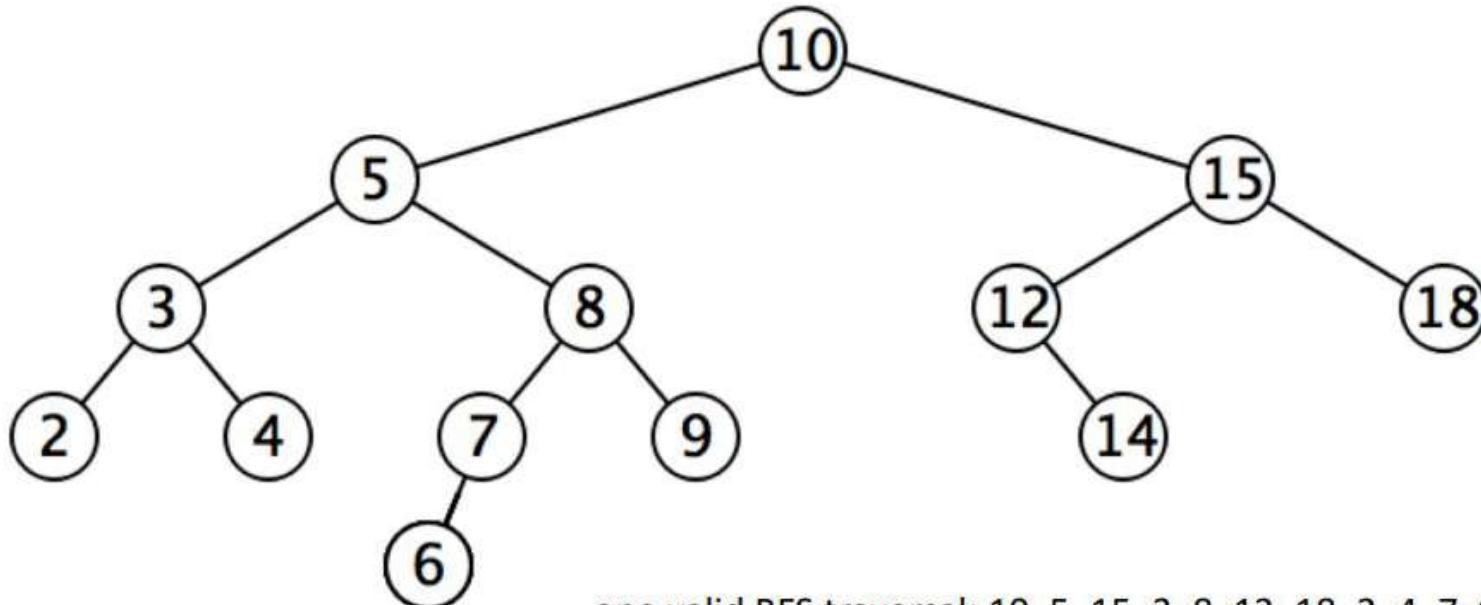
- **BFS:** Tries to explore all the neighbors it can reach from the current node. It will use a **queue** data structure.
- **DFS:** Tries to reach the farthest node from the current node and come back (backtrack) to the current node to explore its other neighbors. This will use a **stack** data structure.

# Graph traversals: BFS

**Breadth** First Search - a traversal on graphs (or on trees since those are also graphs) where you traverse level by level. So in this one we'll get to all the shallow nodes before any "deep nodes".

Intuitive ways to think about BFS:

- opposite way of traversing compared to DFS
- a sound wave spreading from a starting point, going outwards in all directions possible.
- mold on a piece of food spreading outwards so that it eventually covers the whole surface



one valid BFS traversal: 10, 5, 15, 3, 8, 12, 18, 2, 4, 7, 9, 14, 6

## Breadth-First Traversal Algorithm

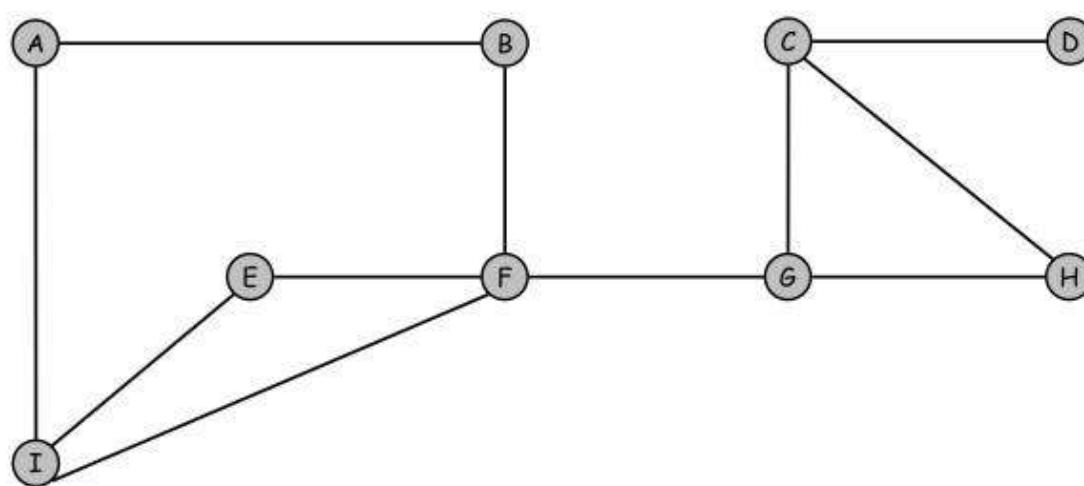
In this method, After visiting a vertex  $v$ , we must visit all its adjacent vertices  $w_1, w_2, w_3, \dots$ , before going down to the next level to visit vertices adjacent to  $w_1$  etc.

The method can be implemented using a queue.

```
enqueue the starting vertex
while(queue is not empty){
    dequeue a vertex v from the queue;
    visit v.
    enqueue vertices adjacent to v that were never enqueued;
}
```

- A BFS traversal of a graph results in a breadth-first tree or in a forest of such trees.

