

# **Advanced Data Structures**



**UNIT-I TREES**

# UNIT1:Trees



- **Syllabus**

- Difference between linear and non-liner data structure
- Trees and Binary trees - basic terminology, representation using (array and) linked organization
- Binary tree- properties, converting tree to binary tree
- Binary tree traversals recursive and non-recursive
- Level wise Depth first and Breadth first.
- Binary Search Tree (BST), BST operations

# What is Abstract Data Type (ADT)?



- The Data Type which can be represented in the form of its operation is called as ADT
- Abstract means hide
- *What is to be done is given but how it is implemented is hidden.*

ADT

Declaration of data

Declaration of operation

Encapsulation of data & operation (wrapping the variables and methods together as a single unit)

# NEW TOPIC (UNIT1:Trees)



- Syllabus
- Difference between linear and non-liner data structure
- Trees and Binary trees - basic terminology, representation using linked organization
- Binary tree- properties, converting tree to binary tree
- Binary tree traversals recursive and non-recursive
- Level wise Depth first and Breadth first.
- Binary Search Tree (BST), BST operations

# Linear Data Structures & Non Linear Data Structures



Linear: Ex. Linked list, stack, queue

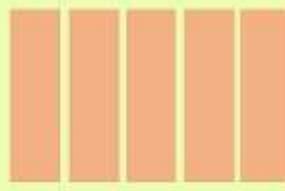
- Elements are arranged in linear fashion (in sequence.)
- Only one-one relation can be handled using linear data structure.

• Non Linear Data Structures :Ex. Tree, graphs

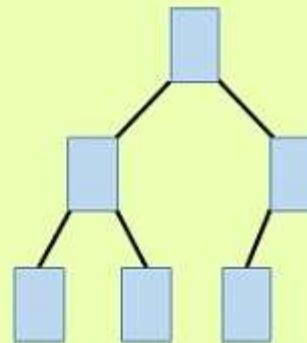
All one-many, many-one , many-many relations are handled using non linear data structures.

- Here every data element can have number of predecessors as well as successors.

# Difference in linear and non-linear data structure



**Linear Data  
Structure**



**Non -Linear Data  
Structure**

# Linear VS Non-Linear Data Structure



## LINEAR

- Level Single
- Traversal Easy
- Implementation Easier
- E.g. Array, Linked List, Stack, Queue
- Data elements can be connected to adjacent data elements in the sequence

## NON-LINEAR

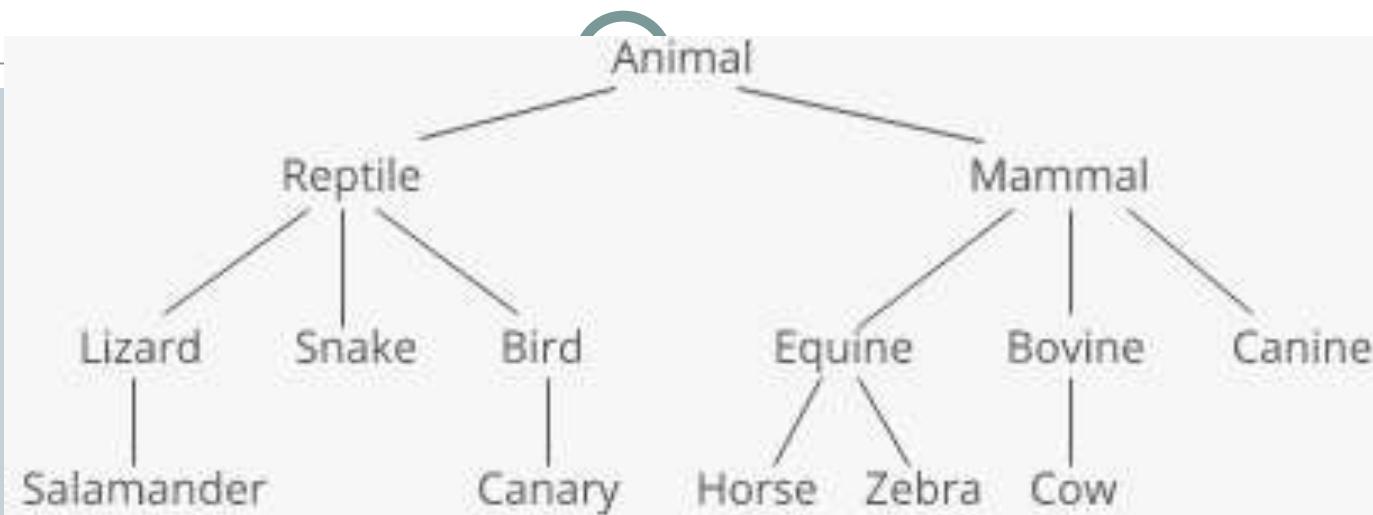
- Level Multilevel
- Traversal Complex
- Implementation Complex
- E.g. Trees, Graphs
- Data elements can be connected to 2 or more than 2 other data elements to signify / reflect more intricate relationship

# NEW TOPIC (UNIT1:Trees)



- Syllabus
- Difference between linear and non-linear data structure
- Trees and Binary trees - basic terminology, representation using (array and) linked organization
- Binary tree- properties, converting tree to binary tree
- Binary tree traversals recursive and non-recursive
- Level wise Depth first and Breadth first.
- Binary Search Tree (BST), BST operations

A **tree** organizes values hierarchically.

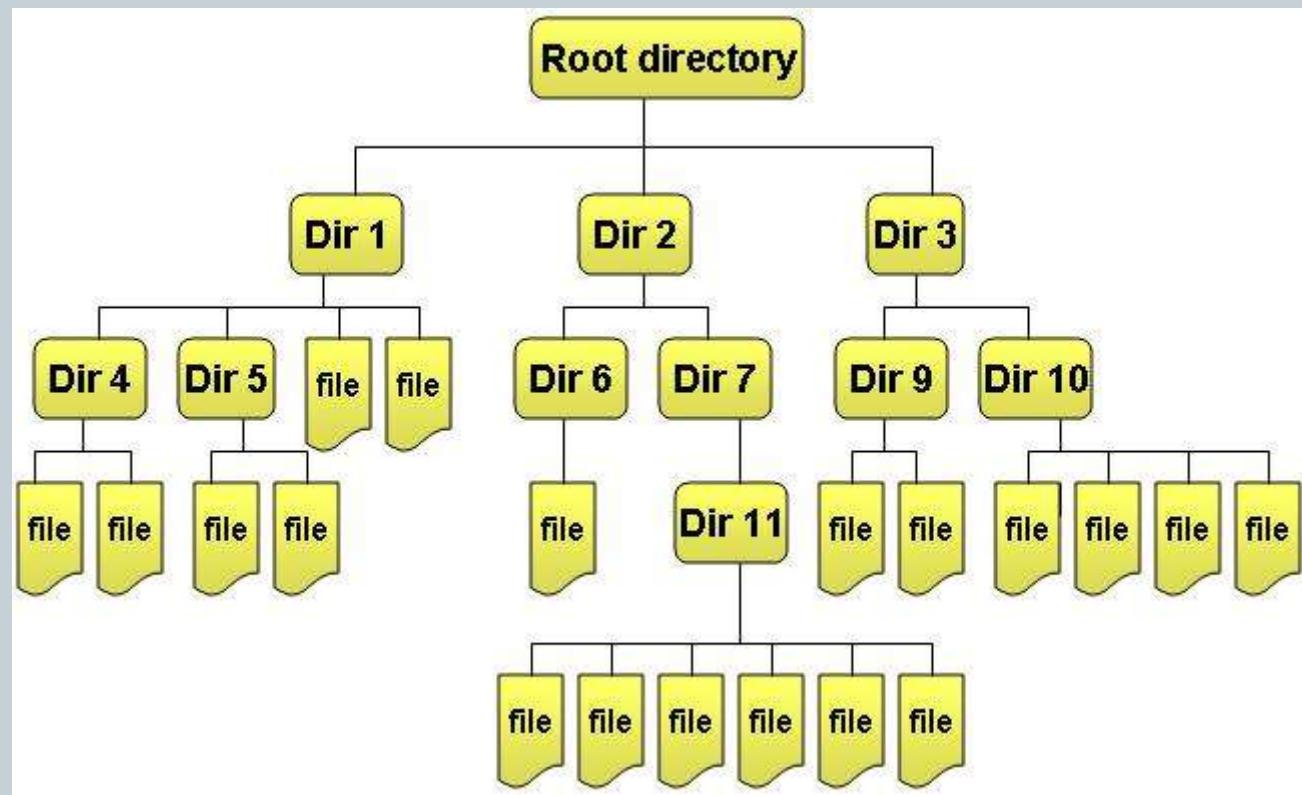


Each entry in the tree is called a **node**, and every node links to zero or more child nodes.

**Example uses:**

- **Filesystems**—files inside folders inside folders
- **Comments**—comments, replies to comments, replies to replies
- **Family trees**—parents, grandparents, children, and grandchildren

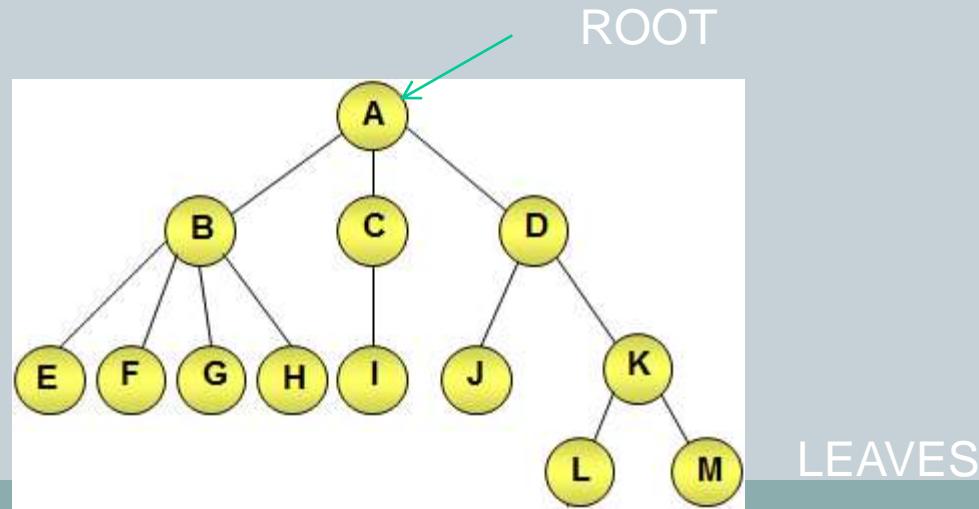
# Example: Hierarchy in Data Elements



# Tree Basics



- Nonlinear data structure
- Represents a **hierarchical relationship** among the various data elements
- Used in applications where **relation between data elements needs to be represented in a hierarchy**

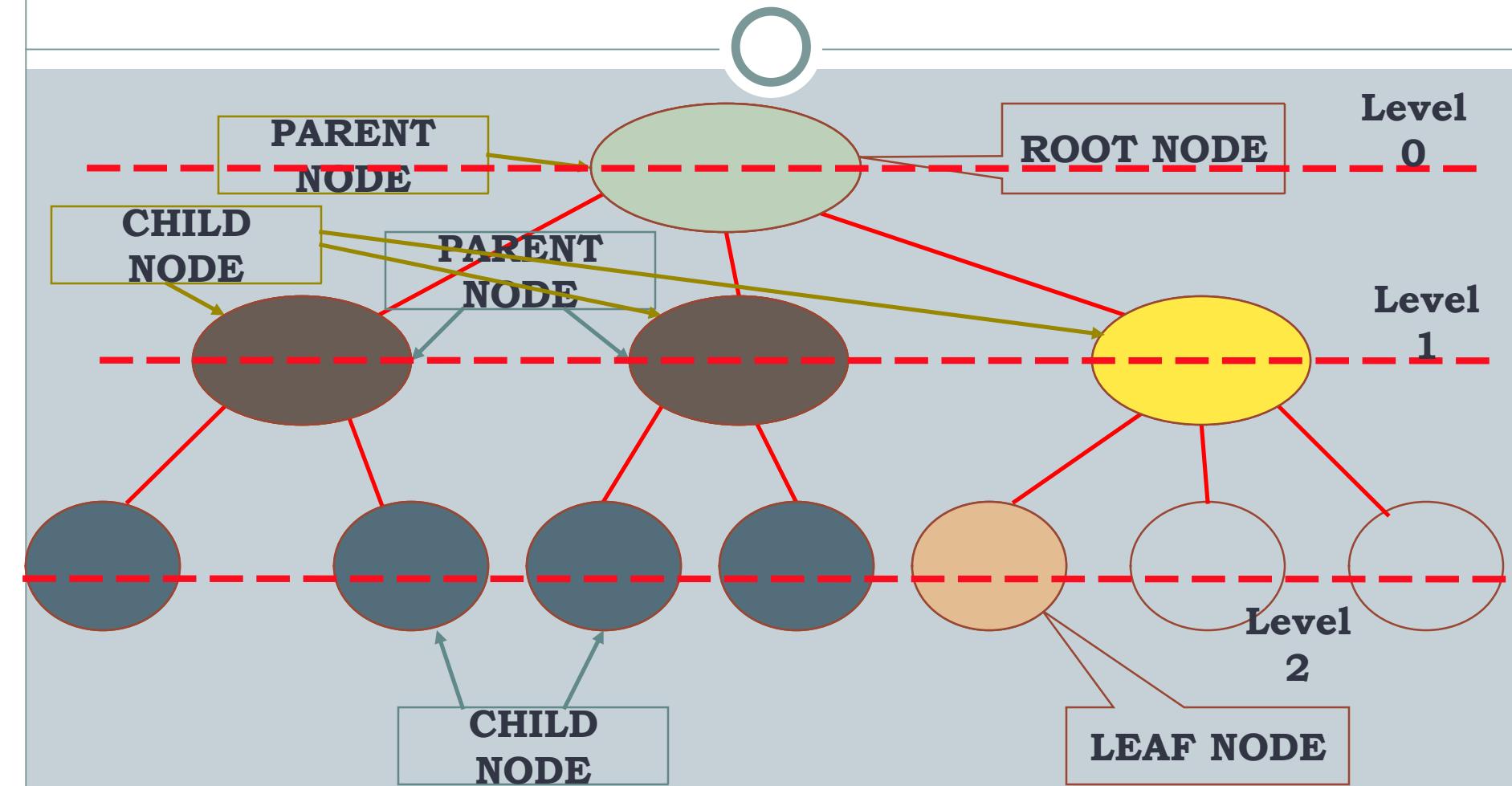


# *Definition of Tree*



- A tree is a **finite set of one or more nodes** such that:
  - There is a **specially** designated node called the **root**.
  - The remaining nodes are partitioned into  $n \geq 0$  disjoint sets  $T_1, \dots, T_n$ , where each of these sets is a tree.
  - We call  $T_1, \dots, T_n$  the **subtrees** of the root.

# TREE Representation



# *Definition of Tree*



- A tree is a finite set of one or more nodes such that:
  - There is a specially designated node called the root.
  - The remaining nodes are partitioned into  $n \geq 0$  **disjoint sets**  $T_1, \dots, T_n$ , where each of these sets is a tree.
  - We call  $T_1, \dots, T_n$  the subtrees of the root.



# Basic Terminology

# *Terminology*



- Finite set of elements in tree are called **NODES**
- Finite set of lines in tree are called as **BRANCHES**
- Branch directed towards the node is the **INDEGREE** of the node
- Branch directed away from the node is the **OUTDEGREE** of the node
- The degree of a node is the number of sub trees of the node
- INDEGREE of ROOT is **ZERO**

# *Terminology*



- All other nodes of the tree will have **in degree** one and out degree can be zero or more
- OUTDEGREE of LEAF node is **ZERO**
- A node that is neither ROOT nor LEAF is **INTERNAL NODE**
- The total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node.
- In a tree, height of the root node is said to be **Height of the tree**.
- A node is called **PARENT** node if it has successor nodes i.e. outdegree greater than ZERO

# *Terminology*

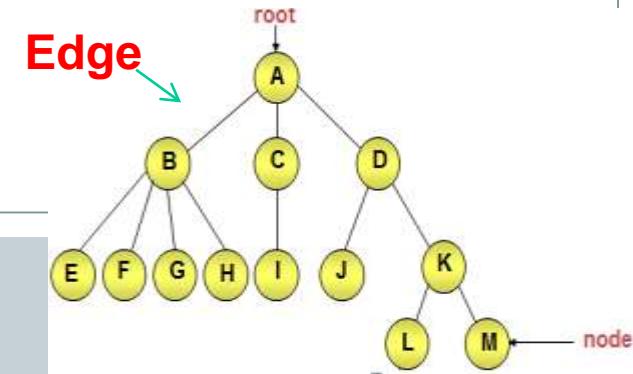


- The node with degree 0 is a leaf or terminal node.
- A node that has sub trees is the **parent** of the sub trees.
- The sub trees of the node are the **children** of the node.
- Children of the same parent are **siblings**.

# (Repeat) Tree Terms



- **Node:** Element in tree
  - **Root:** Topmost node
  - **Child:** Node immediately lower in hierarchy to a node, which is root of **subtree**
  - **Parent :** Node immediately above in hierarchy to a node; Node of whose subtree it is a root
  - **Sibling:** Children of the same parent
  - **Ancestors** of a node : All the nodes along the path from the root to the node
  - **Descendants** of a node: All the nodes along the path from the node to the root
  - **Edge / Branch:** Link from the parent to a child node is referred to as an edge
  - **Leaf** Node: node with no child
  - **Internal** Node: Any node between Root node and a Leaf node



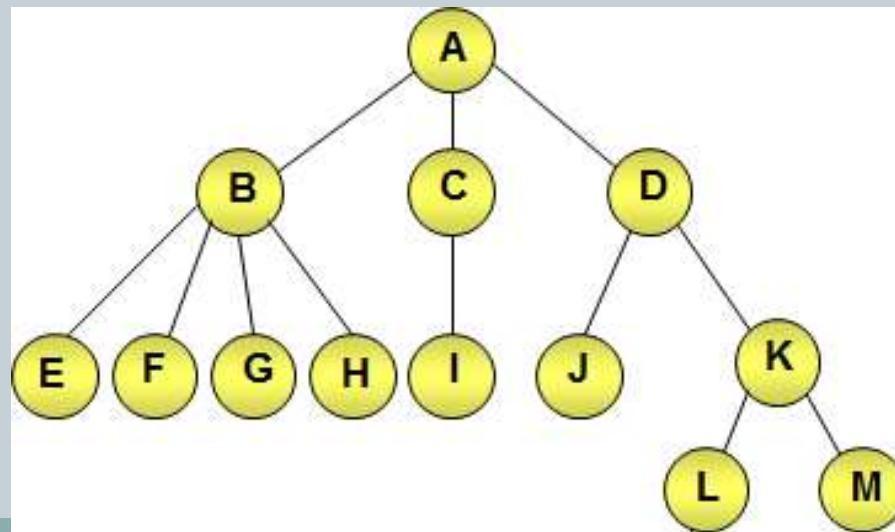
# (Repeat) Tree Terms Cond...



**Siblings/Brothers:** Refers to the children of the same node

**Ancestor:** A,D,K are Ancestor of M

**Descendant:** M,L,K,J are Descendants of D



Nodes B, C, and D are siblings of each other.

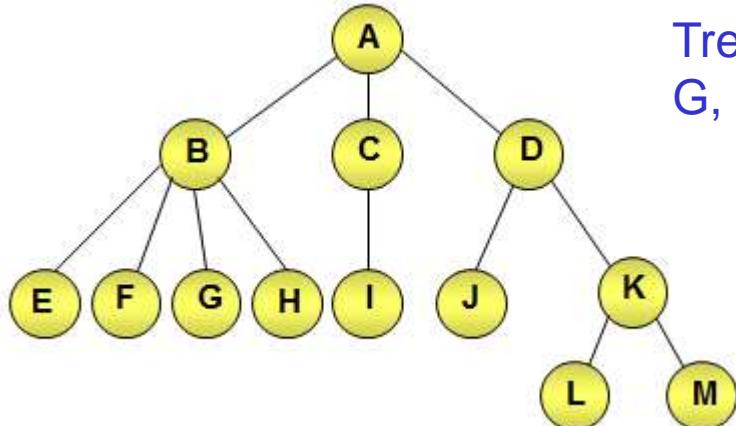
Nodes E, F, G, and H are siblings of each other

# Tree Terms Cond...



## Subtree:

- Portion of a tree, which can be viewed as a separate tree in itself is called a subtree
- Tree below a node in hierarchy, is its subtree
- Tree can contain just 1 node called leaf node



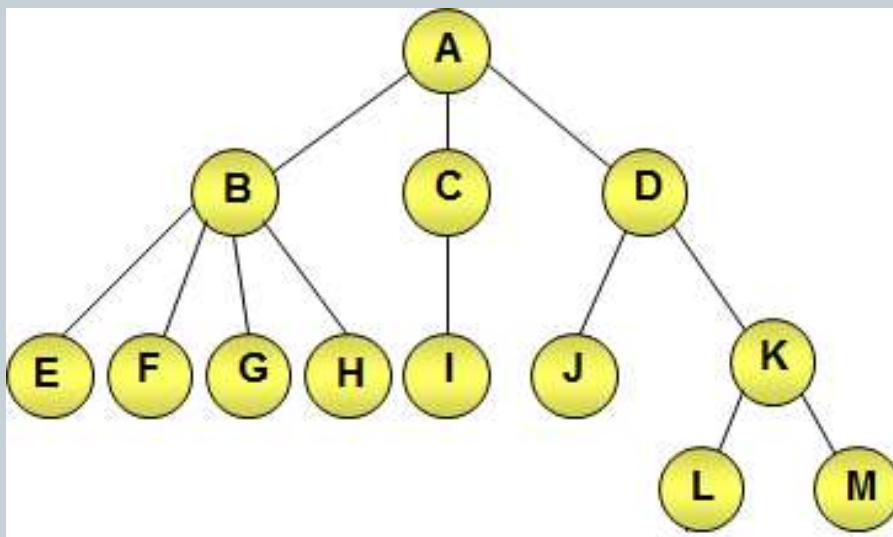
Tree with root B, containing nodes E, F, G, and H is a subtree of node A

E, F, G, and H are children of node B. B is the parent of these nodes.