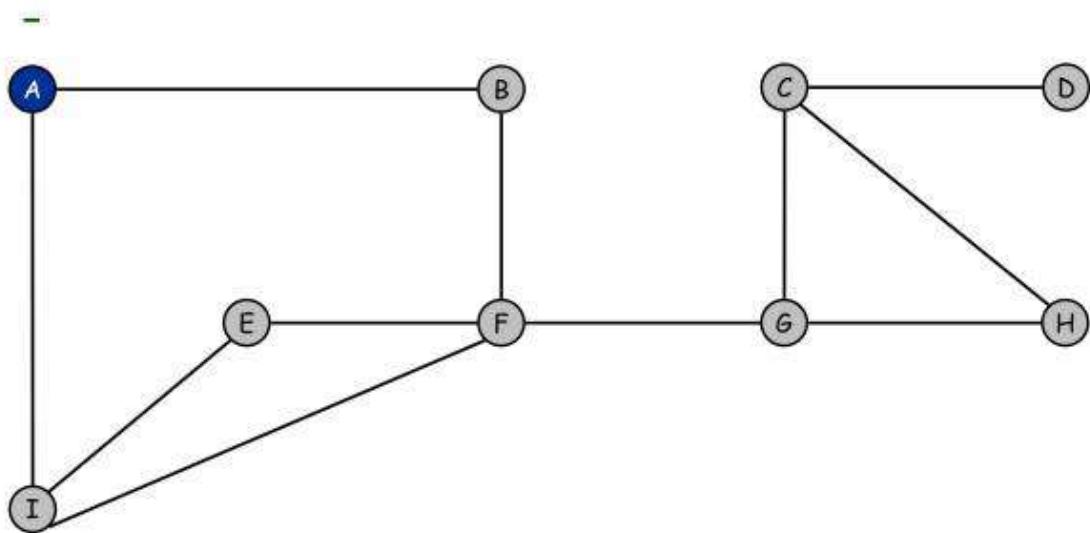
## Breadth First Search



front

FIFO Queue

## Breadth First Search

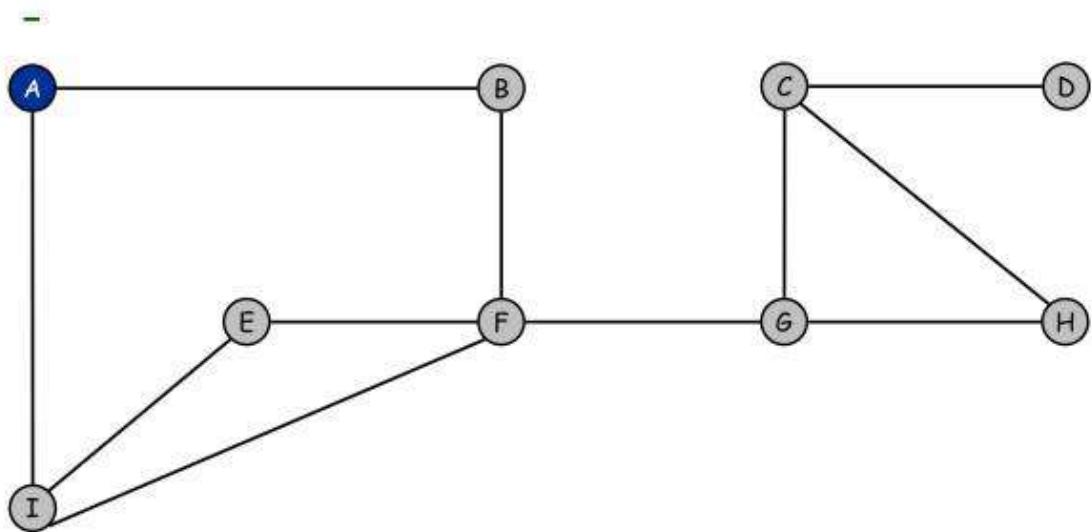


enqueue source node

front **A**

FIFO Queue

## Breadth First Search

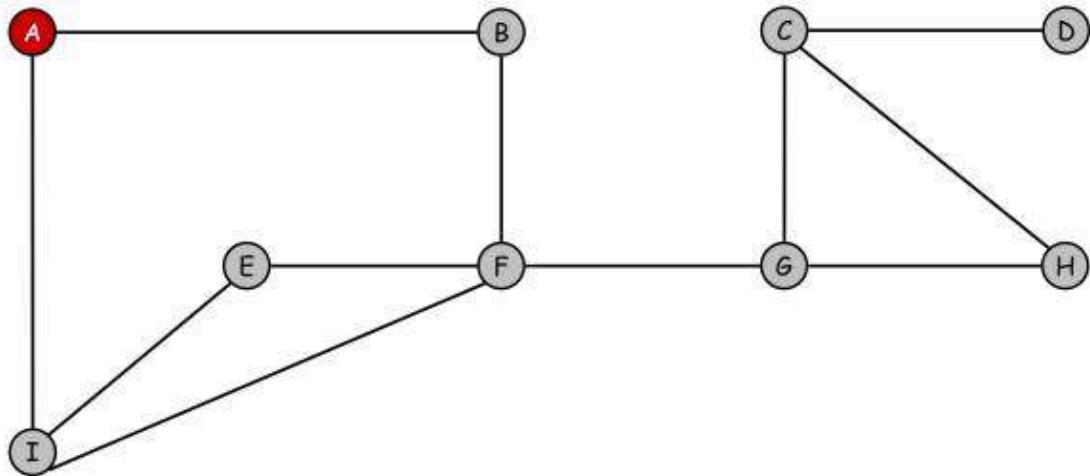


dequeue next vertex

front **A**

FIFO Queue

## Breadth First Search

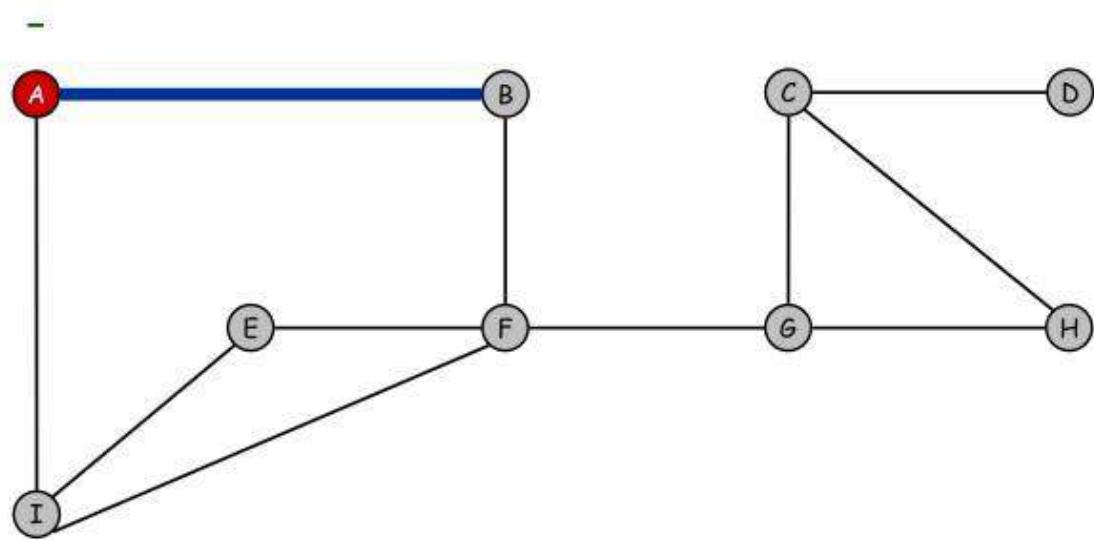


visit neighbors of A

front

FIFO Queue

## Breadth First Search

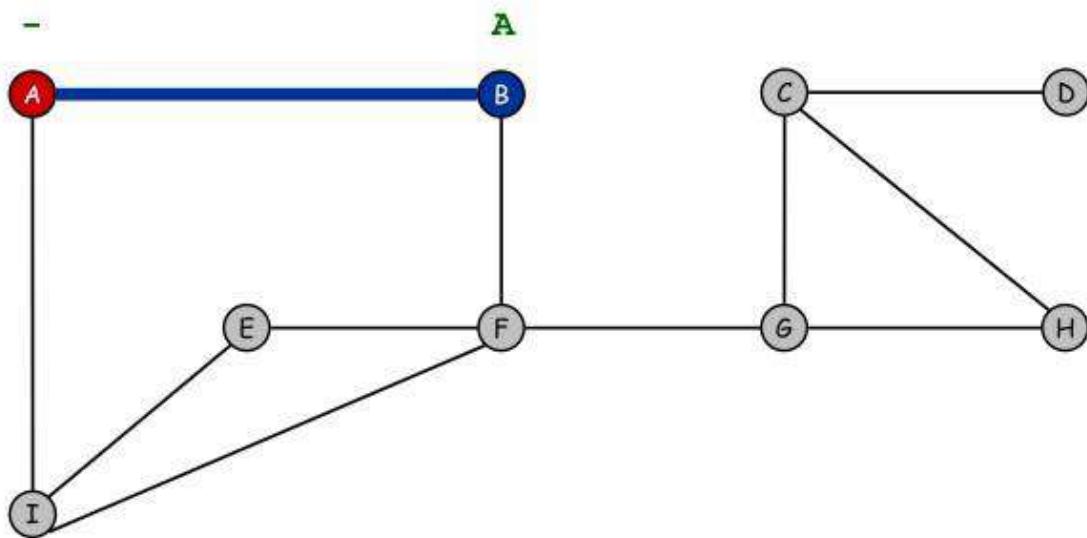


visit neighbors of A

front

FIFO Queue

## Breadth First Search



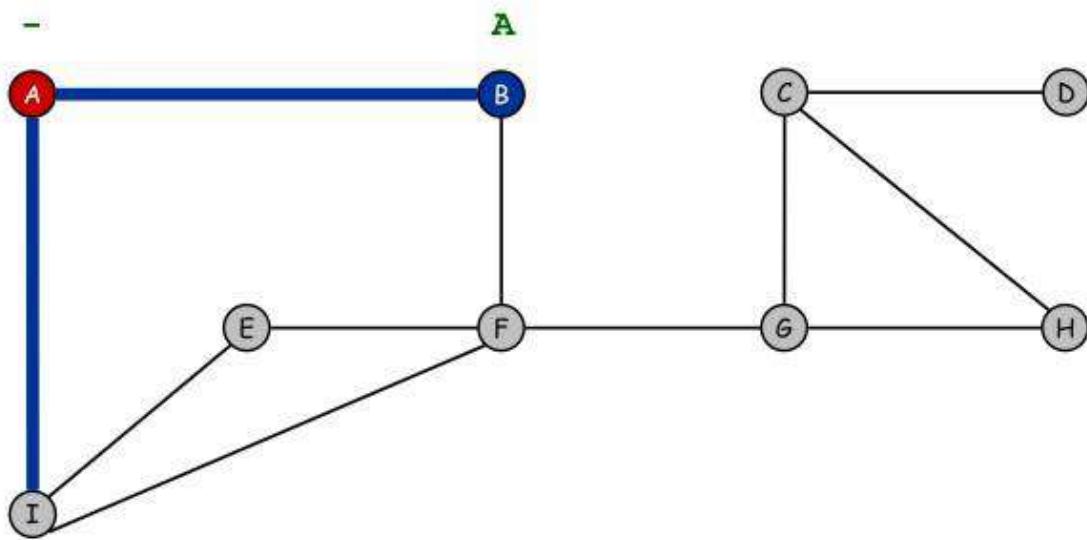
B discovered

front

B

FIFO Queue

## Breadth First Search

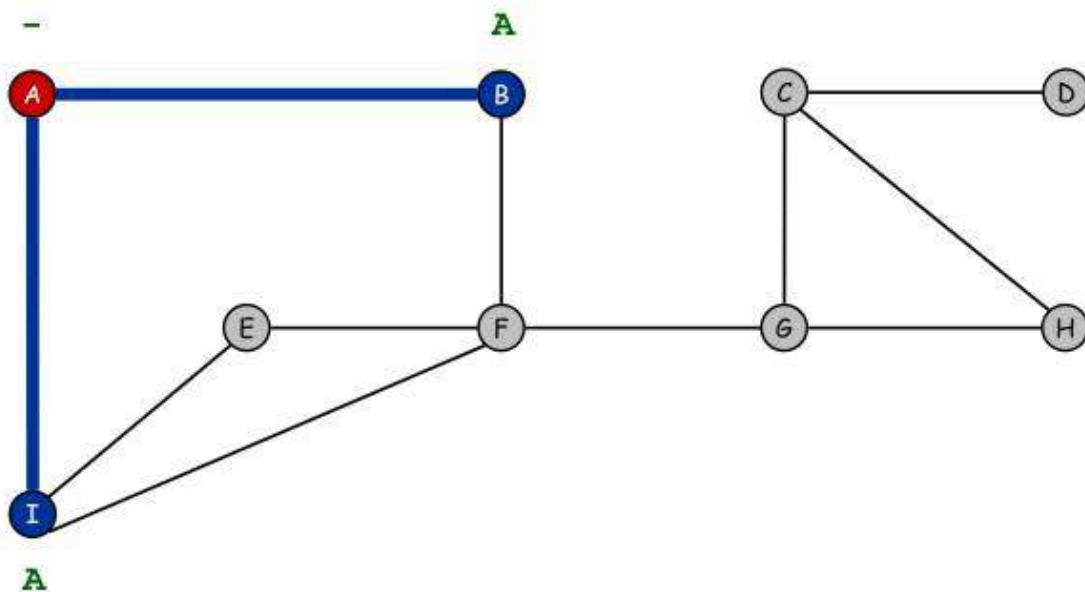


visit neighbors of A

front B

## FIFO Queue

## Breadth First Search

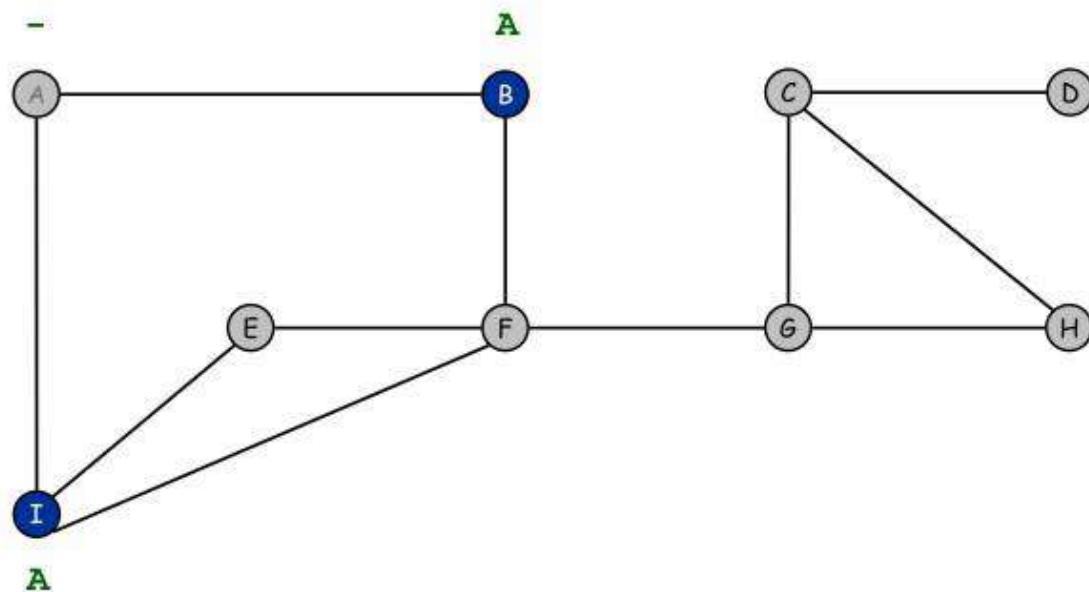


I discovered

front B I

FIFO Queue

## Breadth First Search

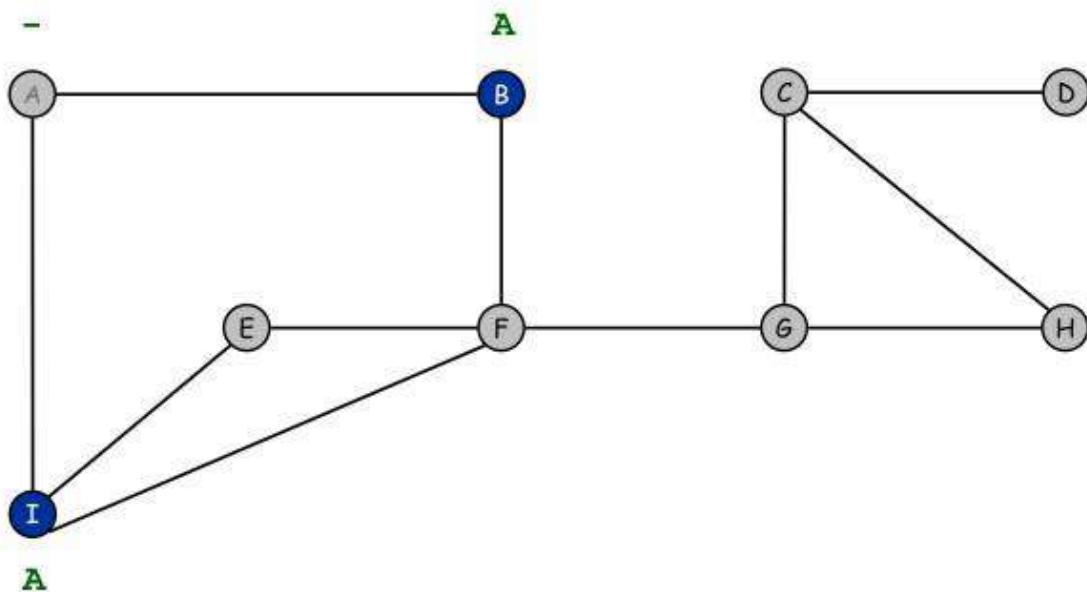


finished with A

front B I

FIFO Queue

## Breadth First Search

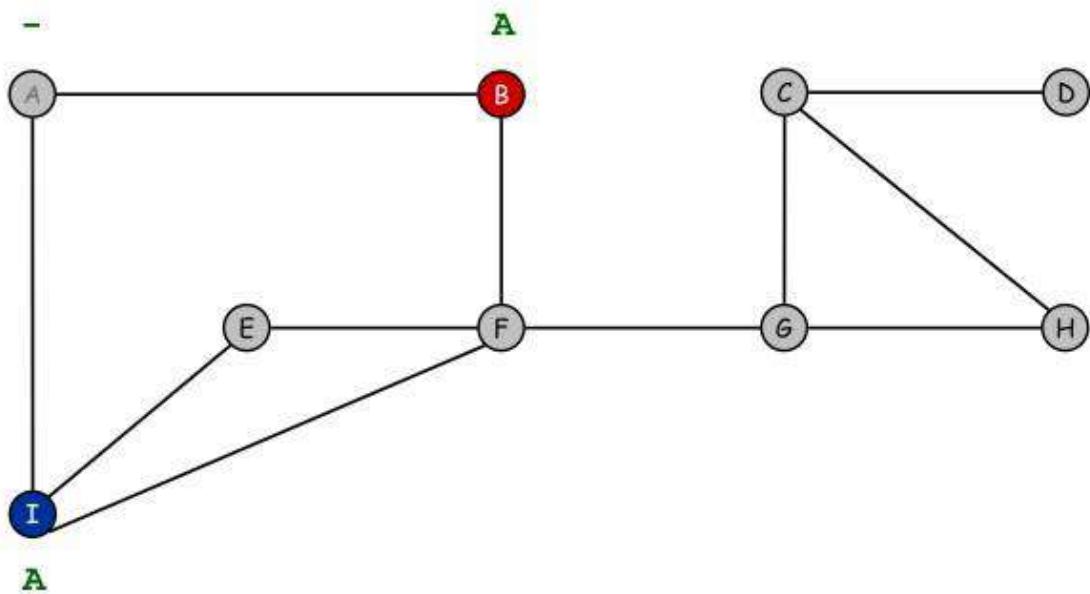


dequeue next vertex

front **B I**

FIFO Queue