# Tree Terms Cond...



- **Degree of a node:** Number of subtrees of a node in a tree



Degree of node G is  
0

Degree of node D is  
2

Degree of node A is  
3

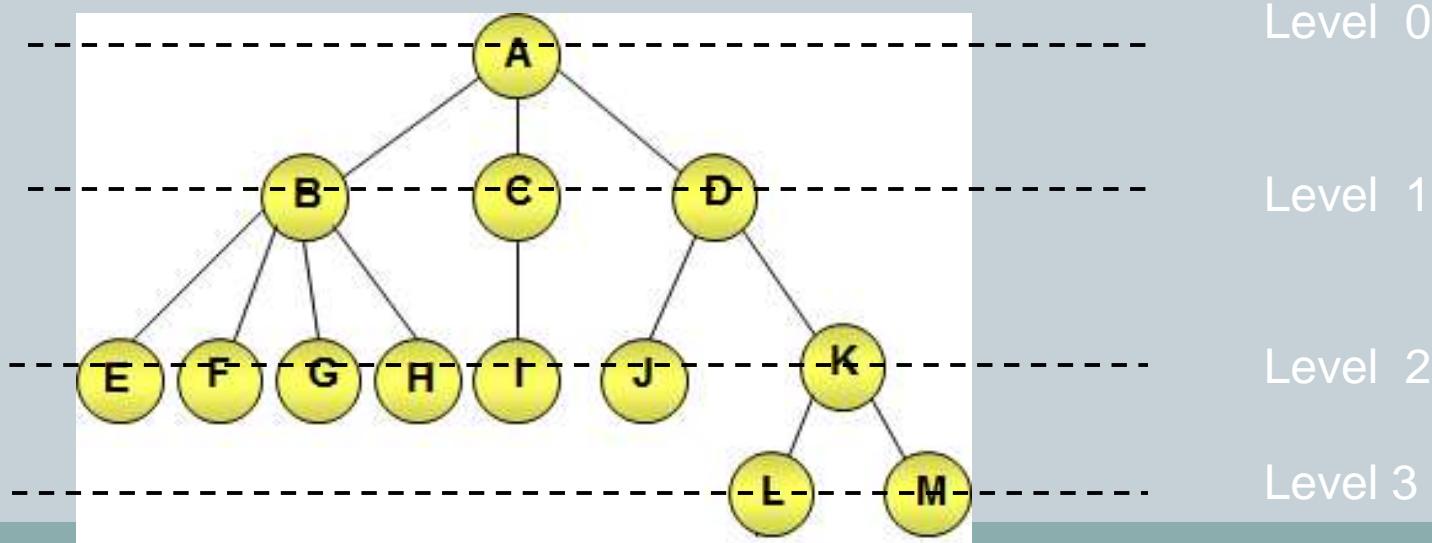
Degree of node C is  
1

# Tree Terms Cond...



**Level of a node:** The distance (in number of nodes) of a node from the root. Root always lies at level 0.

As you move down the tree, the level increases by one.



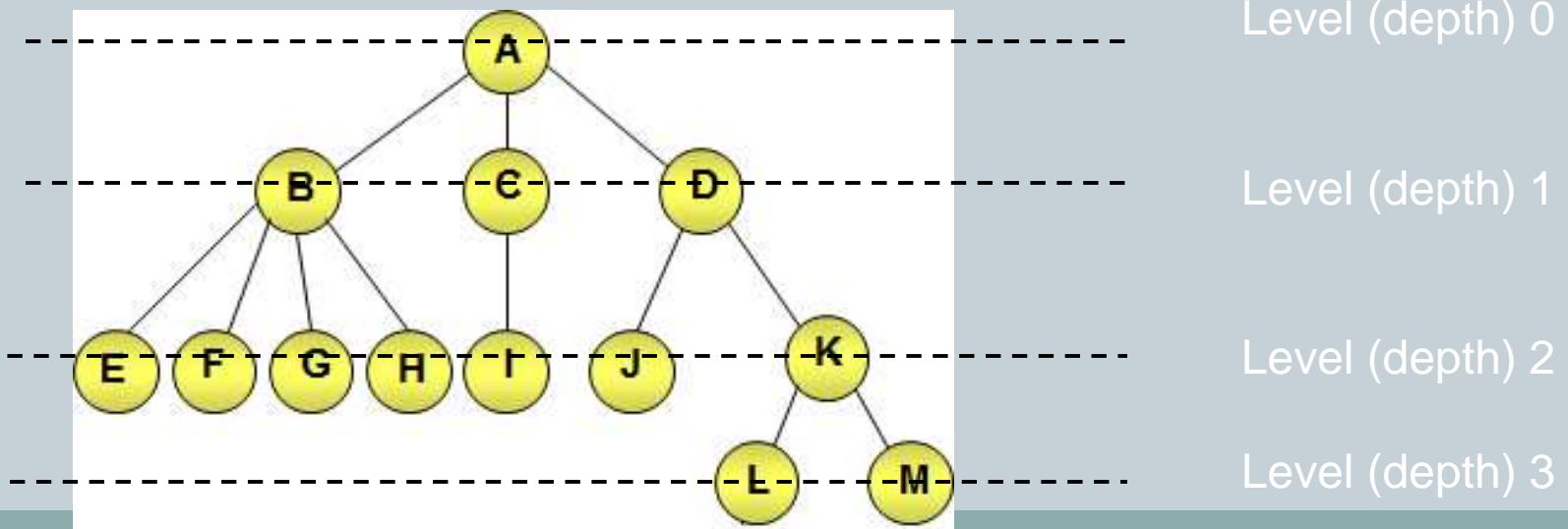
# Tree Terms Cond...



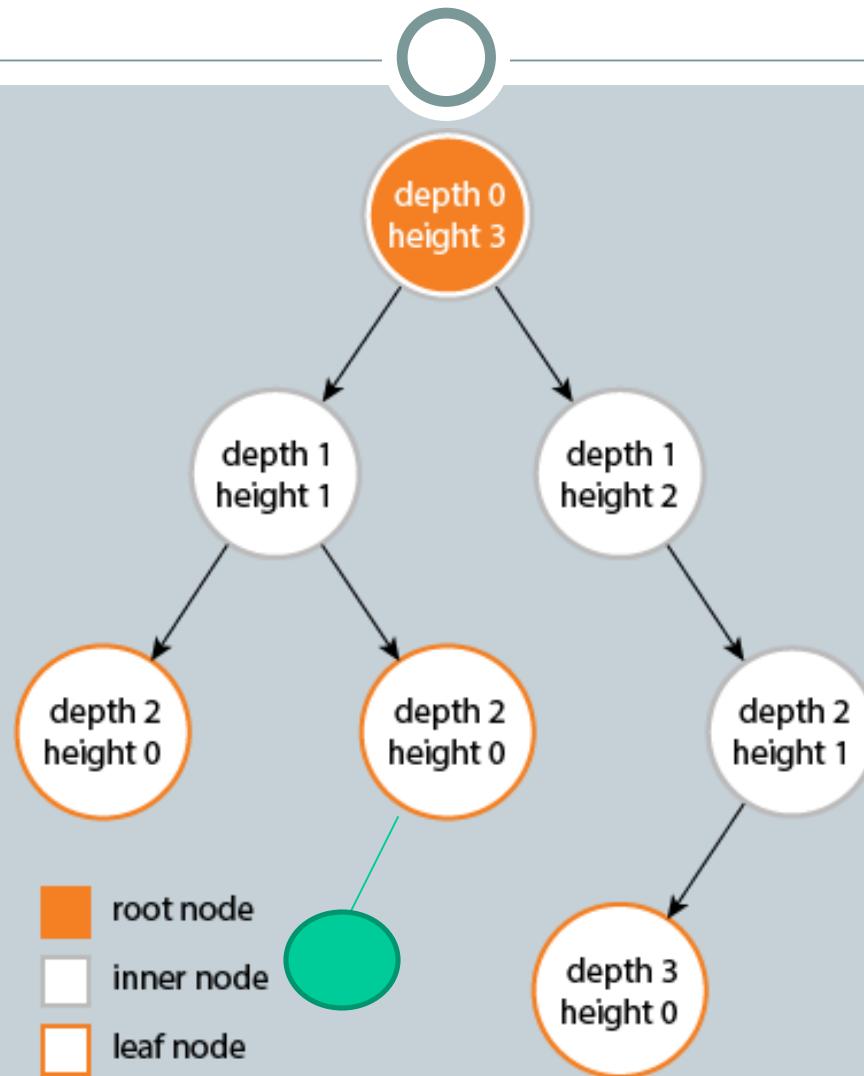
**Depth of a node:** Length of path (number of edges) from root to that node

Depth of K is 2, and A is 0 in the following tree

**Depth / Height of Tree:** Maximum level of any node in the tree



# Height of a node X? Ans: Distance from node X to deepest leaf



Q)  


Distance of a node from root is \_\_\_\_\_ of the node.

- a) Height
- b) Depth
- c) Length
- d) **Level**

The number of edges from the root to a node is called \_\_\_\_\_ of the node.

- a) Height
- b) **Depth**
- c) Length
- d) Level

The number of edges from the node to its deepest leaf is called \_\_\_\_\_ of the node.

- a) **Height**
- b) Depth
- c) Length
- d) None of the mentioned

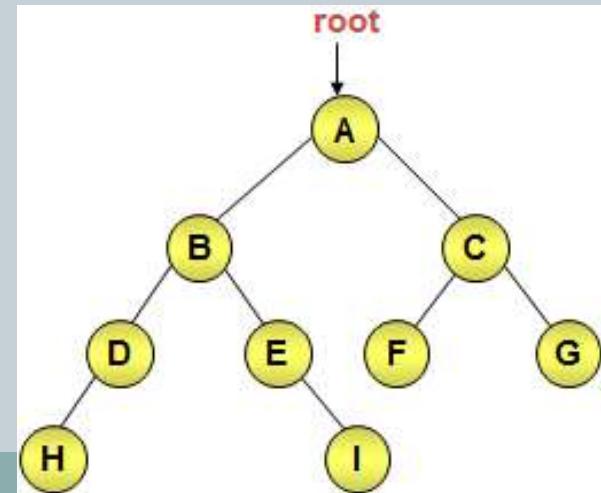
The number of edges from the root to the deepest leaf is called \_\_\_\_\_ of the tree.

- a) **Height**
- b) Depth
- c) Length
- d) None of the mentioned

# Q) Consider the following tree and answer the questions that follow



- a) What is the depth of the tree?
- b) Which nodes are children of node B?
- c) Which node is the parent of node F?
- d) What is the level of node E?
- e) Which nodes are the siblings of node H?
- f) Which nodes are the siblings of node D?
- g) Which nodes are leaf nodes?

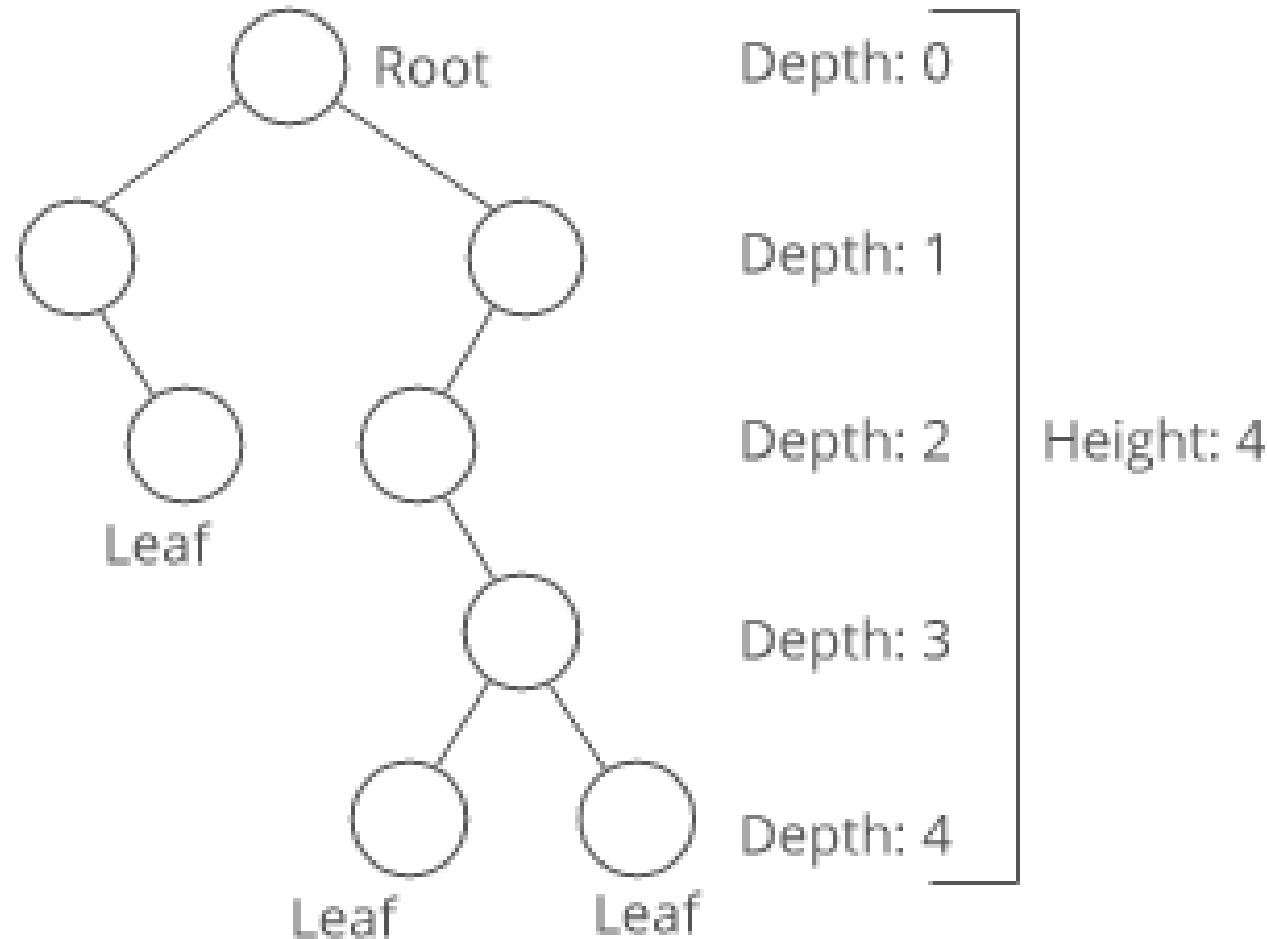




Answer:

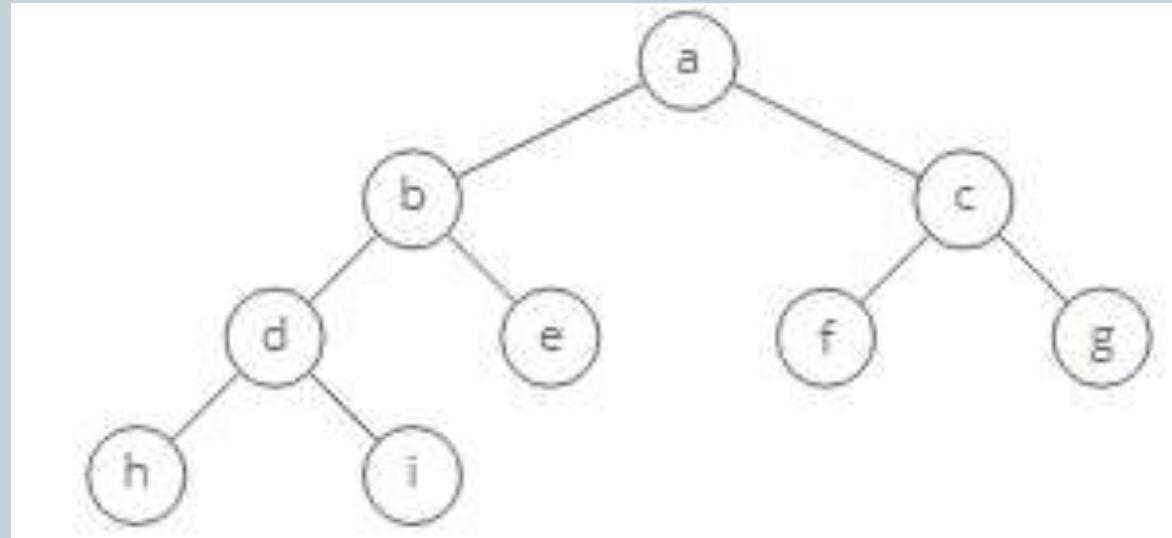
- a. 3
- b. D and E
- c. C
- d. 2
- e. H does not have any siblings
- f. The only sibling of D is E
- g. F, G, H, and I

# Recap



Q)  
()

- Ancestors of e?
- Siblings of f?
- Root?
- Leaves?



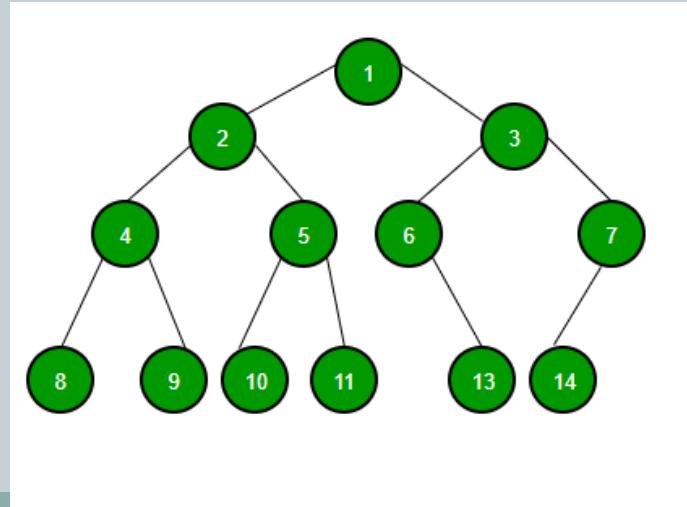
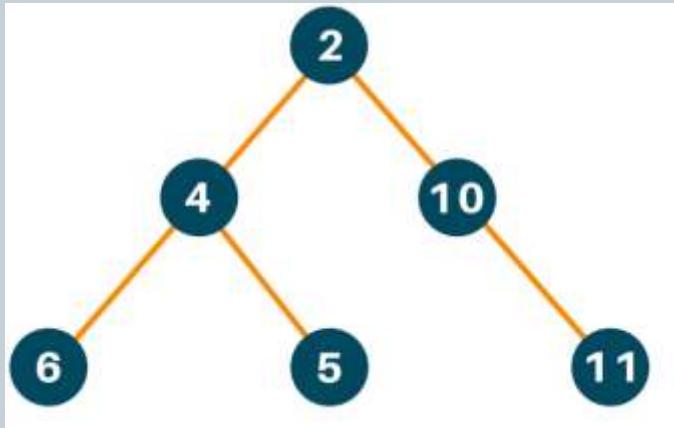


# Binary Tree

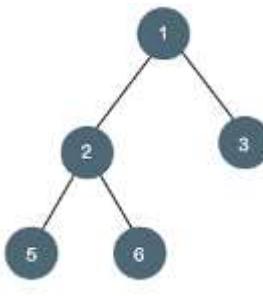
# Binary Tree Definition



- A *Binary tree* is a **finite set of nodes** that either is **empty** or consists of a root and two disjoint binary trees called the *left subtree* and *right subtree*
- A node in a binary tree can have max 2 children (i.e. 0 / 1/ 2 children)



# Binary Tree



- Tree in which each node can have at most two children (i.e. zero/one/two)
- Its Children are named “left child” and “right child”
- **Empty tree** is also a **valid** binary tree
- Unlike general trees, **order** of left/right child is **important**
- Any Tree can be represented as a Binary Tree
- Important tree structure that occurs very often

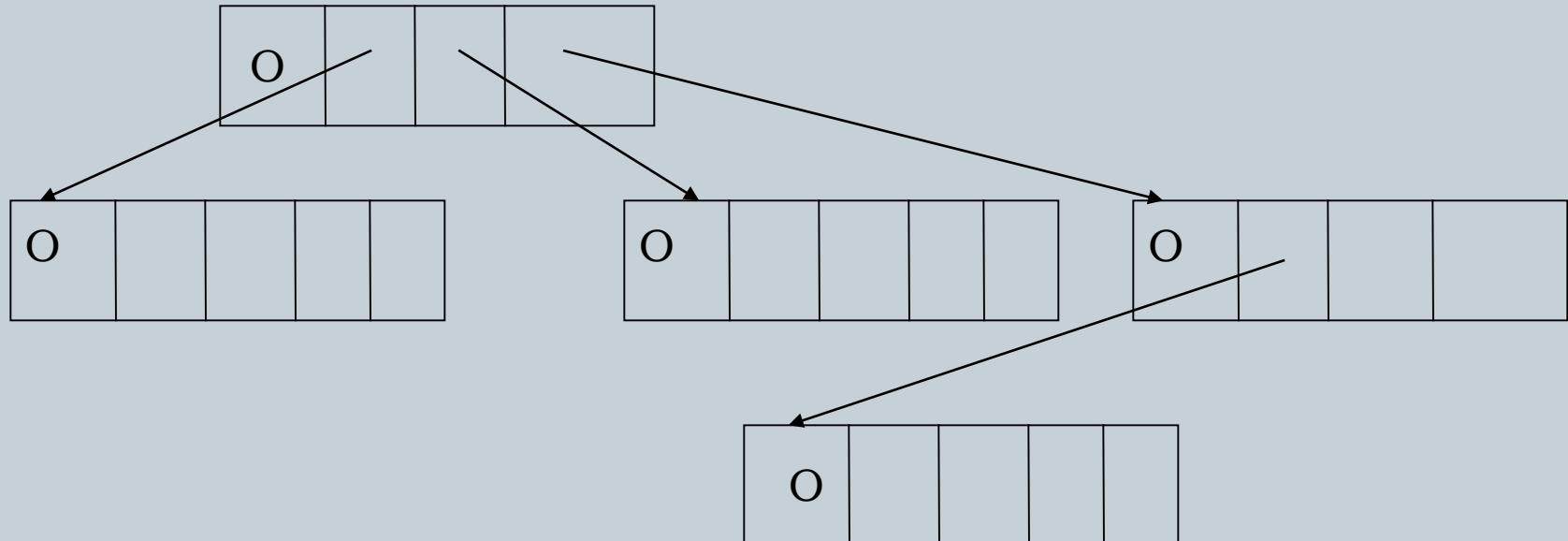


# Representation Using Linked Organization

# A Tree Node



- Every tree node:
  - object – useful information
  - children – pointers to its children nodes

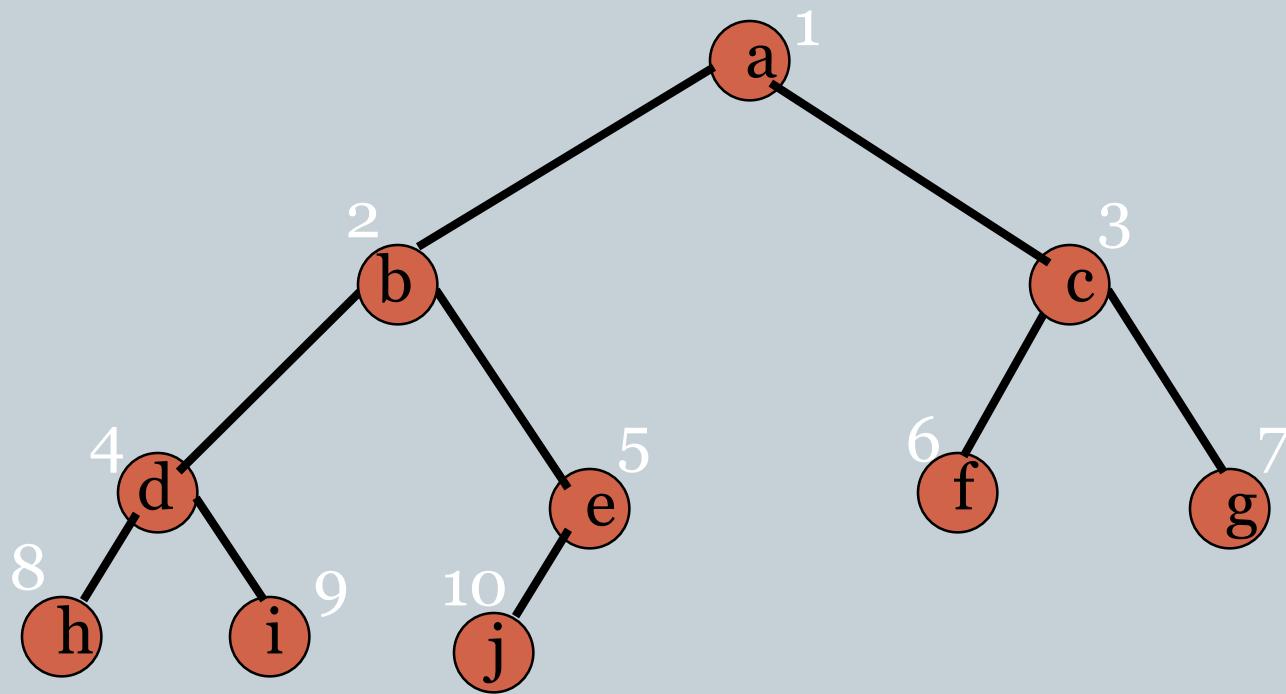


# Binary Tree Representation

- Array representation.
- Linked representation.