## Breadth First Search

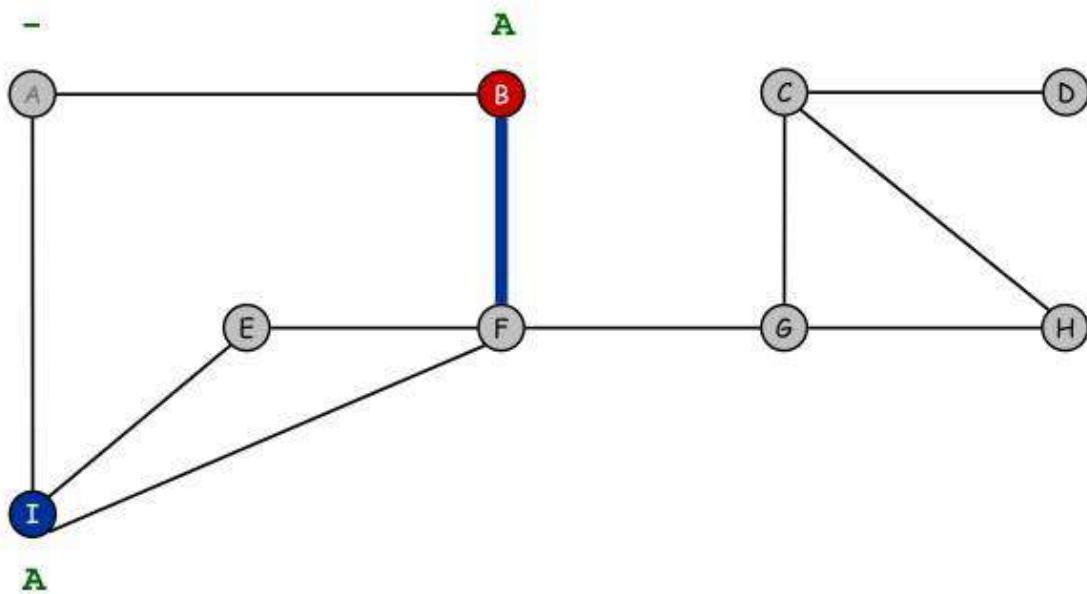


visit neighbors of B

front I

FIFO Queue

## Breadth First Search

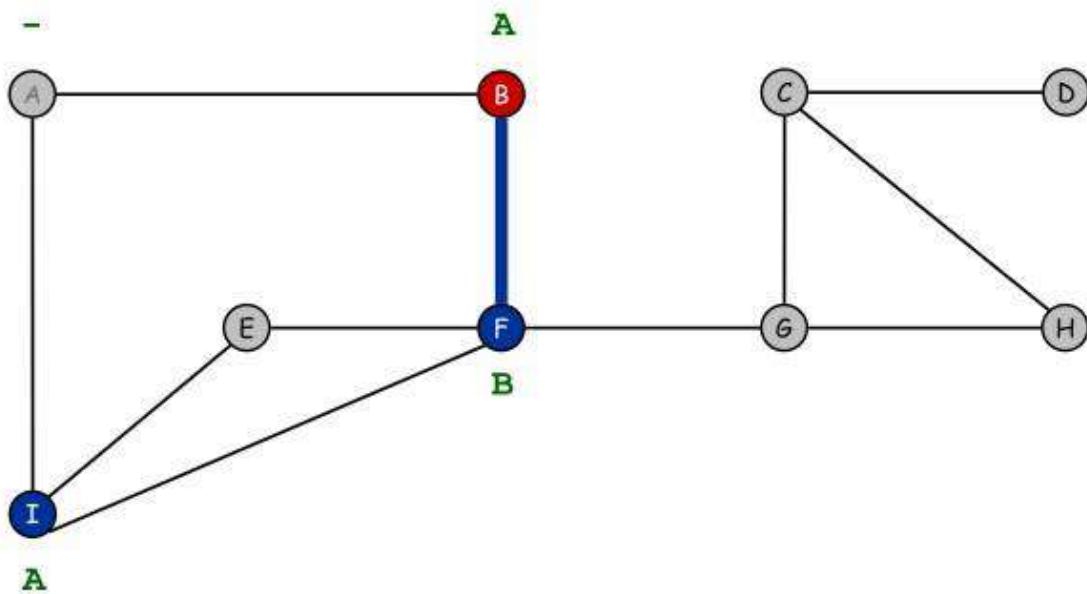


visit neighbors of B

front I

FIFO Queue

## Breadth First Search

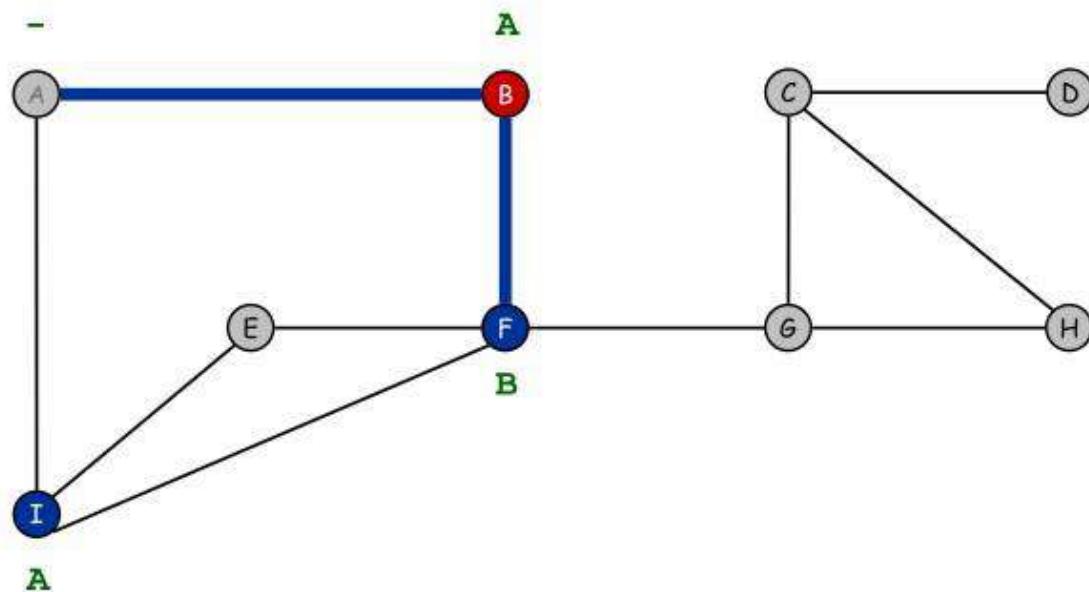


F discovered

front I F

FIFO Queue

## Breadth First Search

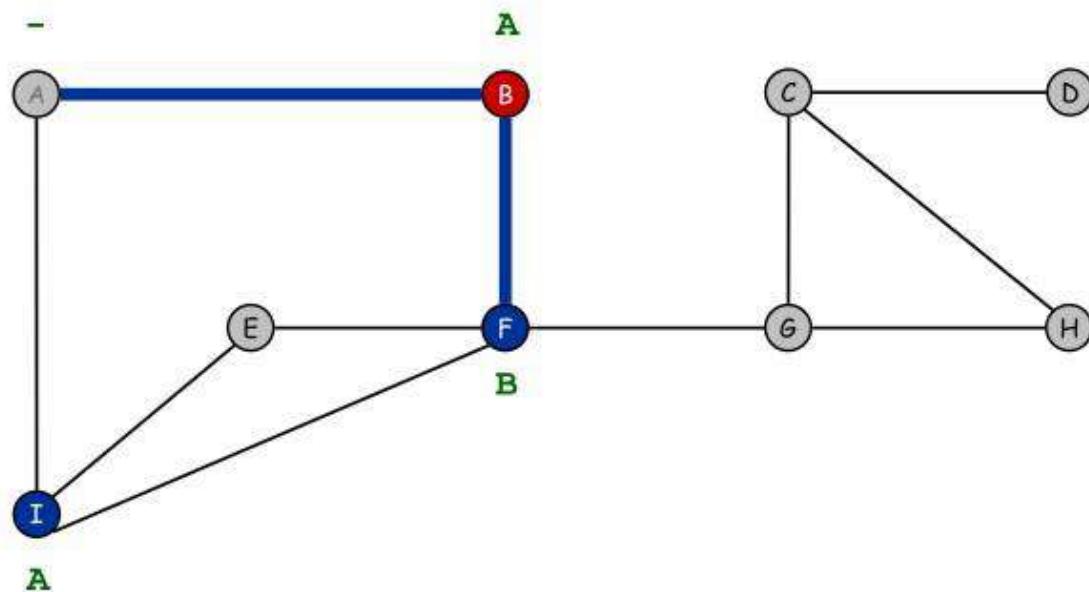


visit neighbors of B

front I F

FIFO Queue

## Breadth First Search



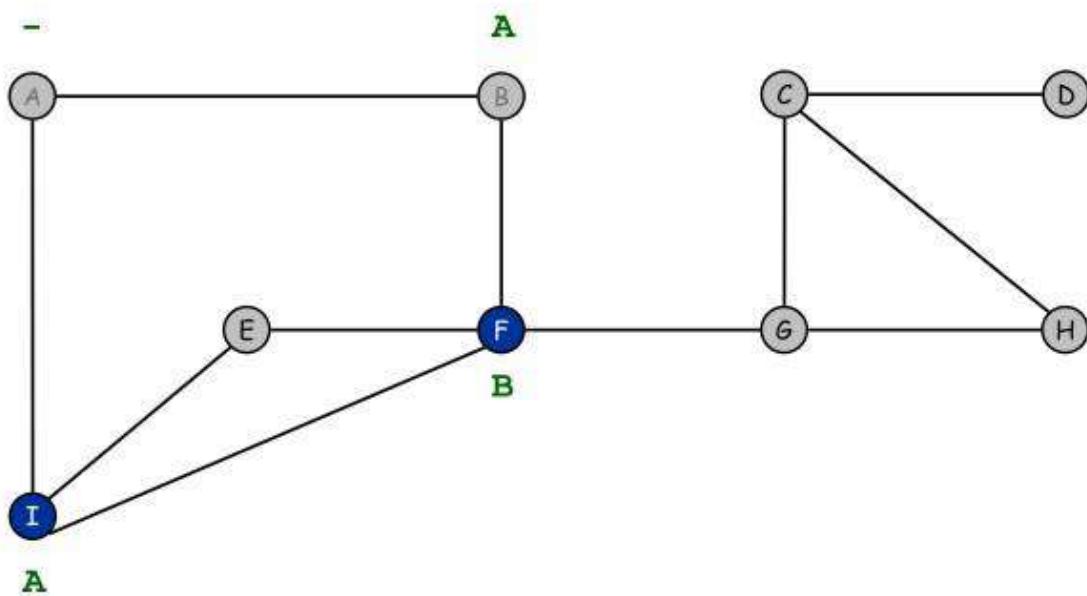
A already discovered

front

I F

FIFO Queue

## Breadth First Search

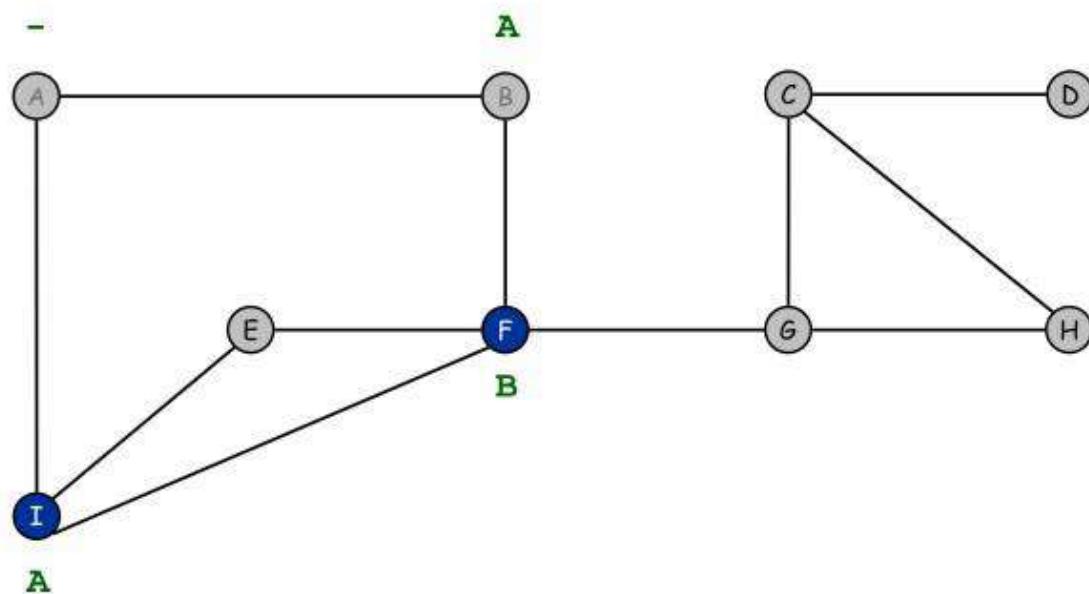


finished with B

front I F

## FIFO Queue

## Breadth First Search

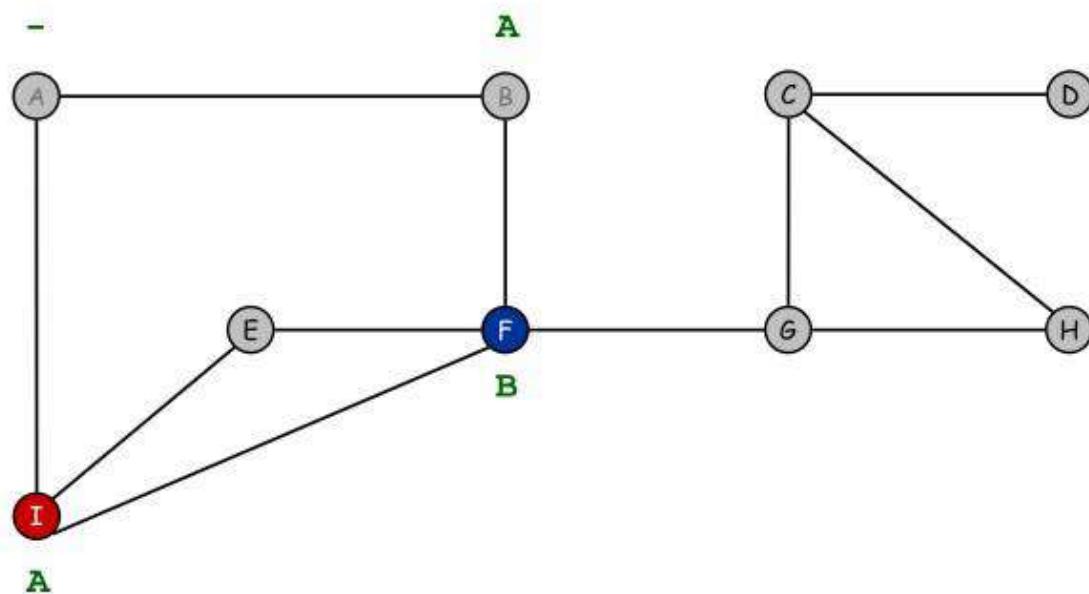


dequeue next vertex

front I F

FIFO Queue

## Breadth First Search

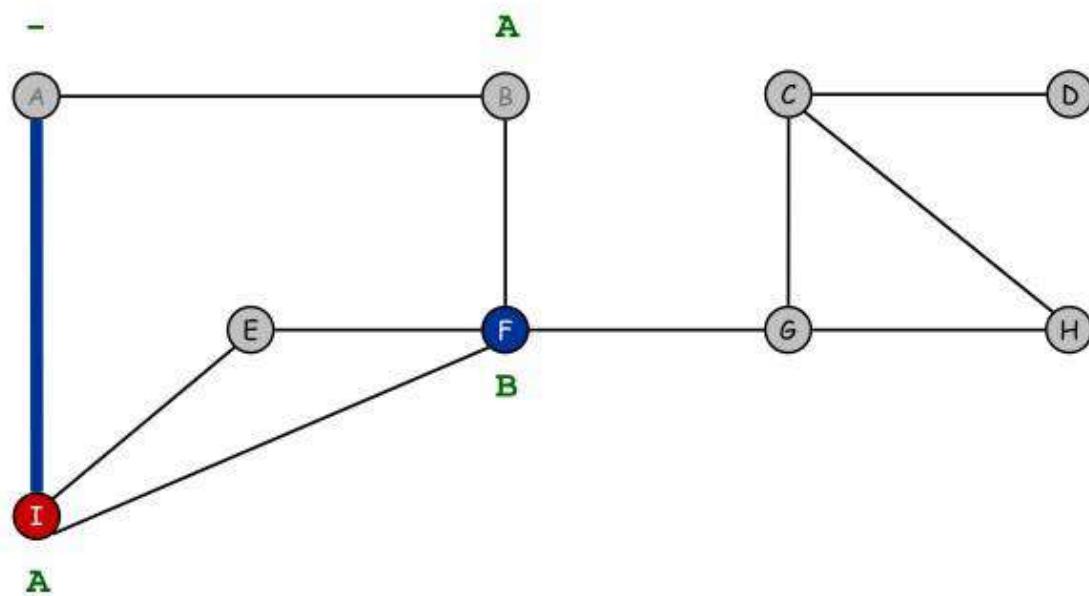


visit neighbors of I

front F

FIFO Queue

## Breadth First Search

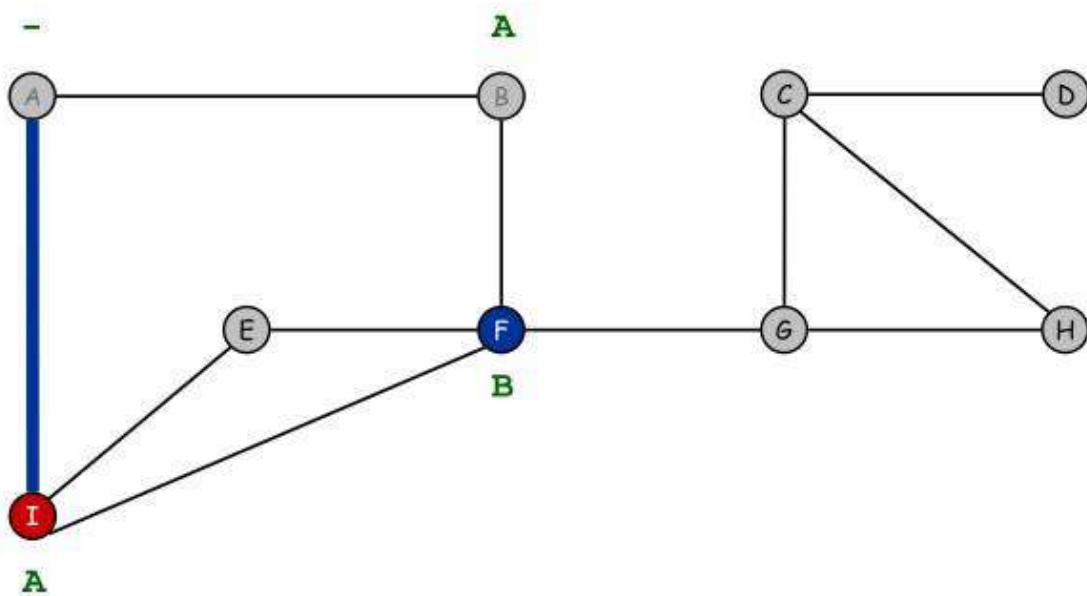


visit neighbors of I

front F

FIFO Queue

## Breadth First Search

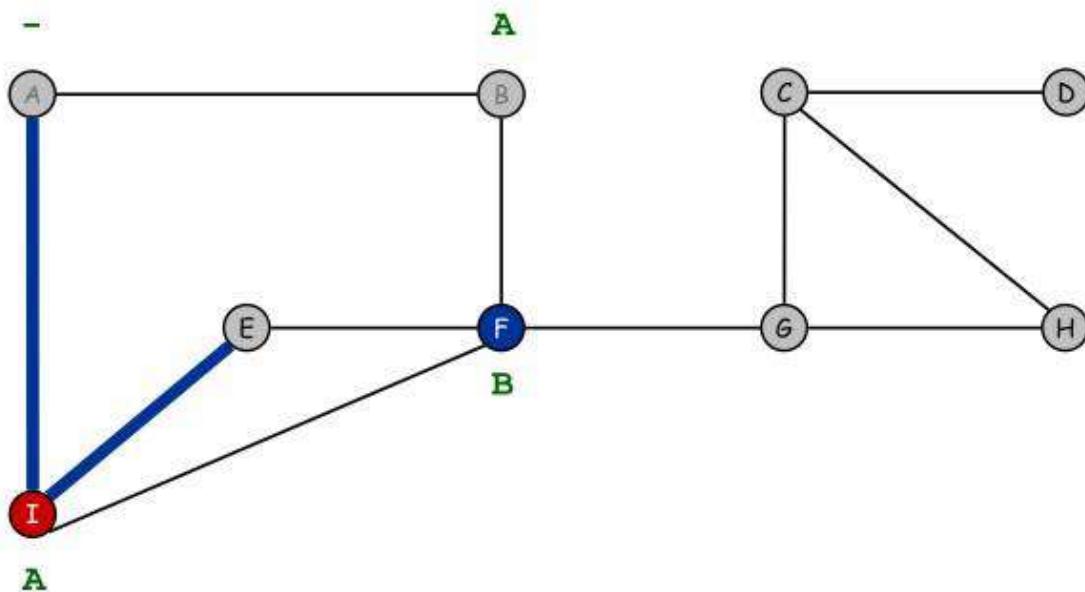


A already discovered