# Array Representation

- Number the nodes using the numbering scheme for a full binary tree. The node that is numbered  $i$  is stored in `tree[i]`.



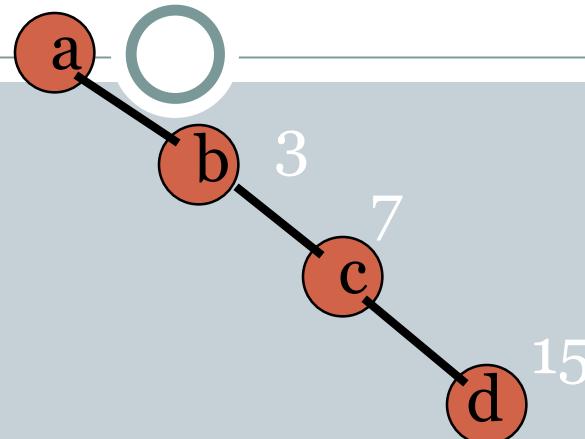
`tree[]` [ a | b | c | d | e | f | g | h | i | j ]

0

5

10

# Right-Skewed Binary Tree



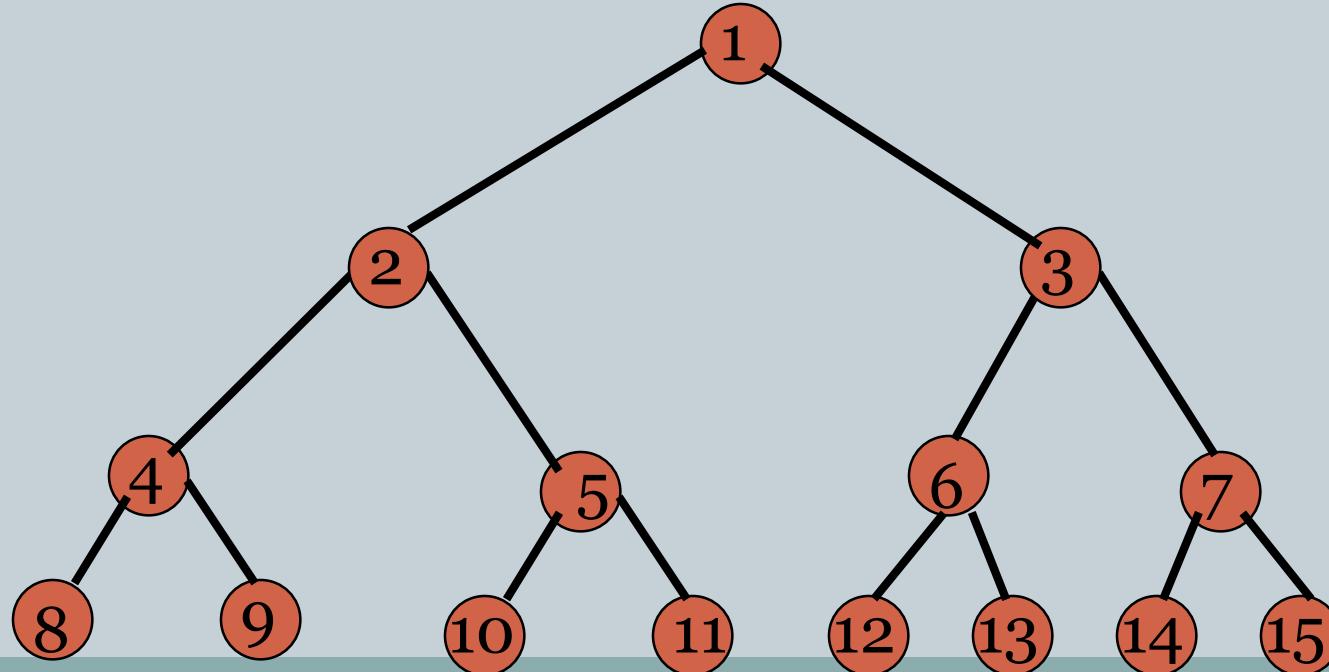
tree[] [ a | - | b | - | - | - | c | - | - | - | - | - | d ]  
0 5 10 15

- To represent a binary tree of depth 'h' using array representation, we need one dimensional array with a maximum size of  $2^{h+1} - 1$ .

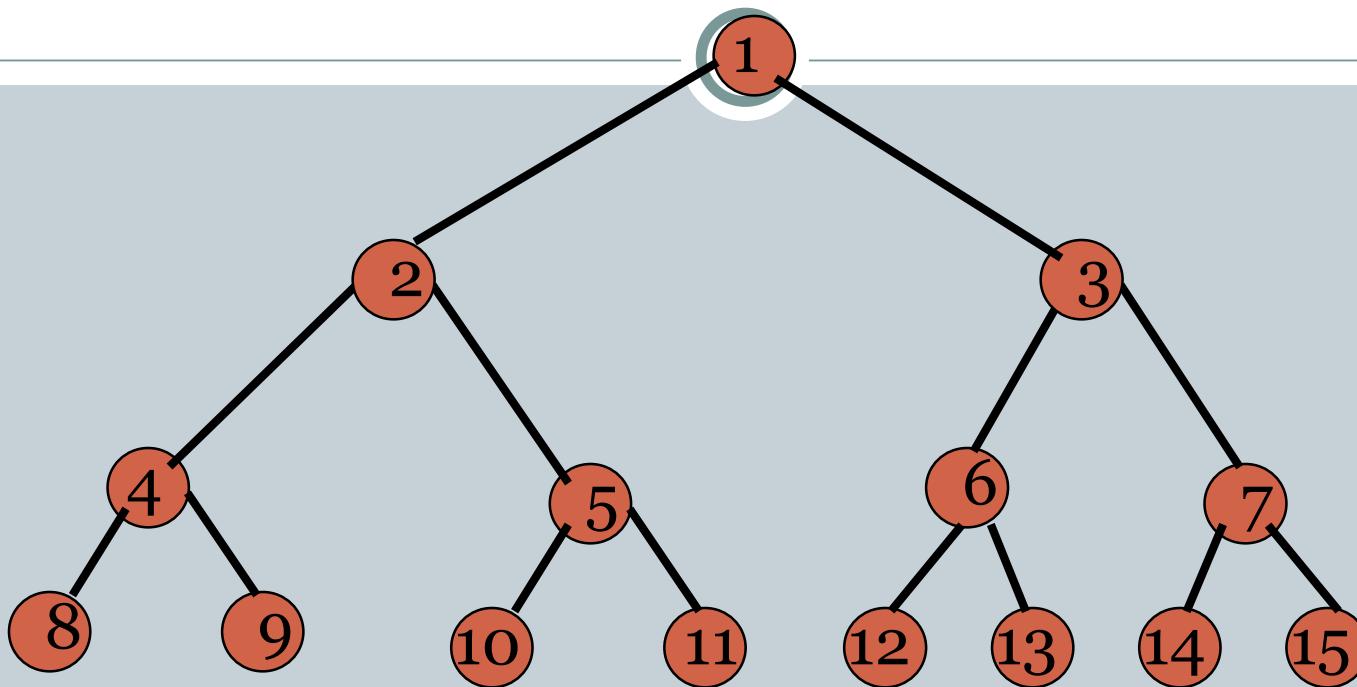
# Numbering Nodes In A Binary Tree



- Number the nodes 1 through  $2^{h+1} - 1$ .
- Number by levels from top to bottom.
- Within a level number from left to right.



# Node Number Properties

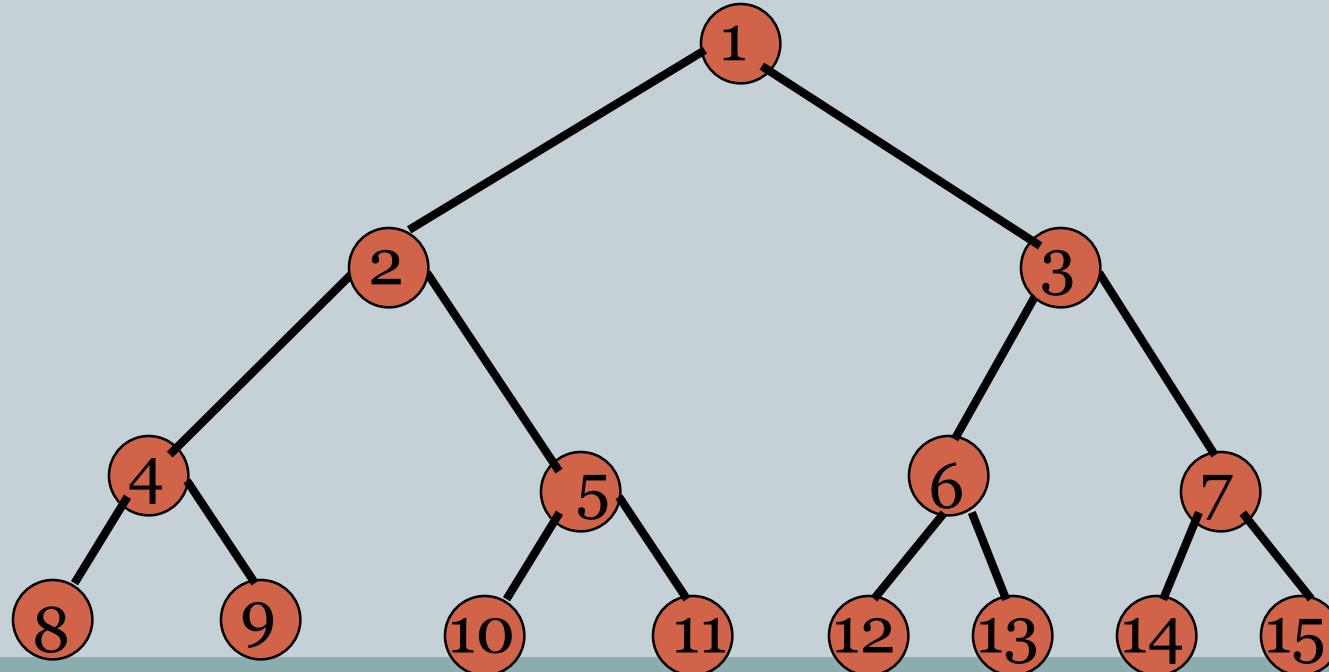


- Parent of node  $i$  is node  $i / 2$ , unless  $i = 1$ .
- Node  $1$  is the root and has no parent.

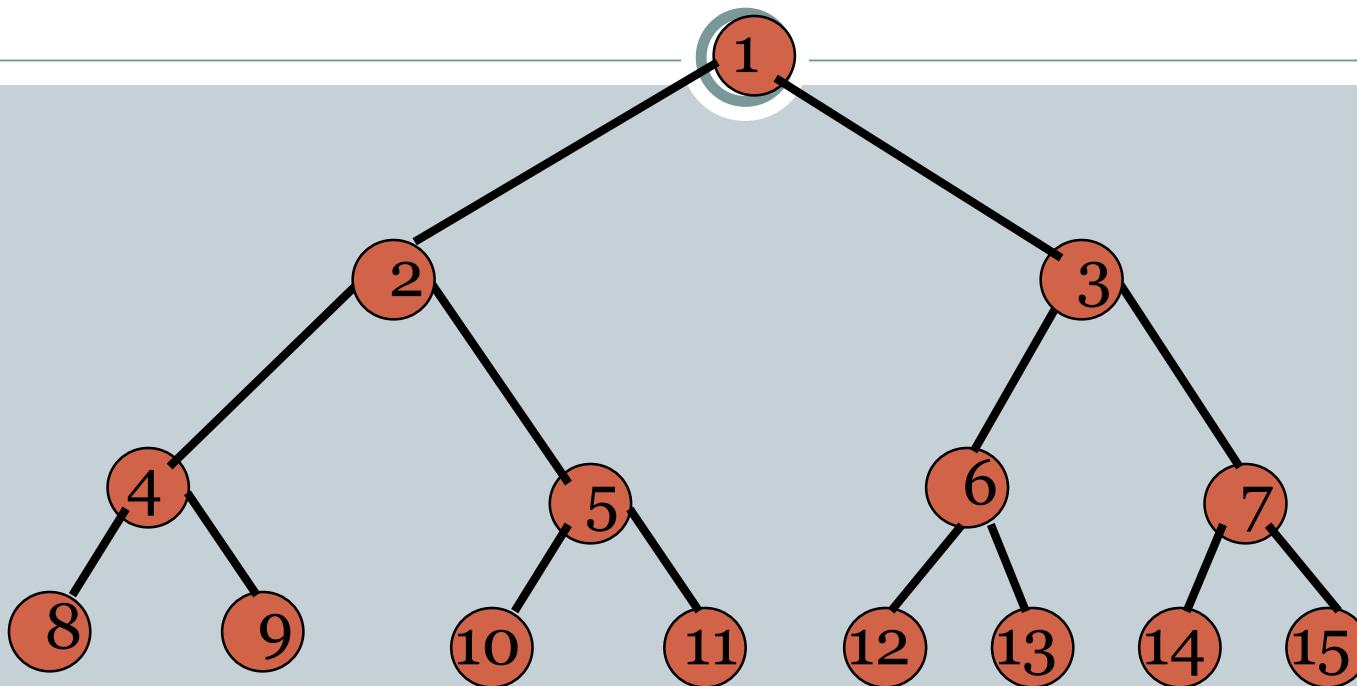
# Numbering Nodes In A Full Binary Tree



- Number the nodes 1 through  $2^{h+1} - 1$ .
- Number by levels from top to bottom.
- Within a level number from left to right.

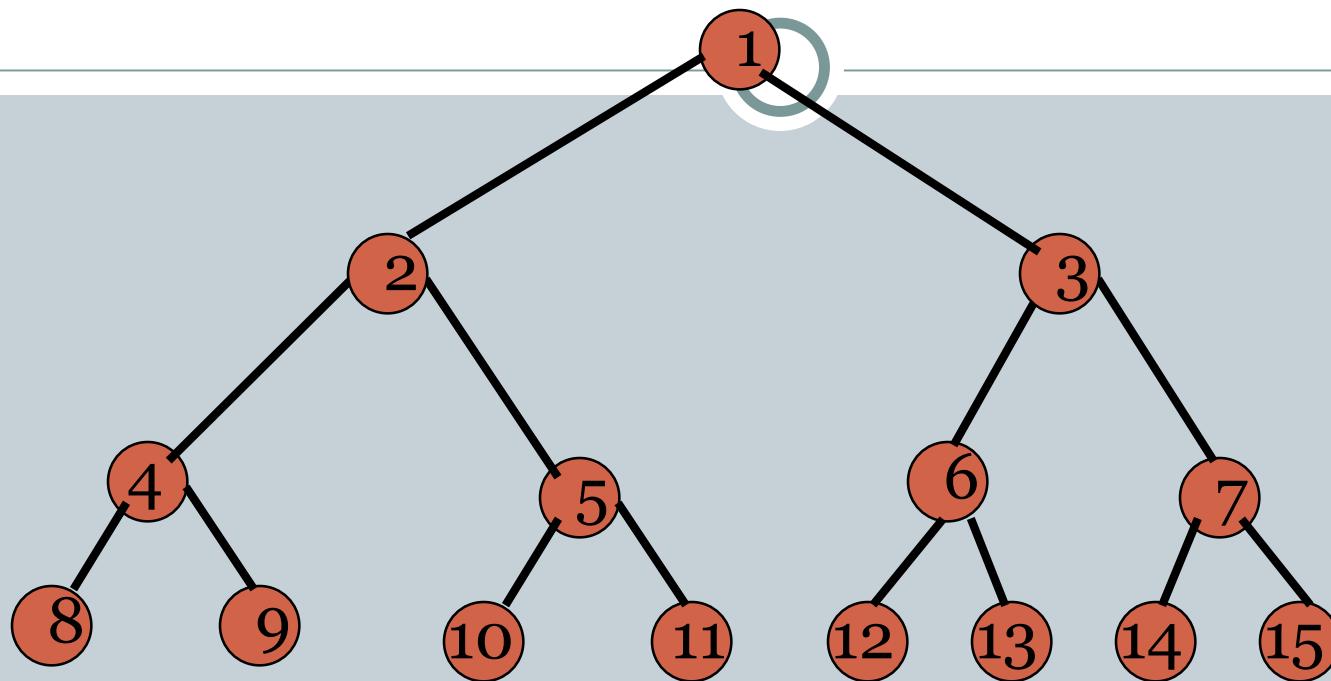


# Node Number Properties



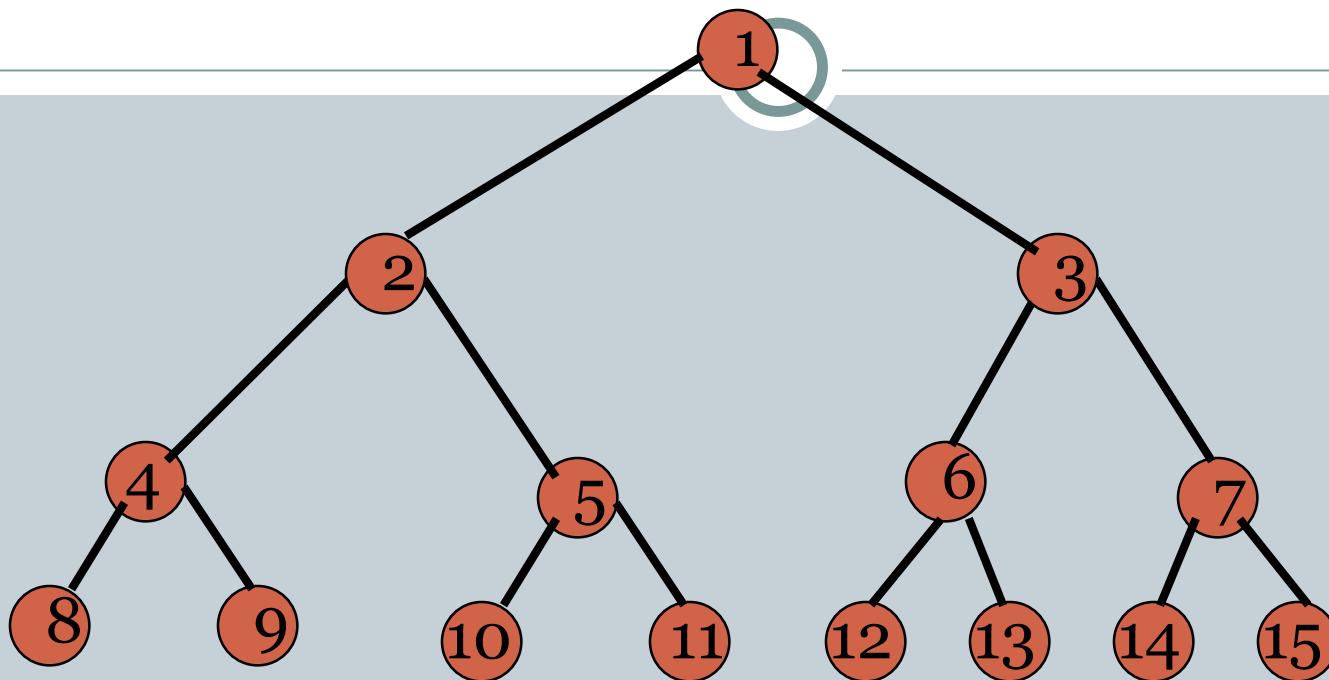
- Parent of node  $i$  is node  $i / 2$ , unless  $i = 1$ .
- Node  $1$  is the root and has no parent.

# Node Number Properties



- Left child of node  $i$  is node  $2i$ , unless  $2i > n$ , where  $n$  is the number of nodes.
- If  $2i > n$ , node  $i$  has no left child.

# Node Number Properties



- Right child of node  $i$  is node  $2i+1$ , unless  $2i+1 > n$ , where  $n$  is the number of nodes.
- If  $2i+1 > n$ , node  $i$  has no right child.