front F

## FIFO Queue

## Breadth First Search

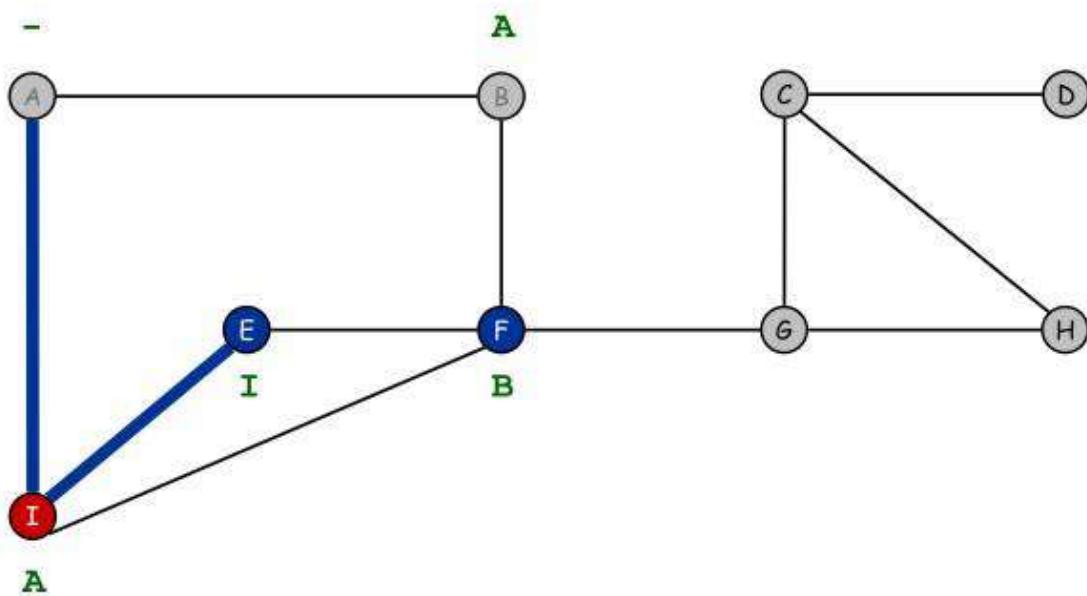


visit neighbors of I

front F

FIFO Queue

## Breadth First Search

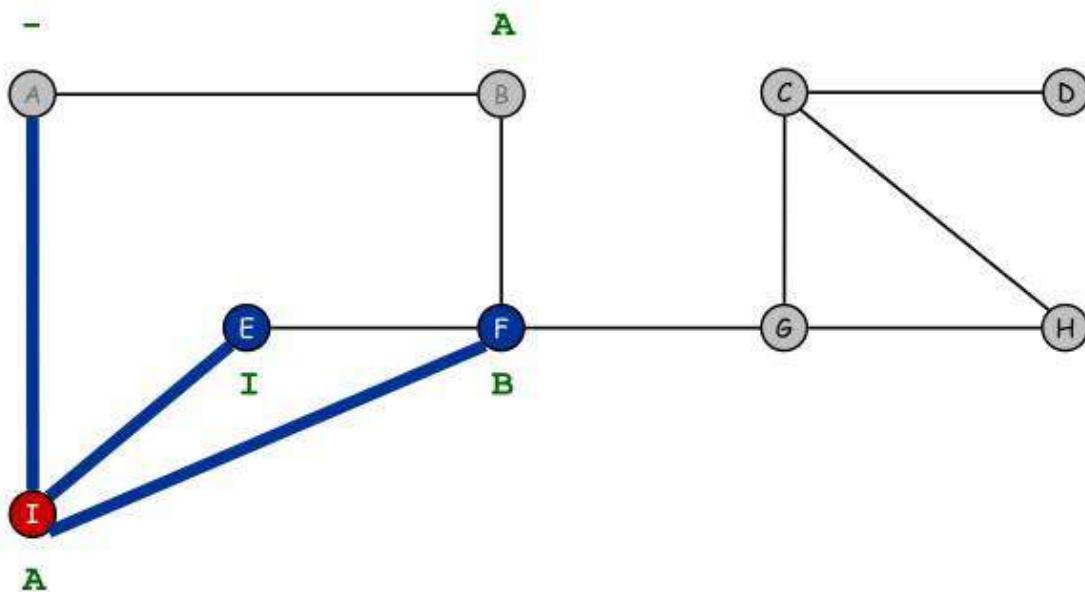


E discovered

front F E

## FIFO Queue

## Breadth First Search

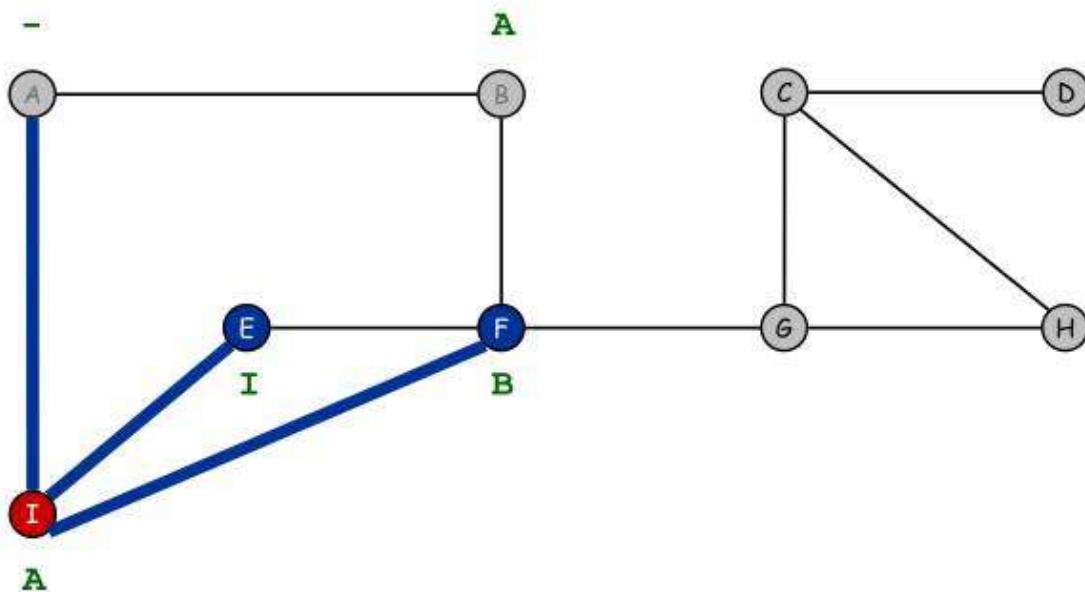


visit neighbors of I

front **F E**

FIFO Queue

## Breadth First Search

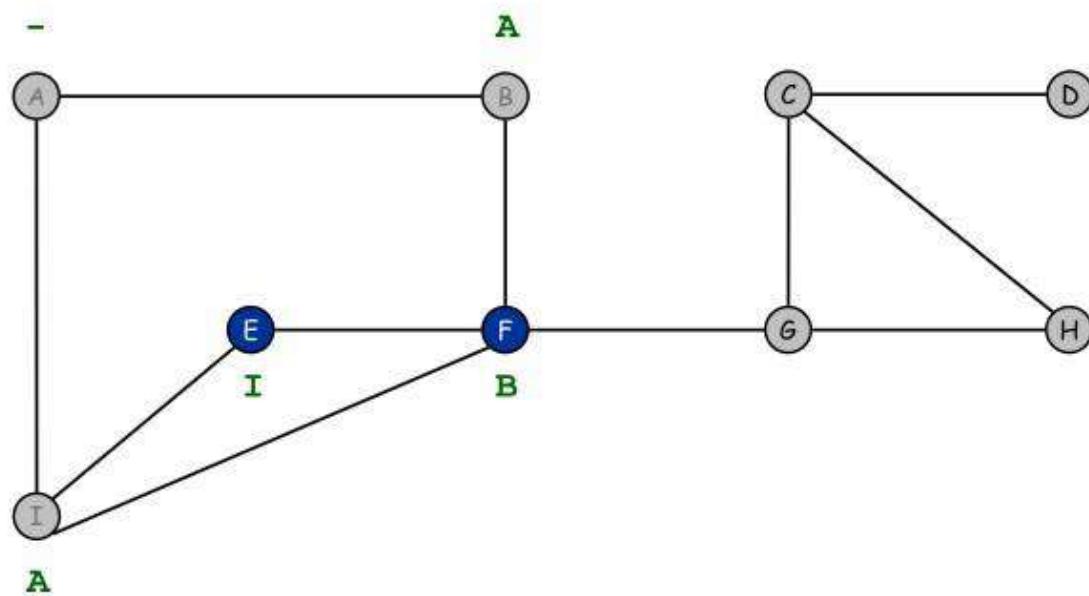


F already discovered

front **F E**

FIFO Queue

## Breadth First Search

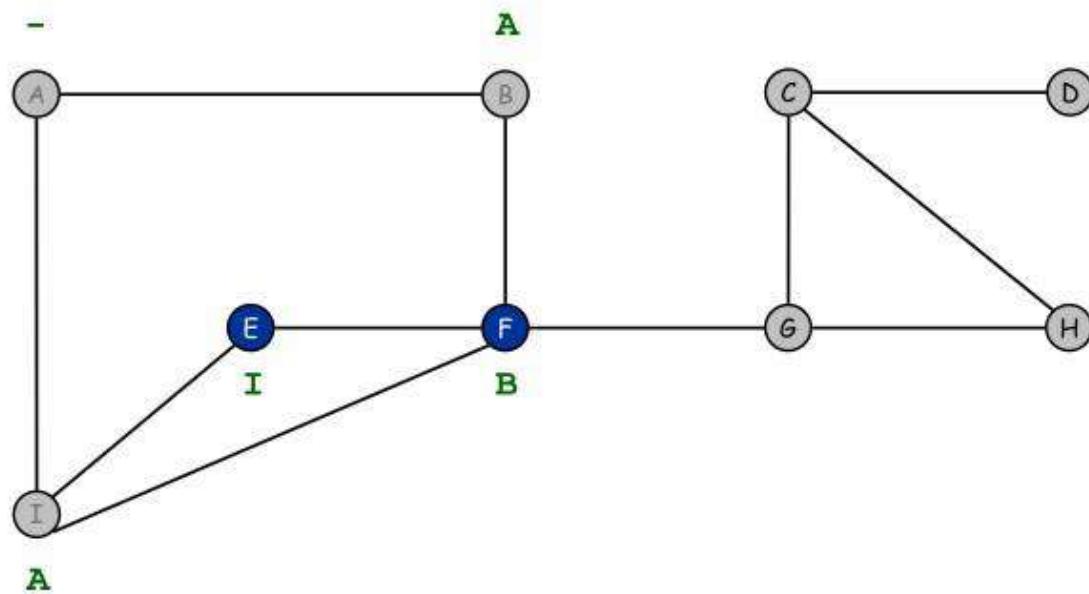


I finished

front **F E**

FIFO Queue

## Breadth First Search

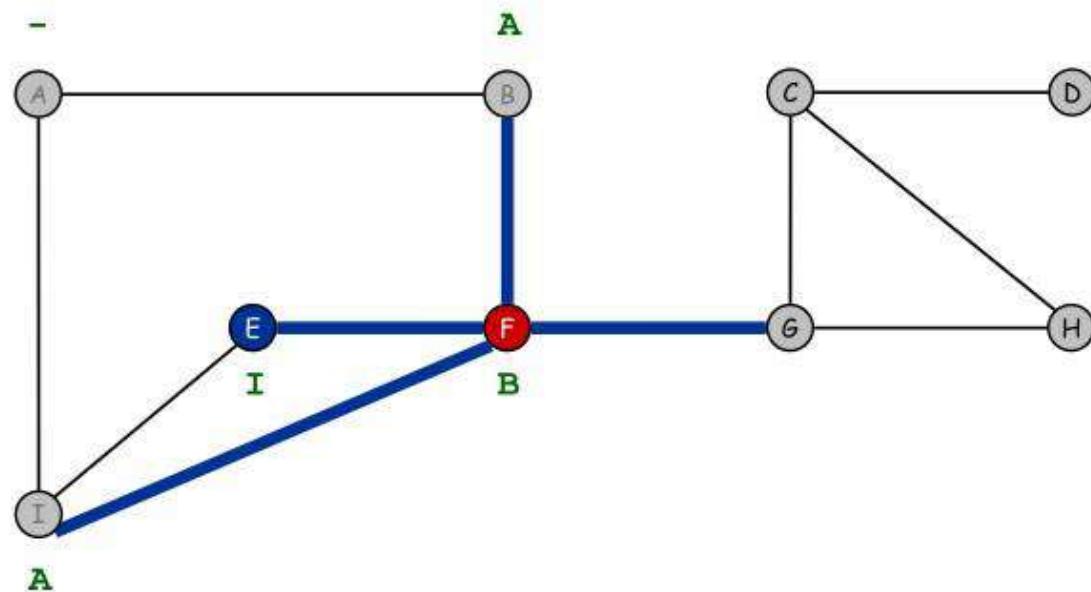


dequeue next vertex

front **F E**

FIFO Queue

## Breadth First Search

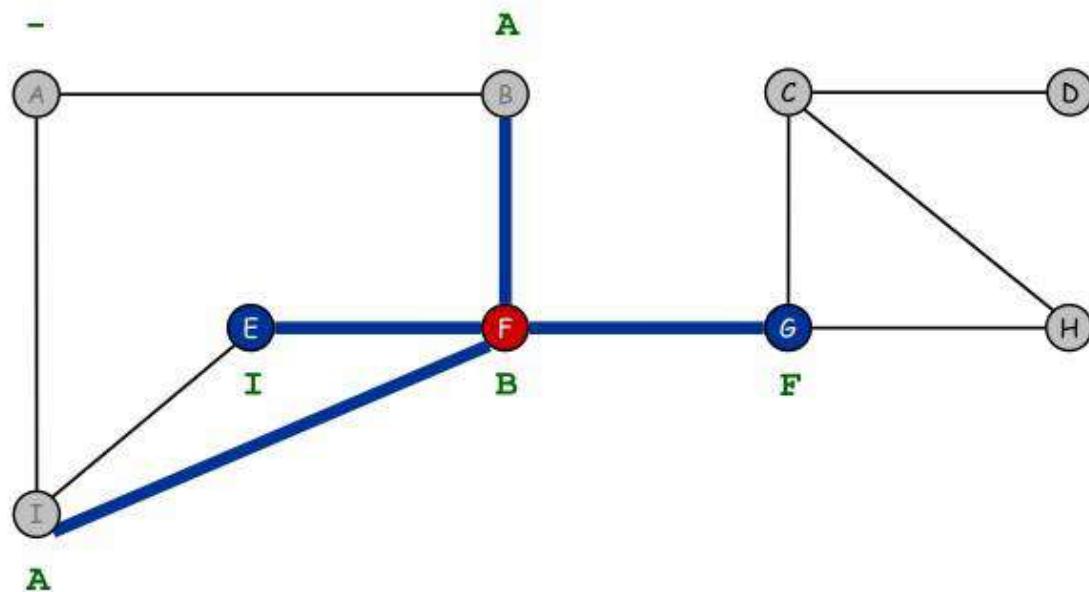


visit neighbors of F

front E

FIFO Queue

## Breadth First Search

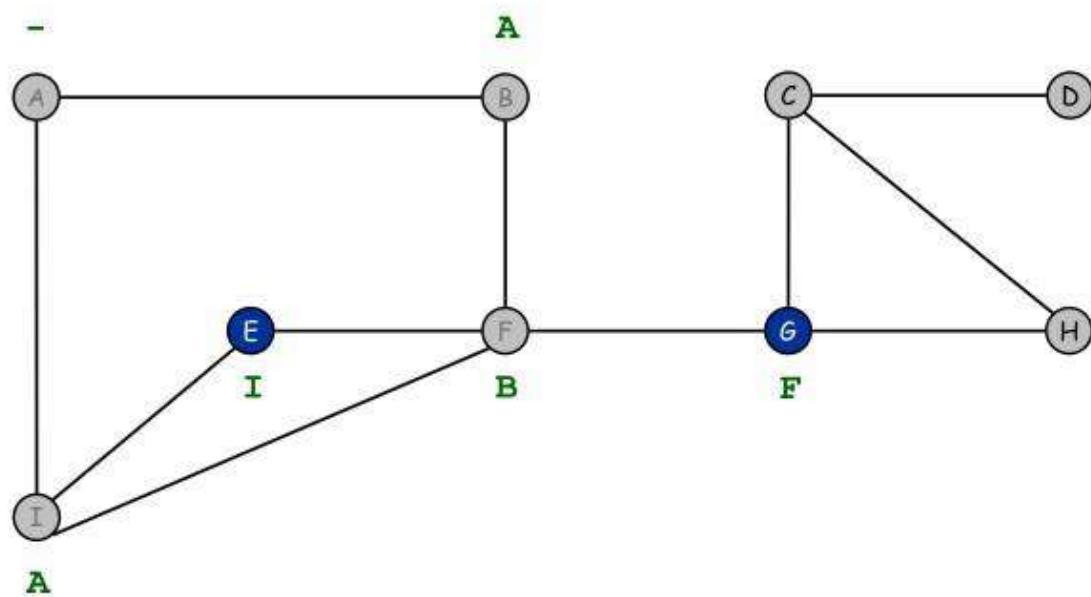


$G$  discovered

front E G

## FIFO Queue

## Breadth First Search

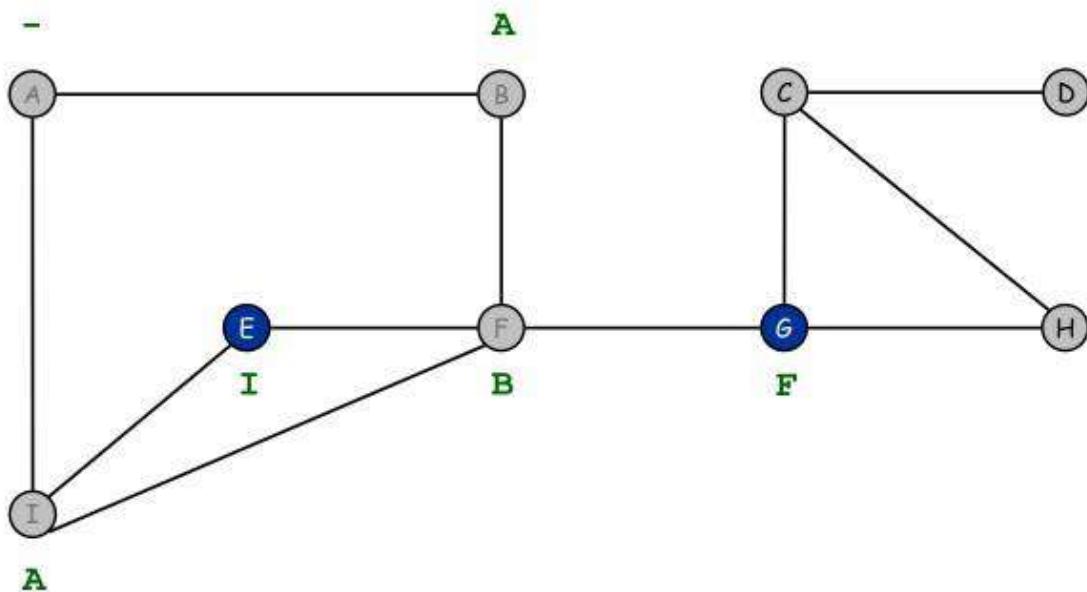


F finished

front E G

FIFO Queue

## Breadth First Search

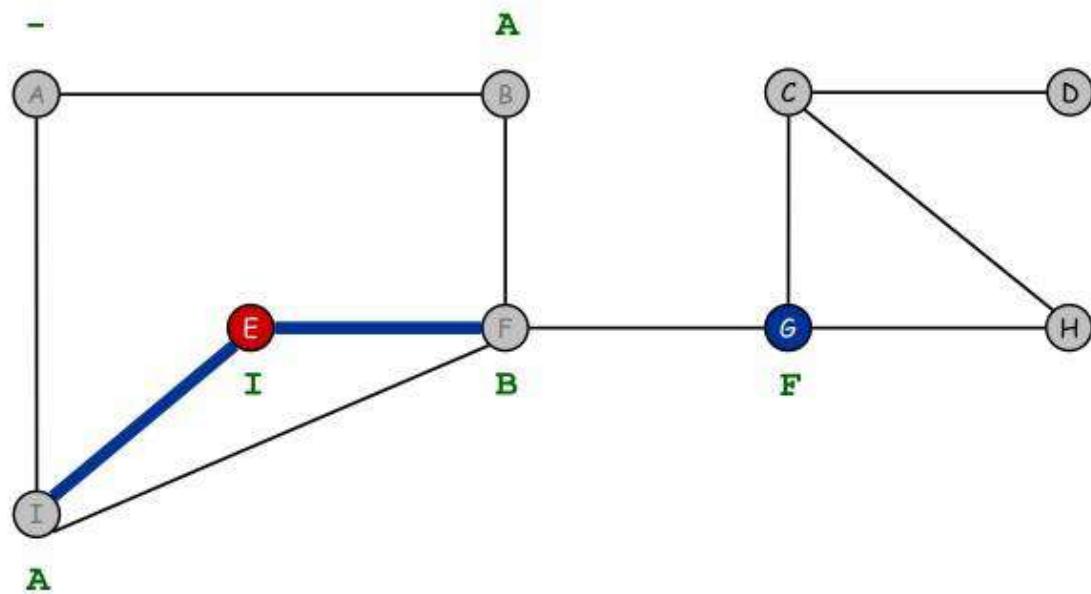