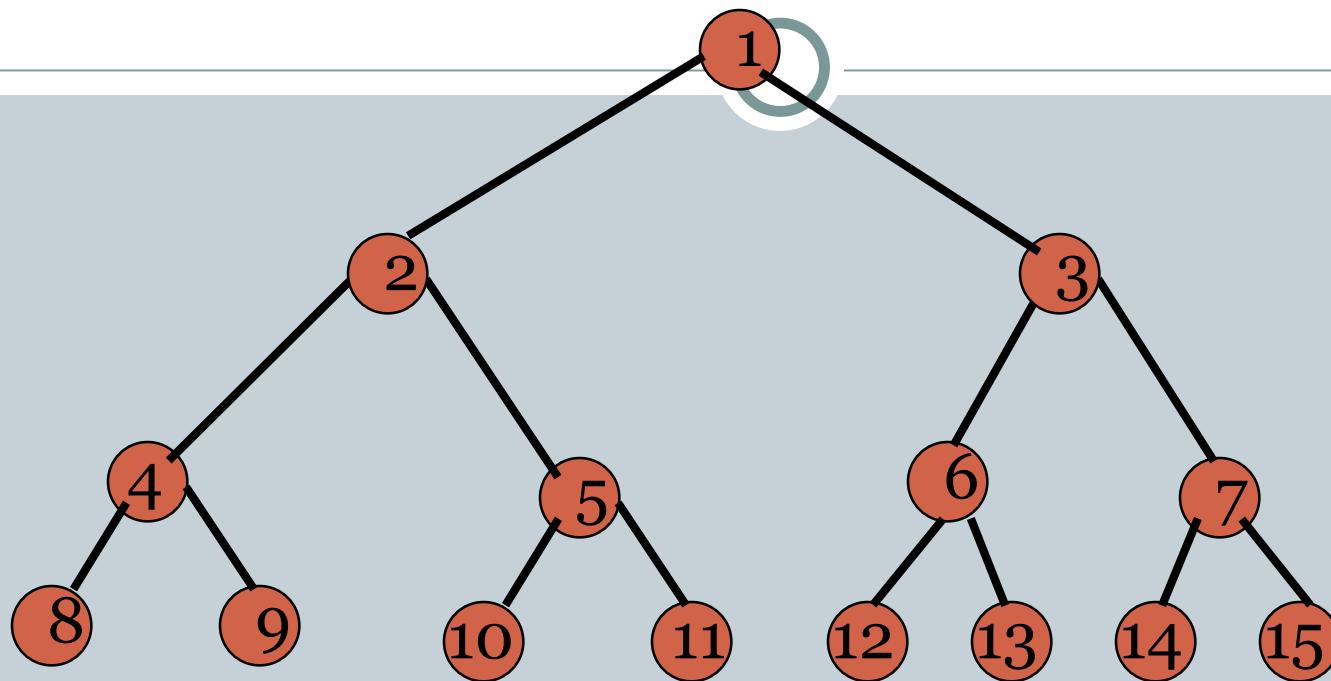
When node  $i$  has a right child, then

Root-to-right-child path length =  $i$

Right child =  $2i+1$

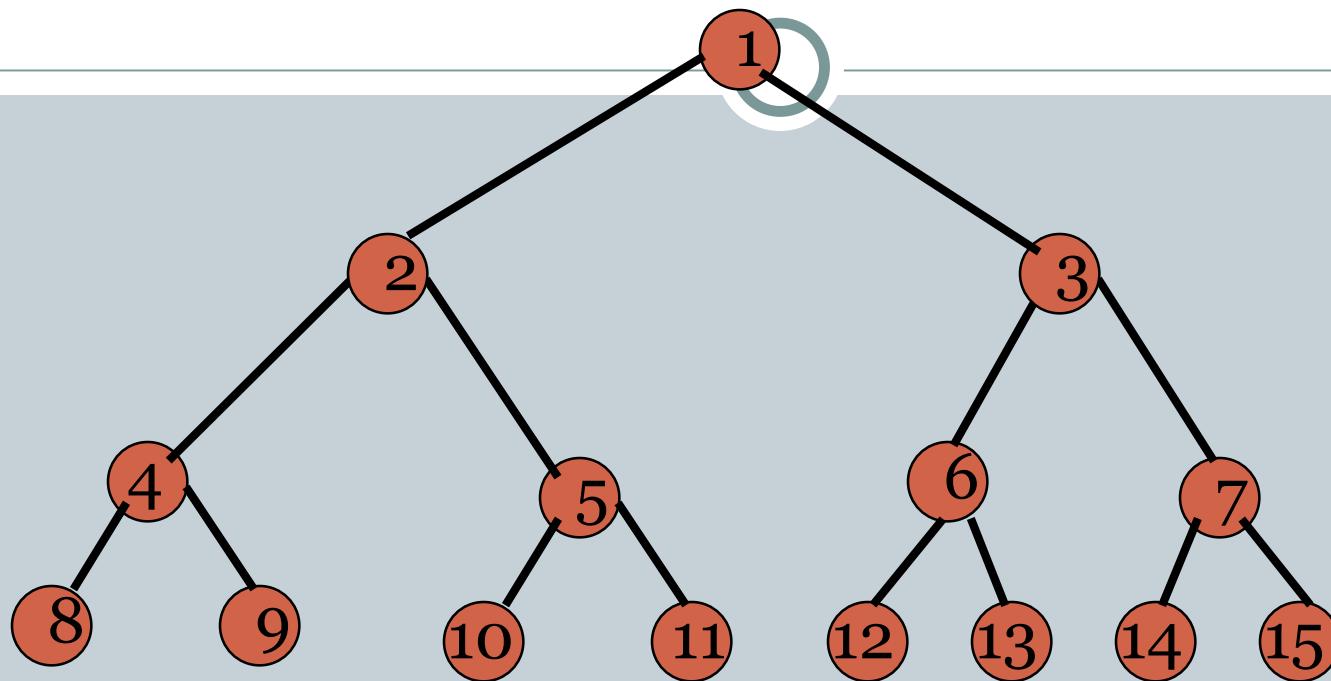
- a)  $i-2$
- b)  $i/2$
- c)  $i+2$
- d)  $2i+1$

# Node Number Properties



- Left child of node  $i$  is node  $2i$ , unless  $2i > n$ , where  $n$  is the number of nodes.
- If  $2i > n$ , node  $i$  has no left child.

# Node Number Properties



- Right child of node  $i$  is node  $2i+1$ , unless  $2i+1 > n$ , where  $n$  is the number of nodes.
- If  $2i+1 > n$ , node  $i$  has no right child.

# Linked Representation



- Each binary tree node is represented as an object whose data type is **BinaryTreeNode**.
- The space required by an **n** node binary tree is **n**\* (space required by one node).

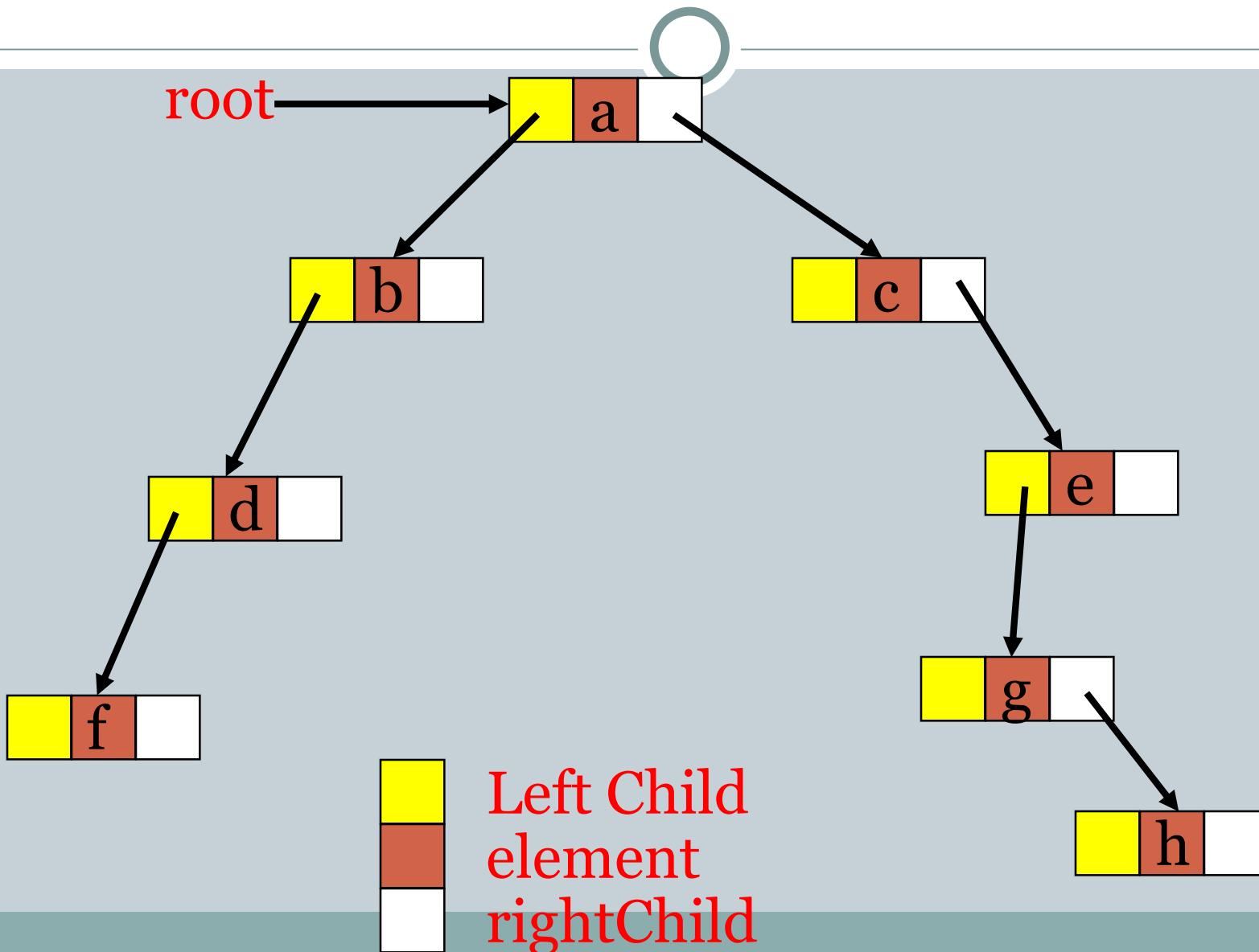
# The Class BinaryTreeNode



## Class BinaryTreeNode

```
{  
<type of data> Object element;  
BinaryTreeNode *leftChild; // left subtree  
BinaryTreeNode *rightChild; // right subtree  
// constructors and any other methods  
// come here  
};
```

# Linked Representation Example



# NEW TOPIC (UNIT1:Trees)



- Syllabus
- Difference between linear and non-liner data structure
- Trees and Binary trees - basic terminology, representation using linked organization
- Binary tree- properties, converting tree to binary tree
- Binary tree traversals recursive and non-recursive
- Level wise Depth first and Breadth first.
- Binary Search Tree (BST), BST operations

# Complete Binary Tree With n Nodes

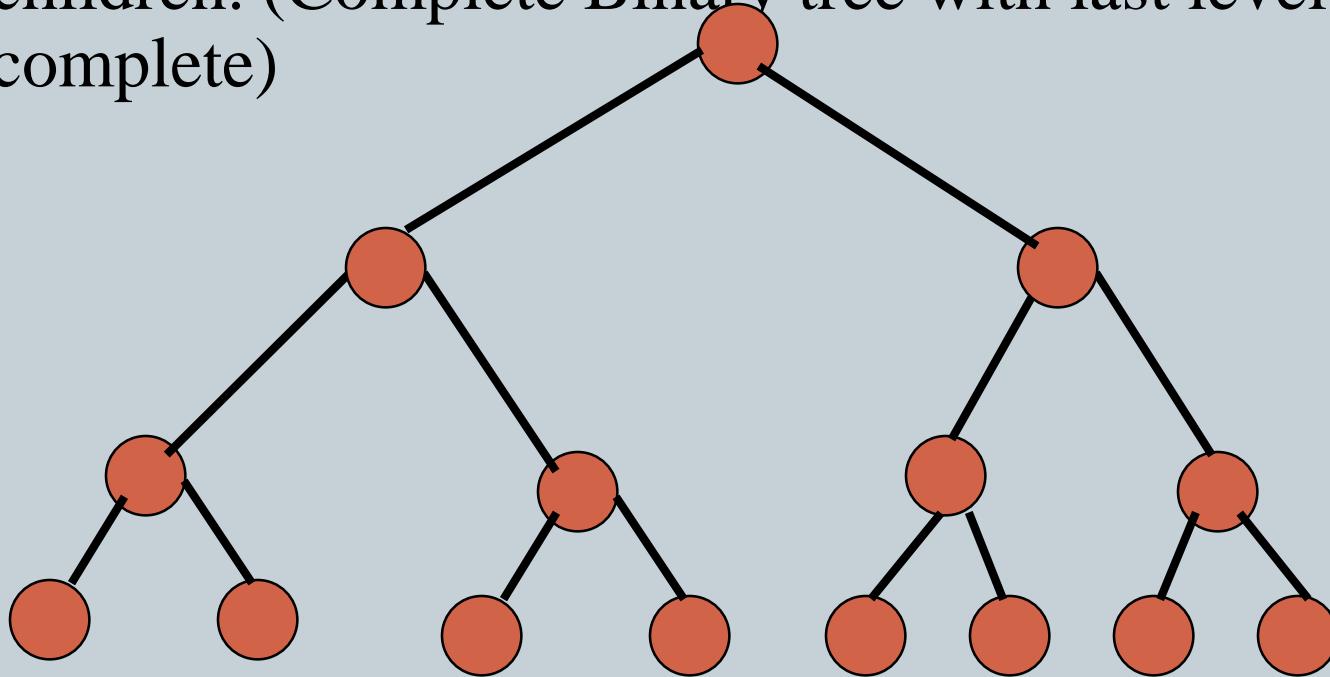
A **complete** binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