dequeue next vertex

front **E G**

FIFO Queue

## Breadth First Search

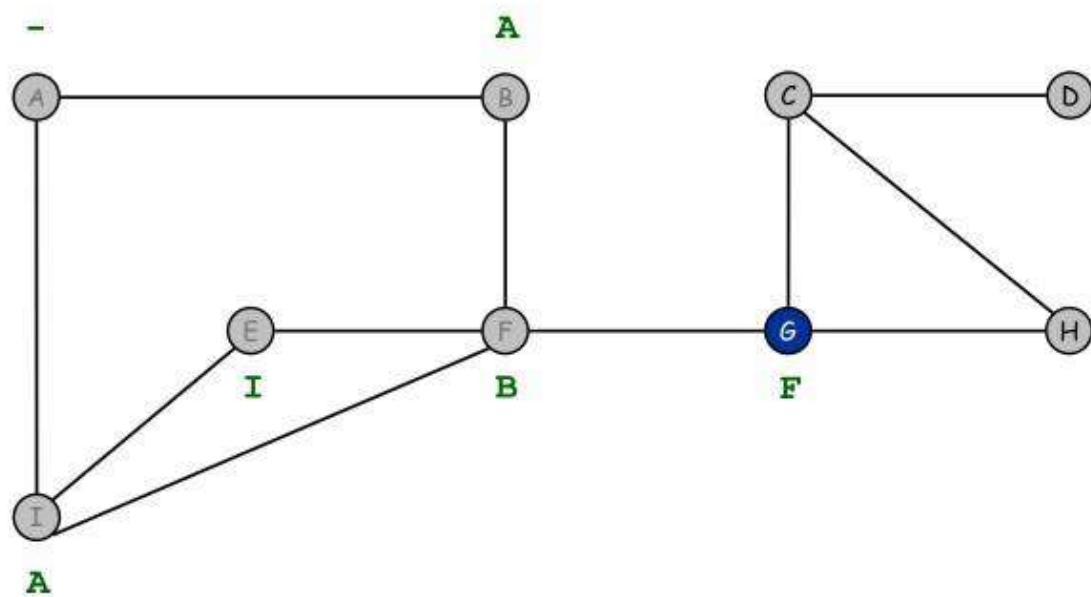


visit neighbors of E

front G

FIFO Queue

## Breadth First Search

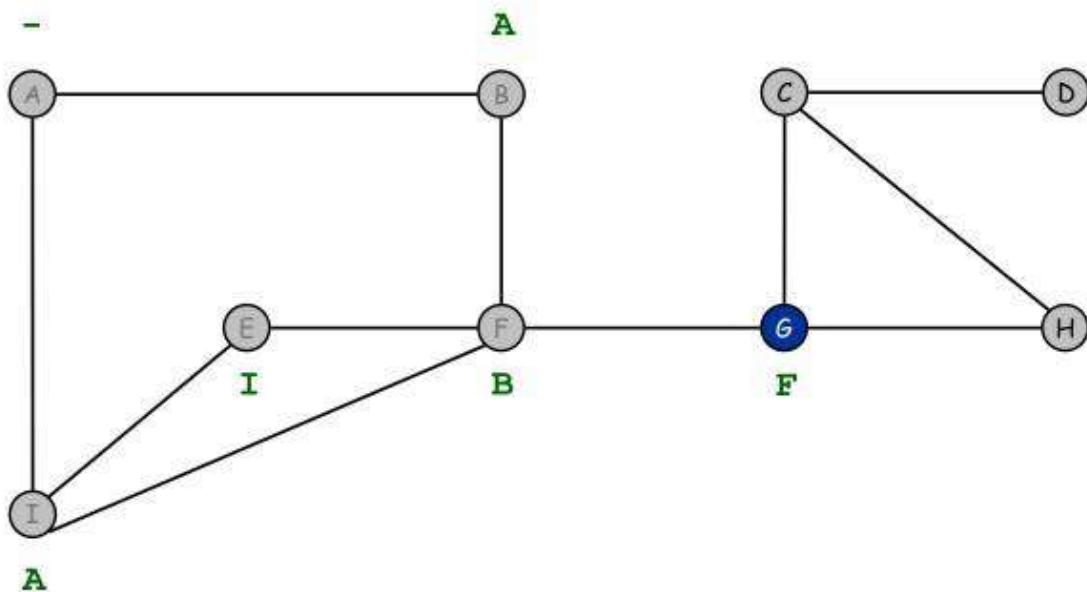


E finished

front G

FIFO Queue

## Breadth First Search

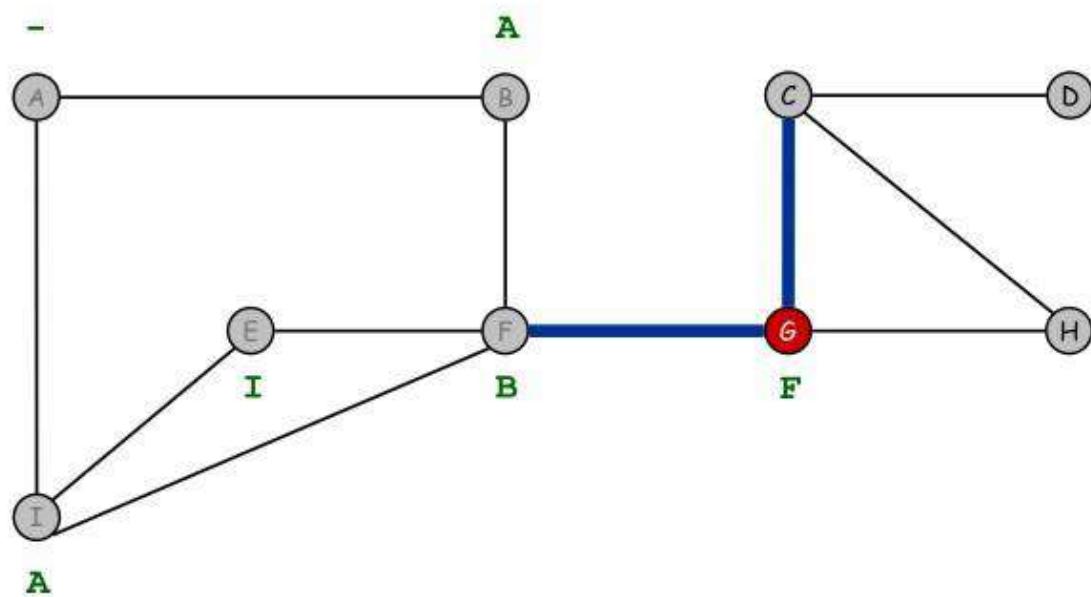


dequeue next vertex

front G

FIFO Queue

## Breadth First Search

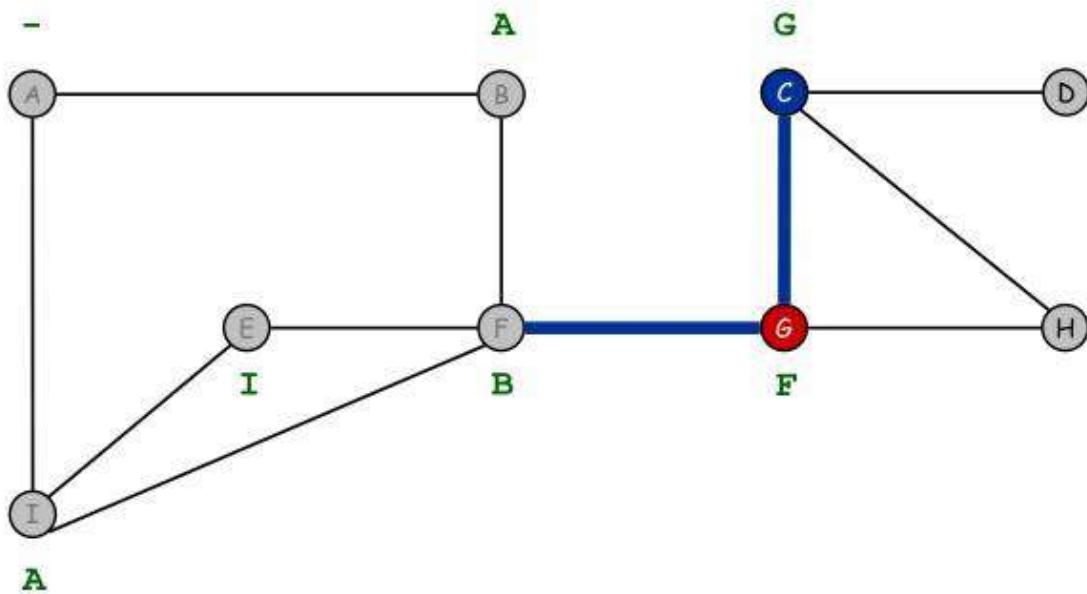


visit neighbors of *G*

front

FIFO Queue

## Breadth First Search

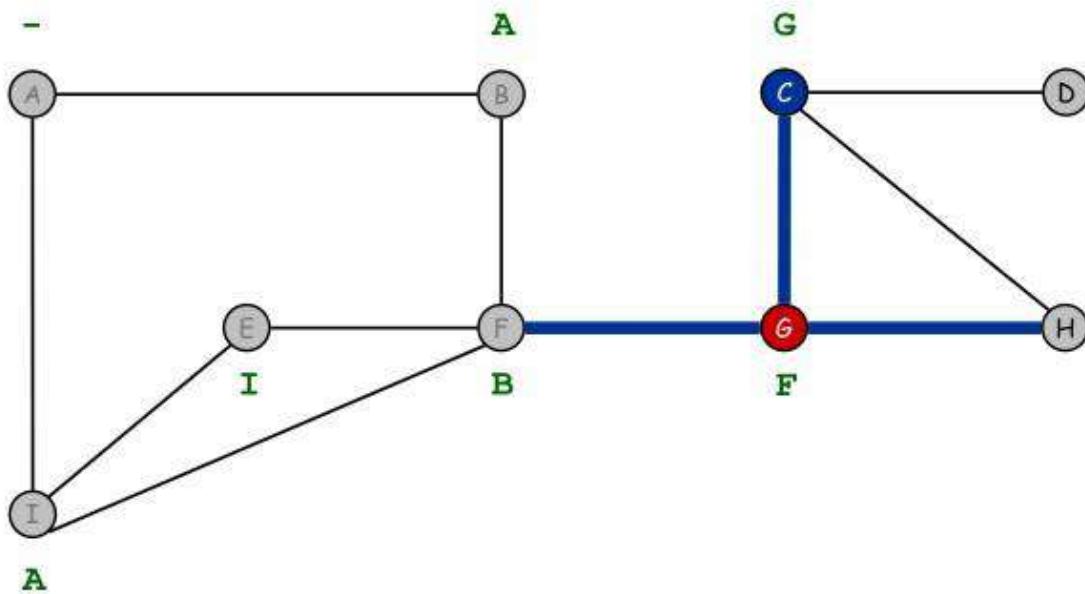


*C* discovered

front **C**

FIFO Queue

## Breadth First Search

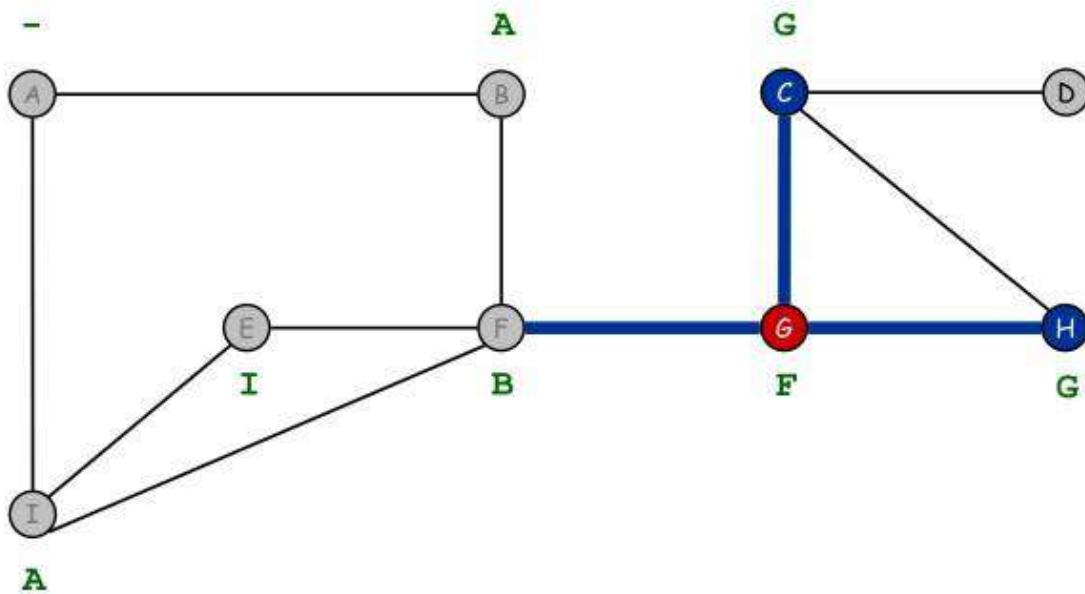


visit neighbors of G

front C

FIFO Queue

## Breadth First Search

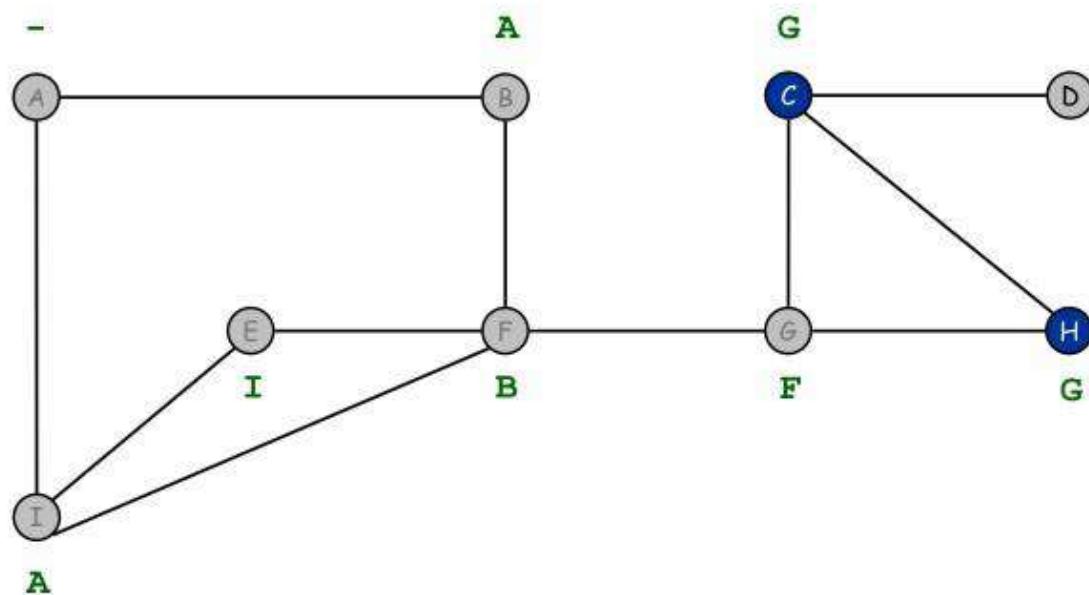


H discovered

front C H

FIFO Queue

## Breadth First Search

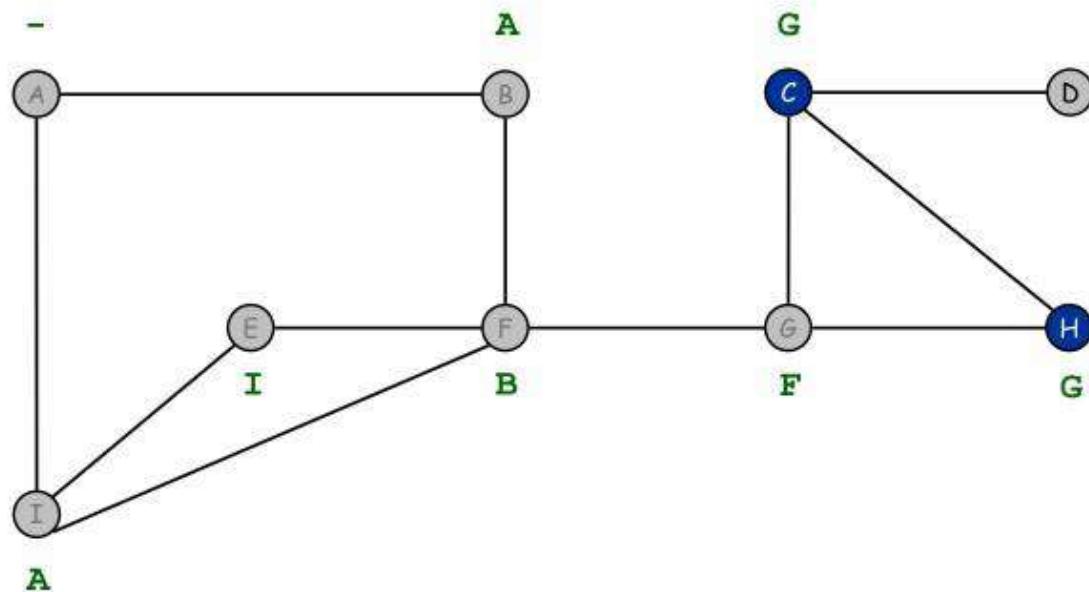


**G finished**

front **C H**

FIFO Queue

## Breadth First Search

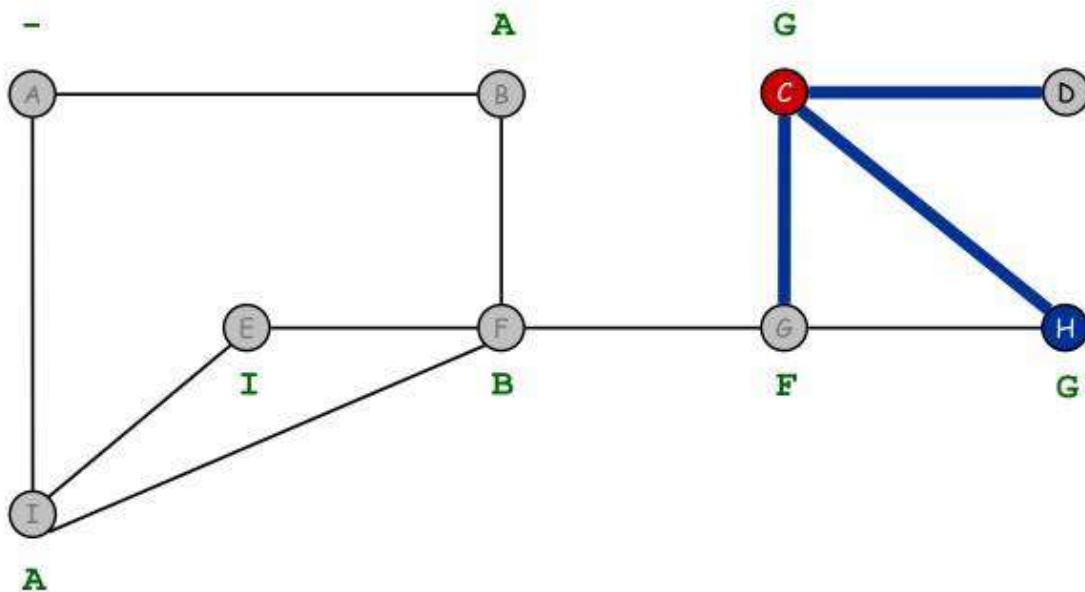


dequeue next vertex

front C H

FIFO Queue

## Breadth First Search

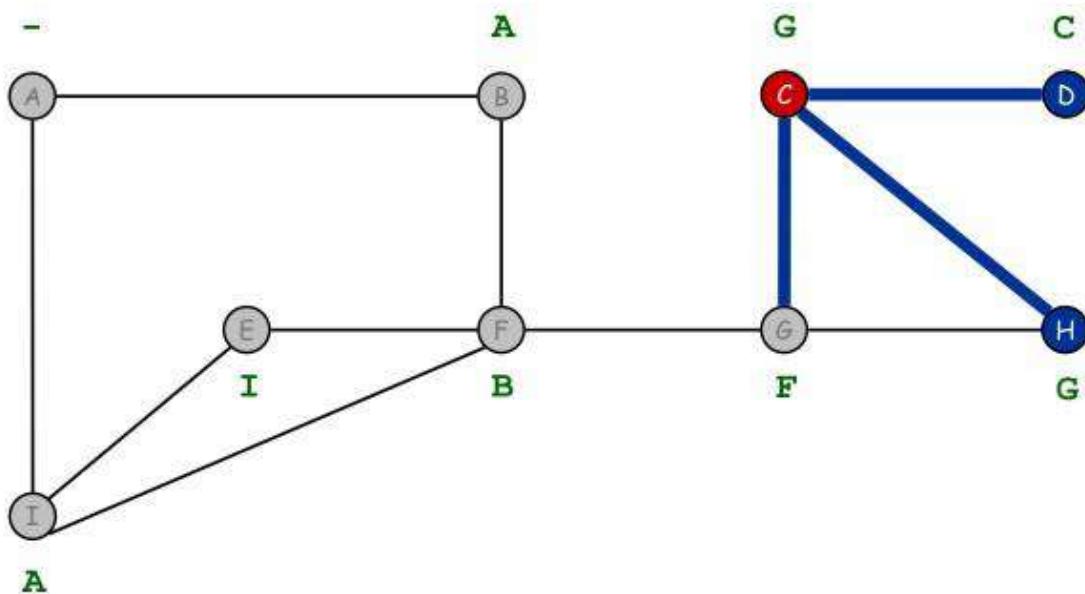


visit neighbors of **C**

front **H**

FIFO Queue

## Breadth First Search

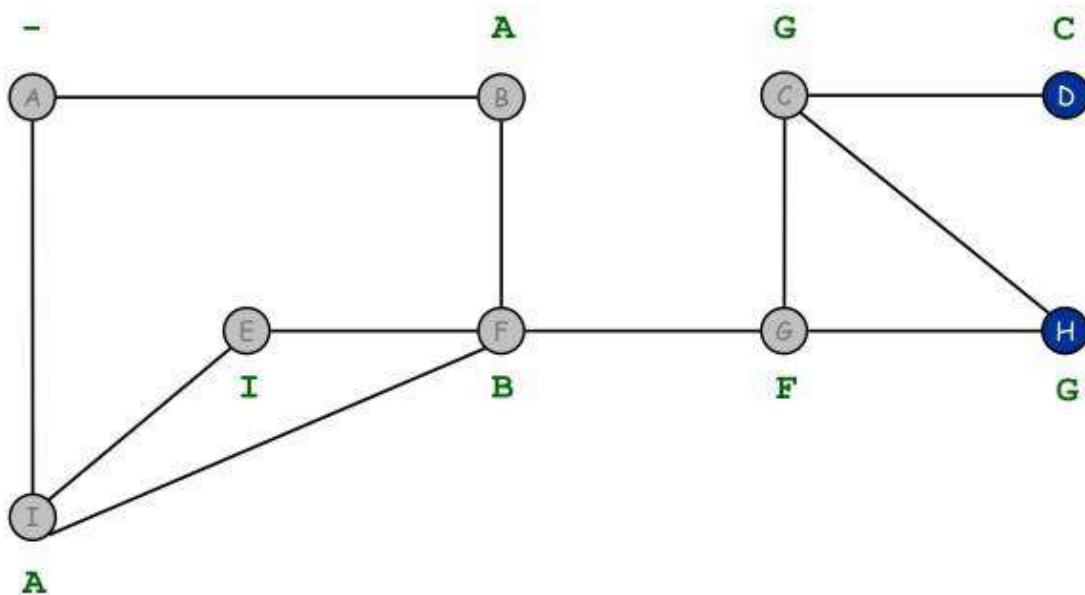


D discovered

front H D

## FIFO Queue

## Breadth First Search

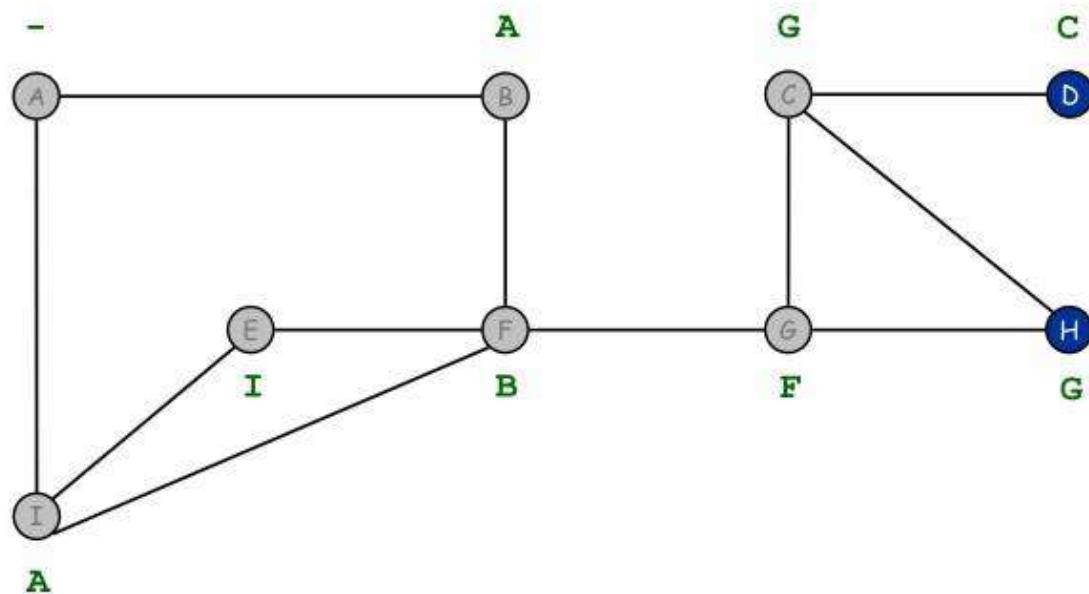


C finished

front H D

FIFO Queue

## Breadth First Search

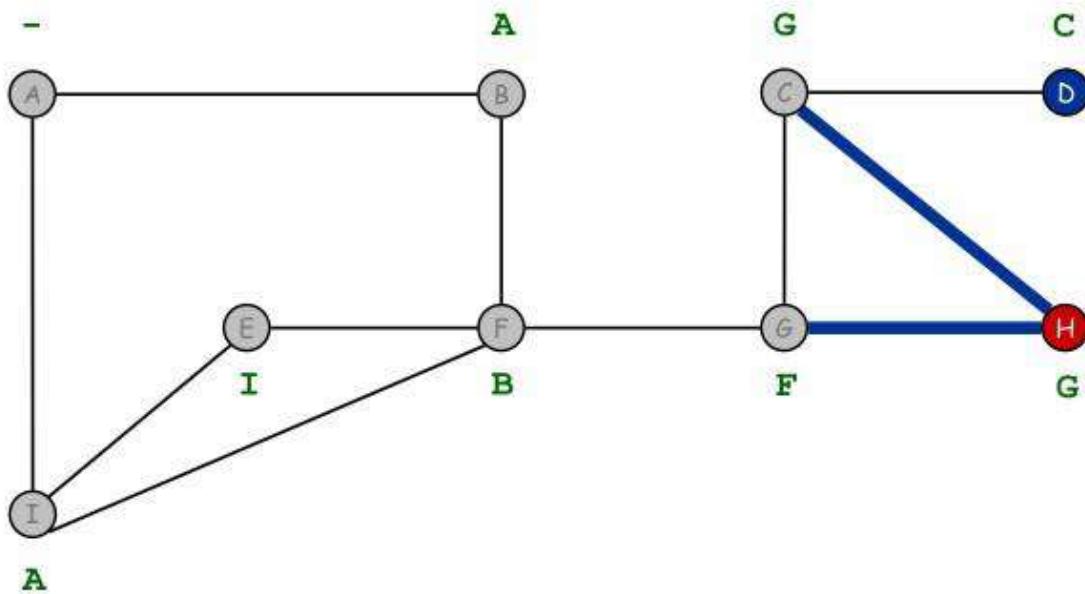


get next vertex

front H D

FIFO Queue

## Breadth First Search

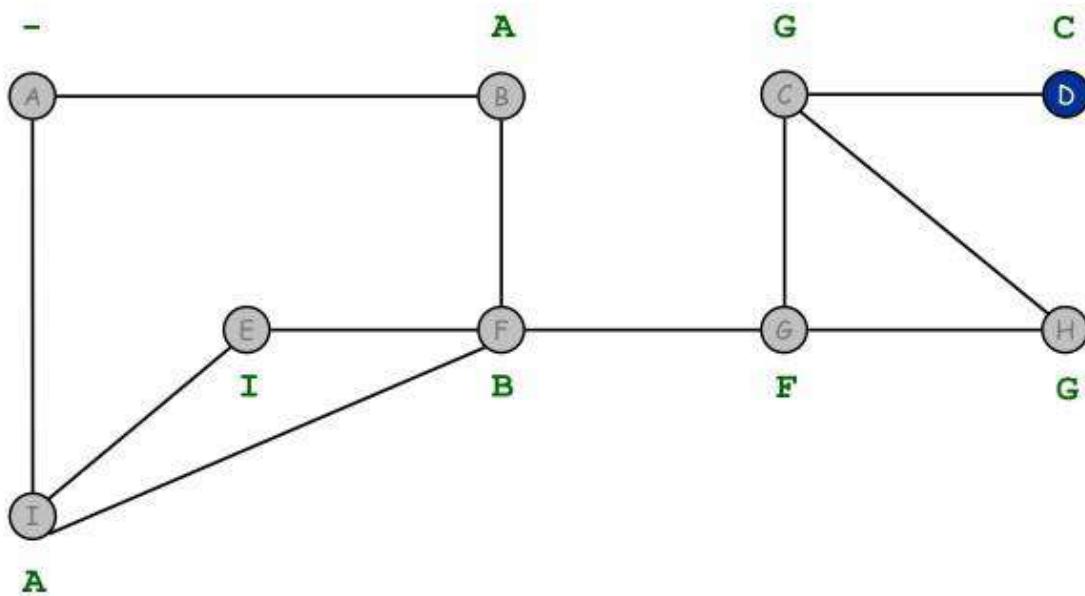


visit neighbors of  $H$

front D

## FIFO Queue

## Breadth First Search

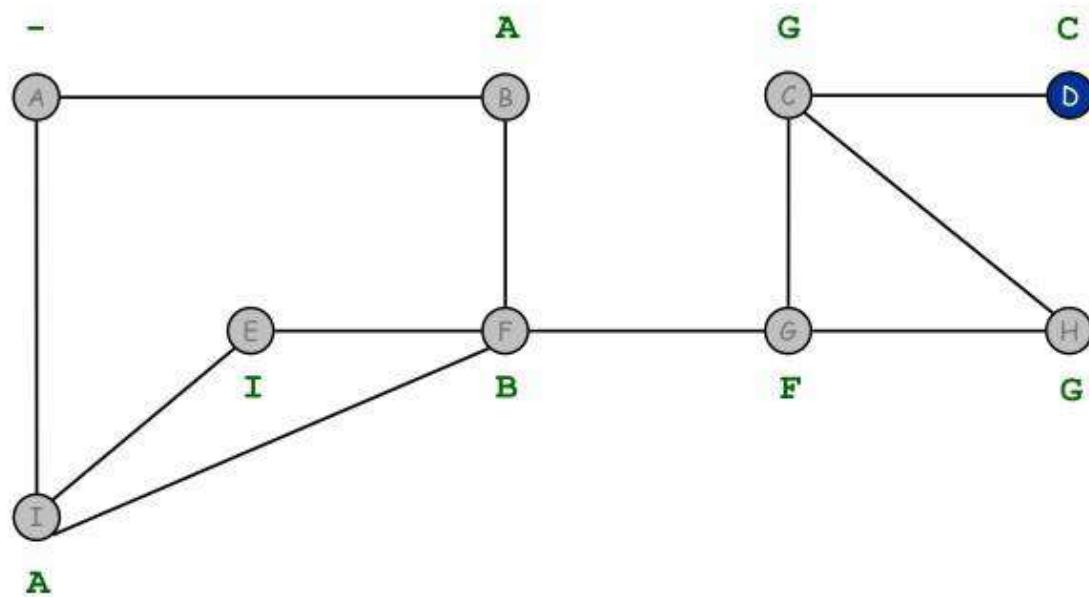


finished H

front D

FIFO Queue

## Breadth First Search

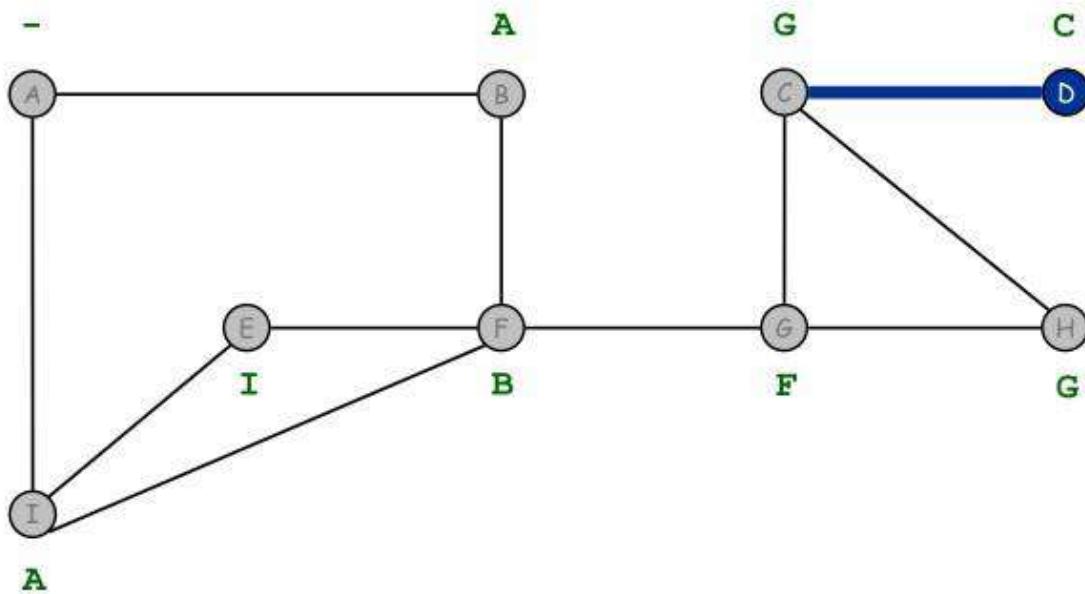


dequeue next vertex

front D

## FIFO Queue

## Breadth First Search

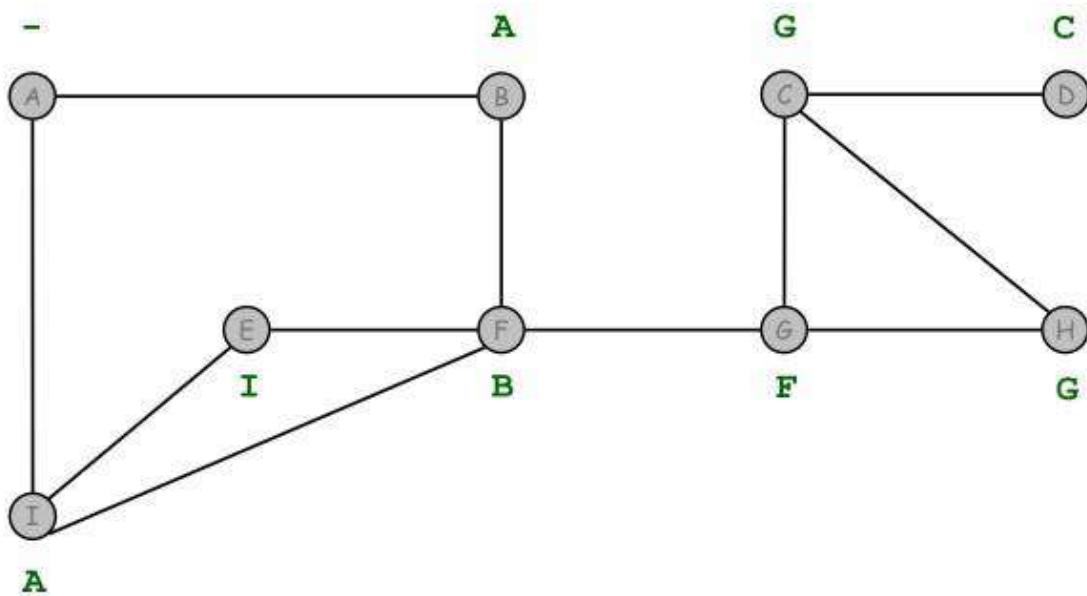


visit neighbors of D

front

FIFO Queue

## Breadth First Search

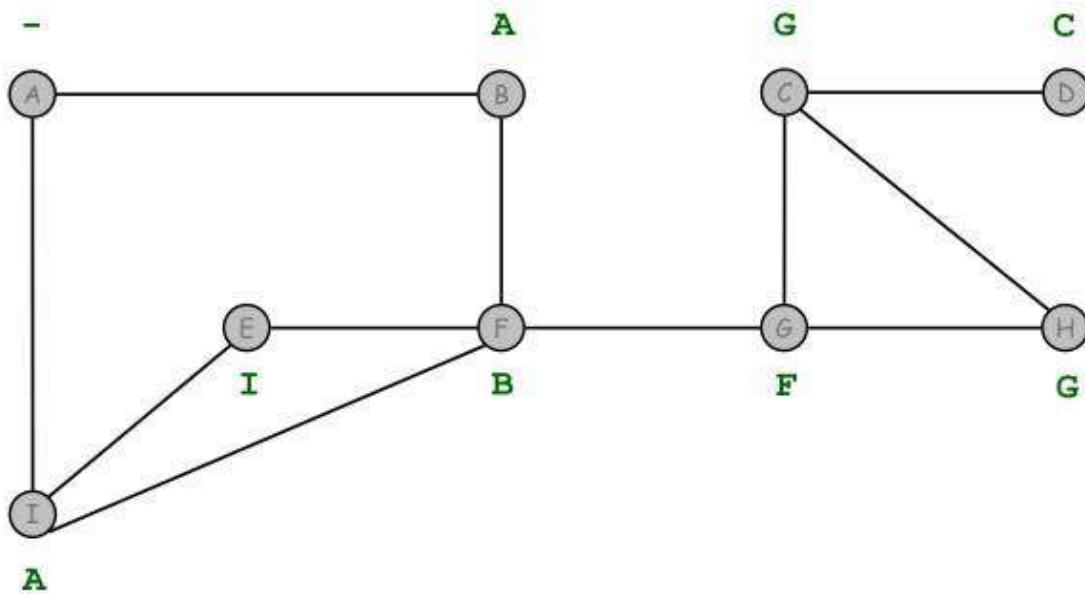


D finished

front

FIFO Queue

## Breadth First Search

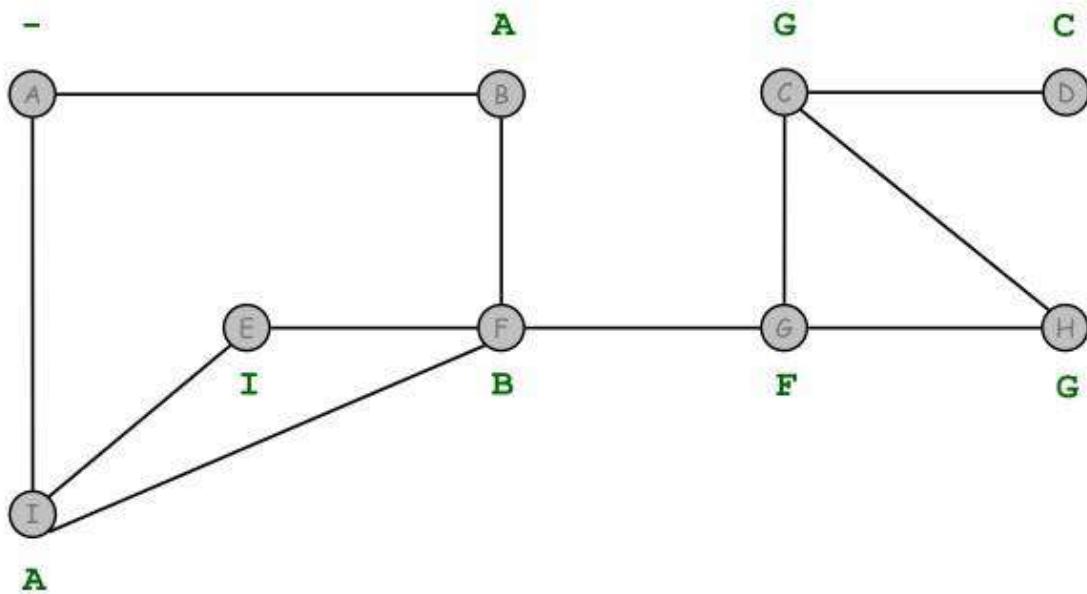


dequeue next vertex

front

FIFO Queue

## Breadth First Search

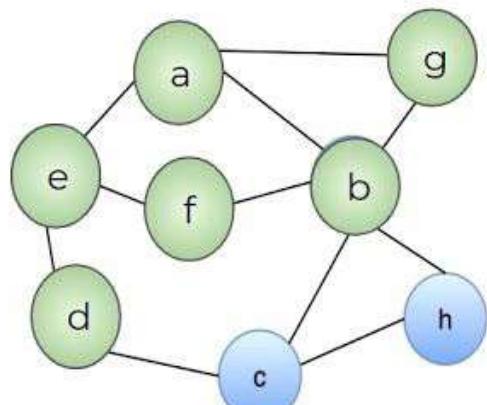


STOP

front

FIFO Queue

# Breadth First Search



Graph

Visited

```
a e b g d f
```

a	e	b	g			
b	a	g	f	c	h	
c	b	h	d			
d	c	e				
e	a	d	f			
f	e	b				
g	a	b				
h	b	c				

Adjacency List

**Q**  $\leftarrow$  b g d f  $\leftarrow$

**BFS order**

```
a e b g d f c h
```

**v** e

BFS

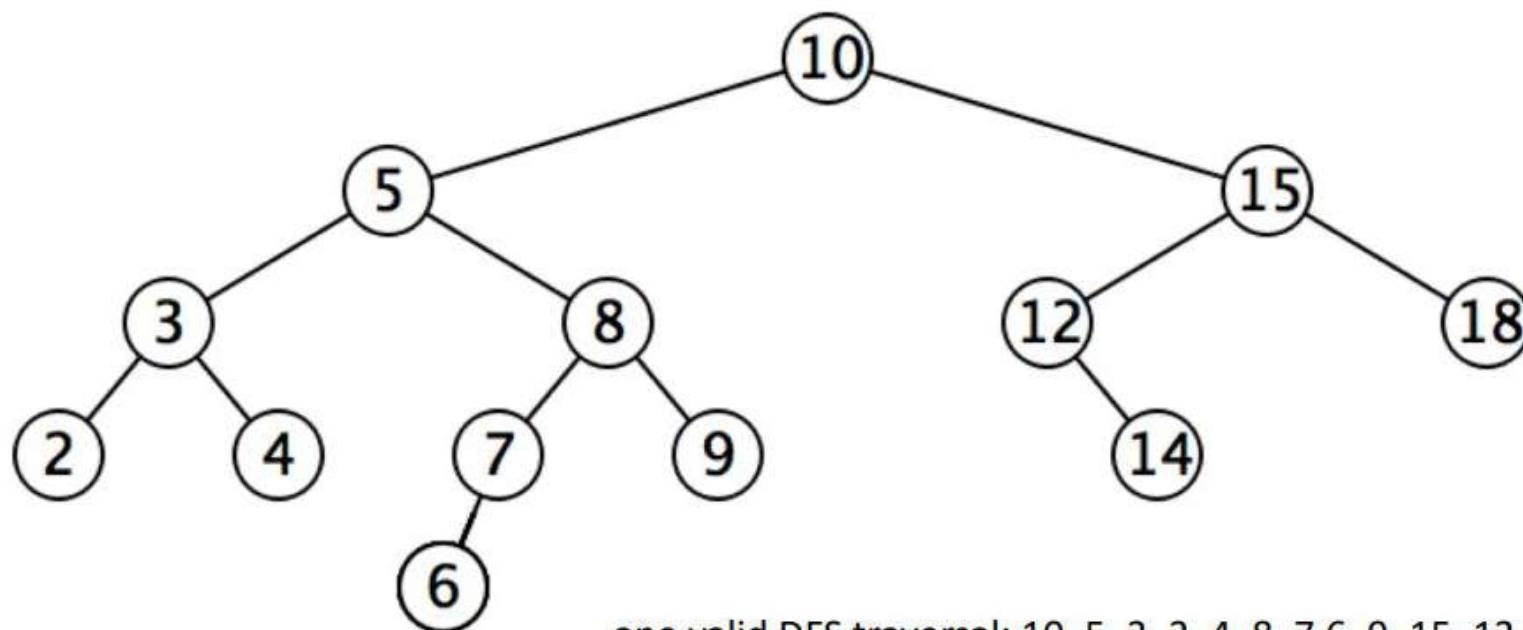
add start vertex to Q  
mark start as visited

while Q not empty  
   $v \leftarrow$  dequeue from Q  
  for each adj vertex  $av$  of  $v$   
    //.... process vertex  $av$ ....  
    if  $av$  is not visited  
      add  $av$  to Q  
      mark  $av$  as visited

# Graph traversals:

**Depth** First Search - a traversal on graphs (or on trees since those are also graphs) where you traverse “deep nodes” before all the shallow ones

High-level DFS: you go as far as you can down one path till you hit a dead end (no neighbors are still undiscovered or you have no neighbors). Once you hit a dead end, you backtrack / undo until you find some options/edges that you haven’t actually tried yet.



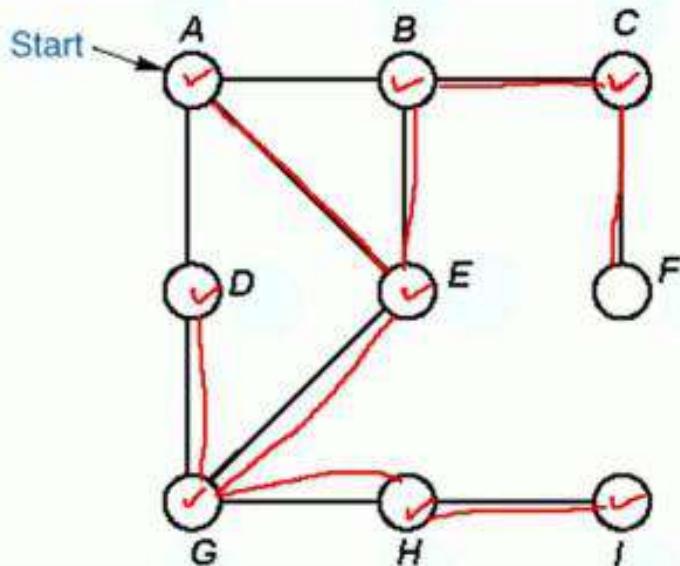
Kind of like wandering a maze – if you get stuck at a dead end (since you physically have to go and try it out to know it’s a dead end), trace your steps backwards towards your last decision and when you get back there, choose a different option than you did before.

# Depth-First Traversal

- A DFS starting at a vertex  $v$  first visits  $v$ , then some neighbour  $w$  of  $v$ , then some neighbour  $x$  of  $w$  that has not been visited before, etc.
- When it gets stuck, the DFS backtracks until it finds the first vertex that still has a neighbour that has not been visited before.
- It continues with this neighbour until it has to backtrack again.
- Eventually, it will visit all vertices reachable from  $v$
- Must keep track of vertices already visited to avoid cycles.
- The method can be implemented using recursion or iteration.
- A DFS traversal of a graph results in a depth-first tree or in a forest of such trees.

# Example

Depth-first traversal using an explicit stack.



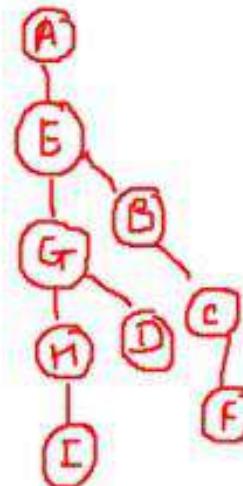
Order of  
Traversal

1	2	3	4	5	6	7	8	9
A	E	G	H	I	D	B	C	F

Visited  
array

1	2	3	4	5	6	7	8	9
✓	✓	✓	✓	✓	✓	✓	✓	✓

Dfs  
Tree



Stack  
is empty  
Stack is stop

Note: The DFS-tree for undirected graph is a free tree

# DFS Pseudo code

*dfs (v)*

*initialize stack s*

*initialize visited of all vertices to false*

*visited [v] = true*

*push(s,v)*

*while not empty(s)*

*v=pop(s)*

*add v into dfs sequence*

*for each vertex w adjacent to v*

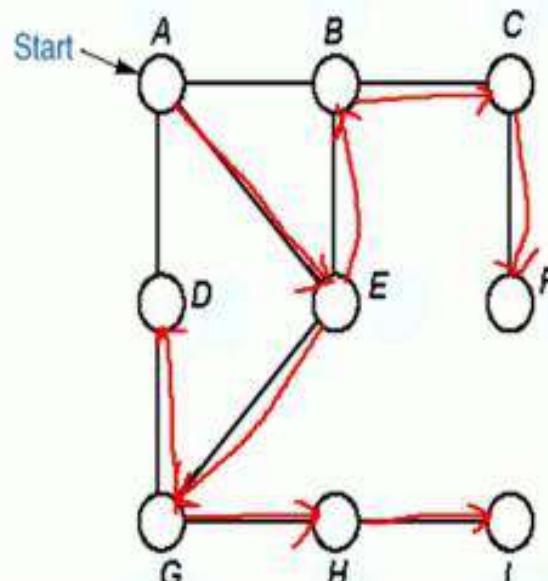
*if not visited [w] then*

*push(s,w)*

*visited [w]=true*

Order of  
Traversal

Visited array



Stack

1	2	3	4	5	6	7	8	9
A	E	G	H	I	D	B	C	F

A	B	C	D	E	F	G	H	I
✓	✓	✓	✓	✓	✓	✓	✓	✓

# Pseudo code (Iterative)

*DFS(Node start)*

*initialize stack s to hold start*  
*mark start as visited*  
*while(s is not empty)*  
    *next = s.pop() // and "process"*  
    *for each node u adjacent to*  
    *next*  
        *if(u is not marked)*  
            *mark u and push onto s*  
    *endwhile*  
*enddfs*

*BFS(Node start)*

*initialize queue q to hold start*  
*mark start as visited*  
*while(q is not empty)*  
    *next = q.dequeue() // and*  
    *"process"*  
    *for each node u adjacent to next*  
        *if(u is not marked)*  
            *mark u and enqueue onto q*  
    *endwhile*  
*Endbfs*



**SOMAIYA**

VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

Somaiya

TRUST 81

# When to use DFS or BFS to solve a Graph problem?

Most of the problems can be solved using either BFS or DFS. It won't make much difference. For example, consider a very simple example where we need to **count the total number of cities connected to each other**. Where in a graph nodes represent cities and edges represent roads between cities.

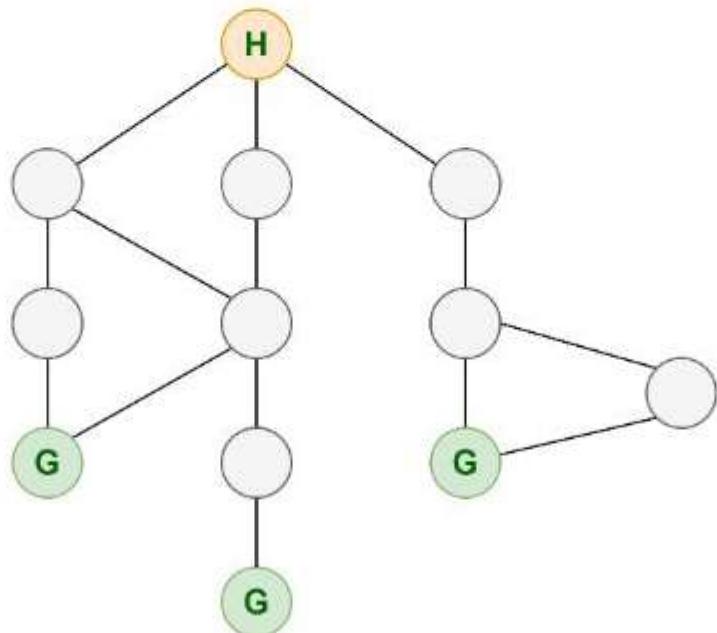
In such a problem, we know that we need to go to every node in order to count the nodes. **So it doesn't matter if we use BFS or DFS as using any of the ways we will be traversing all the edges and nodes of a graph.**

But there are problems when we need to decide to either use DFS or BFS for a **faster solution**. And there is no generalization. **It completely depends on the problem definition. It depends on what we are trying to find in the solution.** We need to understand clearly what our problem wants us to find. And the problem might not directly tell us to use BFS or DFS.

## Examples of choosing DFS over BFS.

### Example 1:

Consider a problem where you are standing at your house and you have multiple ways to go from your house to a grocery store. You are said that every path you choose has one store and is located at the end of every path. You just need to reach any of the stores.



Representation showing path from room to grocery store

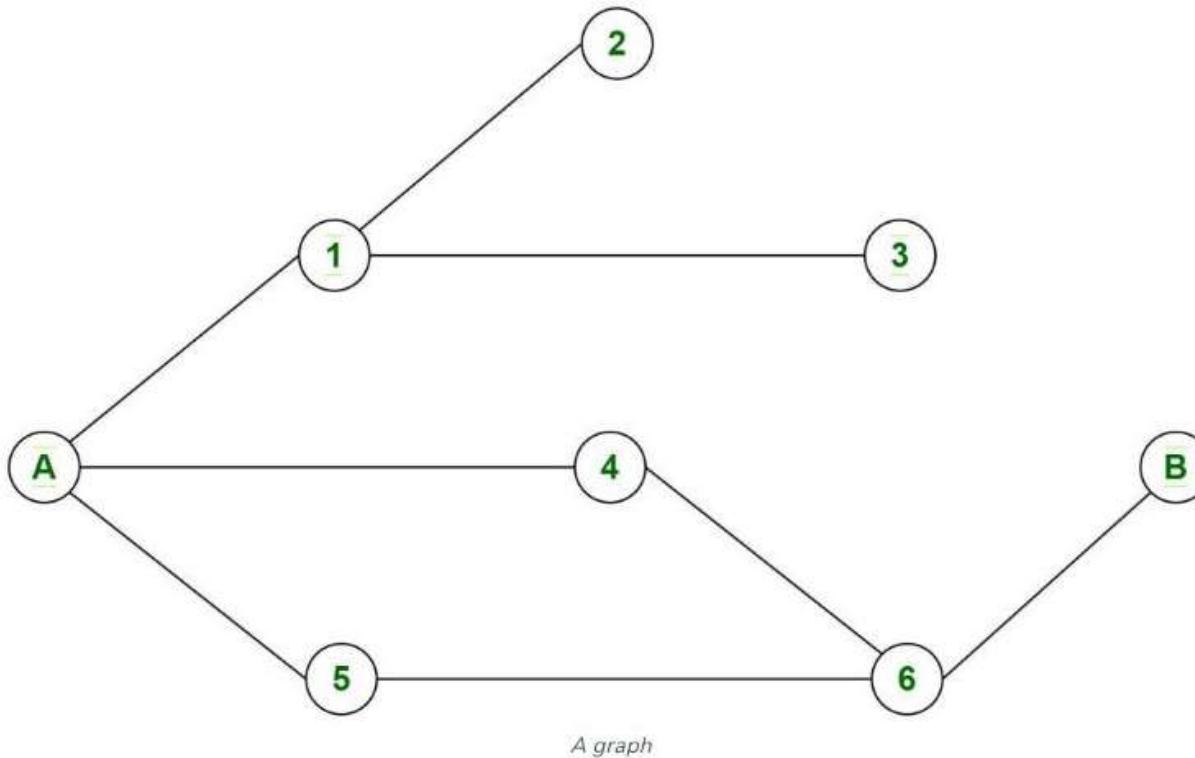
The obvious method here will be to choose **DFS**.

As we know we can find our solution (grocery store) in any of the paths, we can just go on traversing to any neighbor of the current node without exploring all the neighbors. There is no need of going through BFS as it will unnecessarily explore other paths but we can find our solution by traversing any of the paths. Also, we know that our solution is situated farthest from the starting point so if we choose BFS then we will have to almost visit all the nodes as we are visiting all nodes of a level and we will keep doing it till the end where we find a grocery store.

---

### Example 2:

Consider a problem where you need to print all the nodes encountered in any one of the paths starting from node A to node B in the diagram.

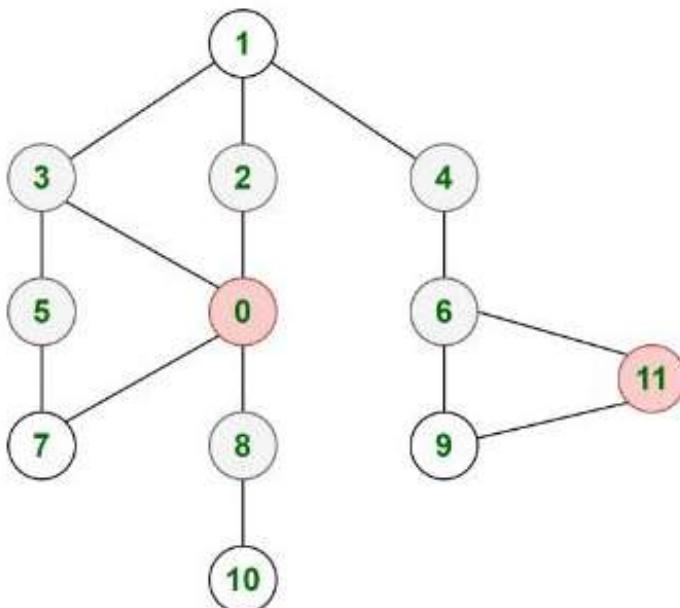


Here there are two possible paths "A  $\rightarrow$  4  $\rightarrow$  6  $\rightarrow$  B" and "A  $\rightarrow$  5  $\rightarrow$  6  $\rightarrow$  B". Here we require to keep track of a single path so there is no need of exploring every other path using BFS. Also, not every path will lead us from A to B. So we need to backtrack to the current node and then explore another path and see if that leads us to B. Need for backtracking tells us that we can think in the DFS direction.

## Examples of choosing BFS over DFS.

### Example 1:

Consider an example of a graph representing connected cities through edges. There are a few nodes colored in red that indicates covid affected cities. White-colored nodes indicate healthy cities. You are asked to find out the time the covid virus will take to affect all the non-affected cities if it takes one unit of time to travel from one city to another.



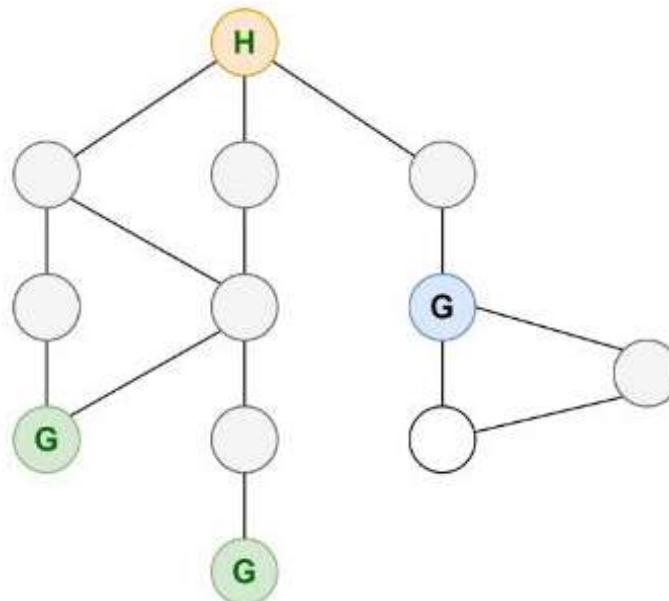
Graph representing above city arrangement

Here thinking of DFS is not even feasible. Here one affected city will affect all of its neighbors in one unit of time. This is how we know that we need to apply BFS as we need to explore all neighbors of the current node first. Another strong reason for BFS here is that both nodes 0 and 11 will start affecting neighbor cities simultaneously. **This shows we require a parallel operation on both nodes 0 and 11.** So we need to start traversing all the neighbor nodes of both nodes simultaneously. So we can push nodes 0 and 11 in the queue and start traversal parallelly. It will require 2 units of time for all the cities to get affected.

1. At time = 0 units, Affected nodes = {0, 11}
2. At time = 1 units, Affected nodes = {0, 11, 3, 2, 8, 7, 6, 9}
3. At time = 2 units, Affected nodes = {0, 11, 3, 2, 8, 7, 6, 9, 5, 1, 4, 10}

## Example 2:

Consider the same example of house and grocery stores mentioned in the above section. Suppose now you need to find the nearest grocery store from the house instead of any grocery store. Consider that each edge is of 1 unit distance. Consider the diagram below:



Graph representing grocery store

Here using DFS like previous will not be feasible. If we use DFS then we will travel down a path till we don't find a grocery store. But once we have found it we are not sure if it is the grocery store at the shortest distance. So we need to backtrack to find a grocery store on other paths and see if any other grocery store has a distance less than the current found grocery store. This will lead us to visit every node in the graph which is not probably the best way to do it.

We can use **BFS** here as BFS traverses nodes level by level. We first check all the nodes at a 1-unit distance from the house. If any of the nodes is a grocery store then we can stop else we will see the next level i.e all the nodes at a distance 2-unit from the house and so on. This will take less time in most situations as we will not be traversing all the nodes. For the given graph we will only explore nodes up to two levels as at the second level we will find the grocery store and we will return the shortest distance to be 2.

## Conclusion:

We can't have fixed rules for using BFS or DFS. It totally depends on the problem we are trying to solve. But we can make some general intuition.

- We will prefer to use BFS when we know that our solution might lie closer to the starting point or if the graph has greater depths.
- We will prefer to use DFS when we know our solution might lie farthest from the starting point or when the graph has a greater width.
- If we have multiple starting points and the problem requires us to start traversing all those starting points parallelly then we can think of BFS as we can push all those starting points in the queue and start exploring them first.
- It's generally a good idea to use BFS if we need to find the shortest distance from a node in the unweighted graph.
- We will be using DFS mostly in path-finding algorithms to find paths between nodes.

## **VLAB Link for experiment**

DFS:

<https://ds1-iiith.vlabs.ac.in/exp/depth-first-search/index.html>

BFS:

<https://ds1-iiith.vlabs.ac.in/exp/breadth-first-search/index.html>