A binary tree T is **full** If each node is either a **leaf or possesses exactly two child nodes.**

# Perfect Binary Tree



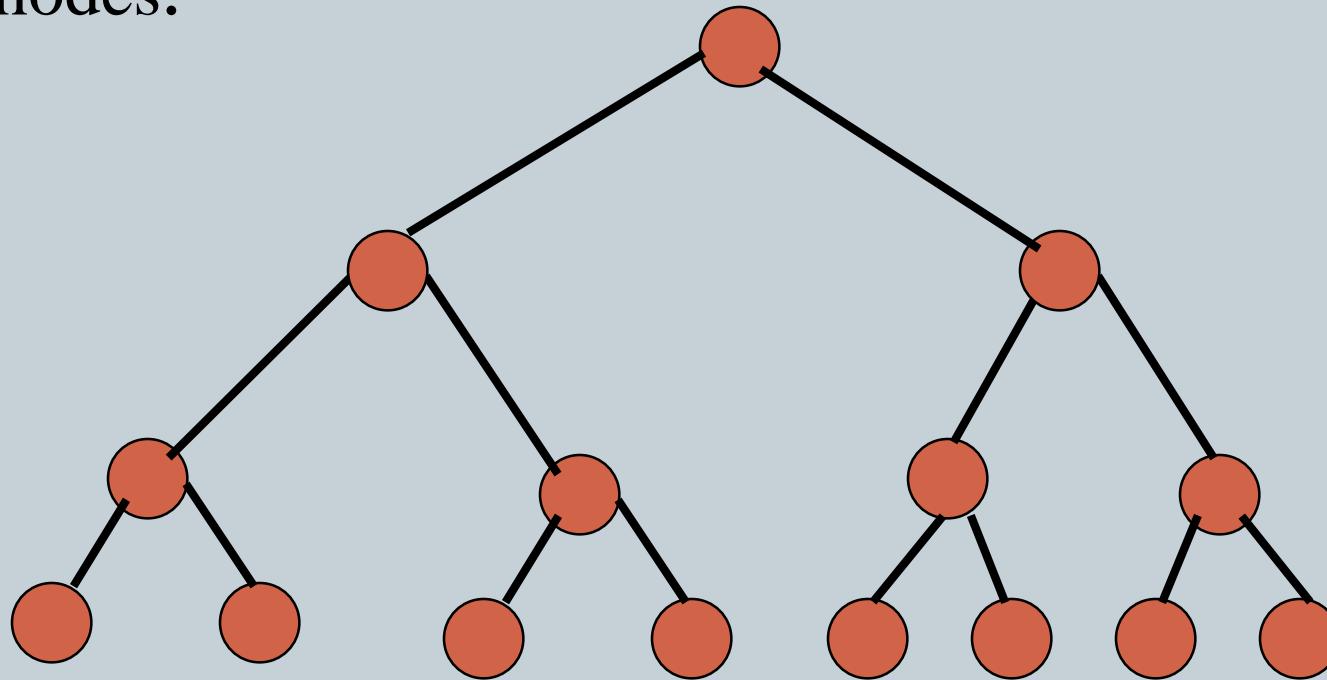
- A perfect binary tree has all internal nodes having 2 children. (Complete Binary tree with last level also complete)



# Perfect Binary Tree



- A perfect binary tree of a given height  $h$  has  $2^{h+1} - 1$  nodes.

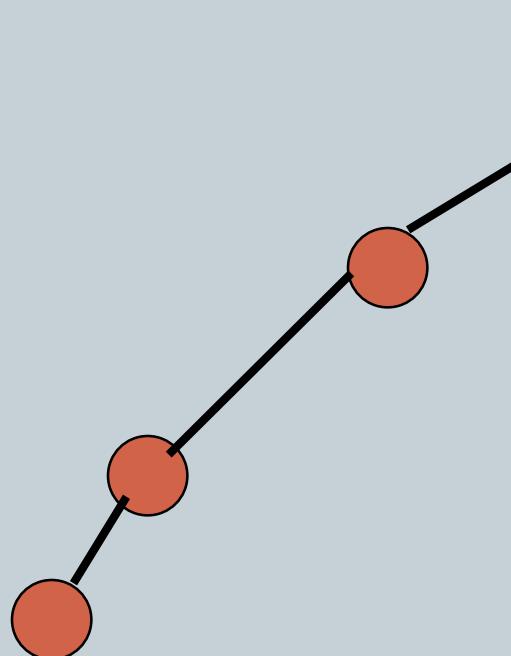


Q) Height=3 What are the MAX number of nodes?  
(Here **a leaf is considered at height 0**)

# Minimum Number Of Nodes



- Minimum number of nodes in a **binary tree** whose height is  $h$ .
- At least one node at each of first  $h$  levels.
- (Here **a leaf is considered at height  $\underline{0}$** )

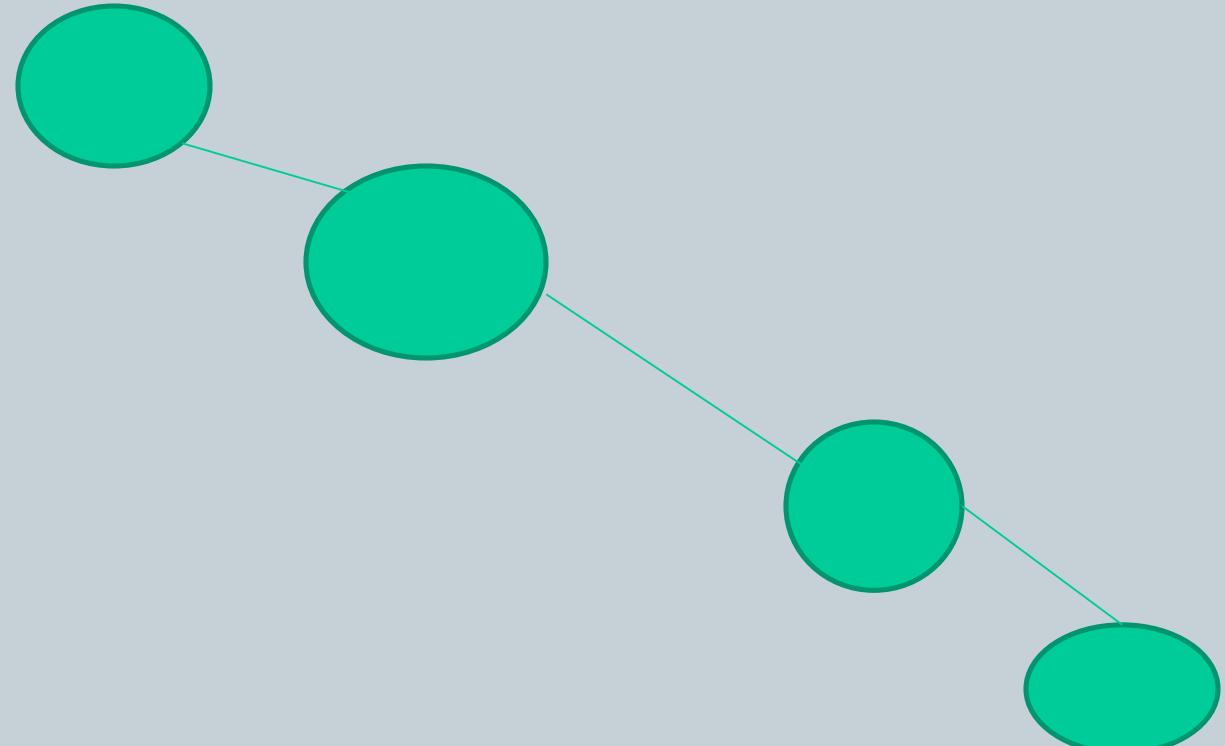


minimum number of nodes is  
 $h+1$  where  $h$  is height

Q)



- What is the **Minimum** number of nodes in a binary tree whose height is **3**?



Q)

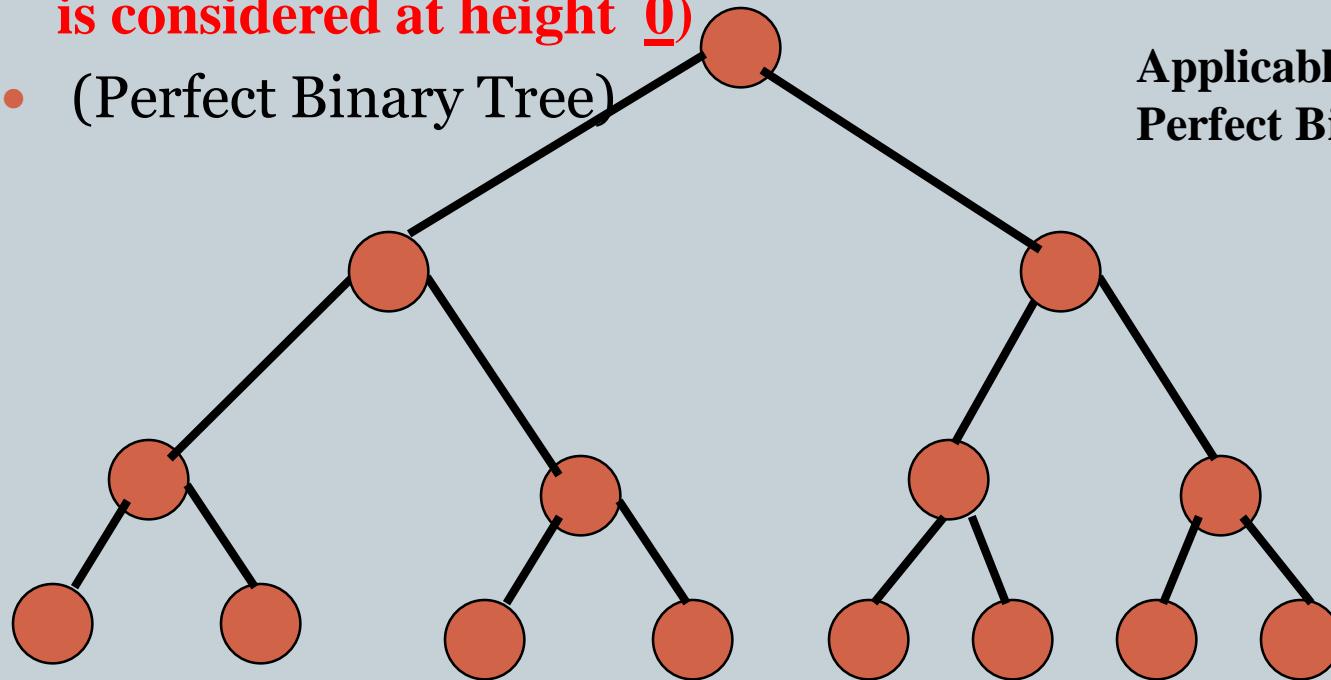


- What is the minimum number of nodes in a tree of height 7?
- 7?
- 8?
- 6?
- $2^7$ ?

# Maximum Number Of Nodes

- All possible nodes at first **h** levels are present (Here **a leaf is considered at height 0**)
- (Perfect Binary Tree)

Applicable for  
Perfect Binary tree

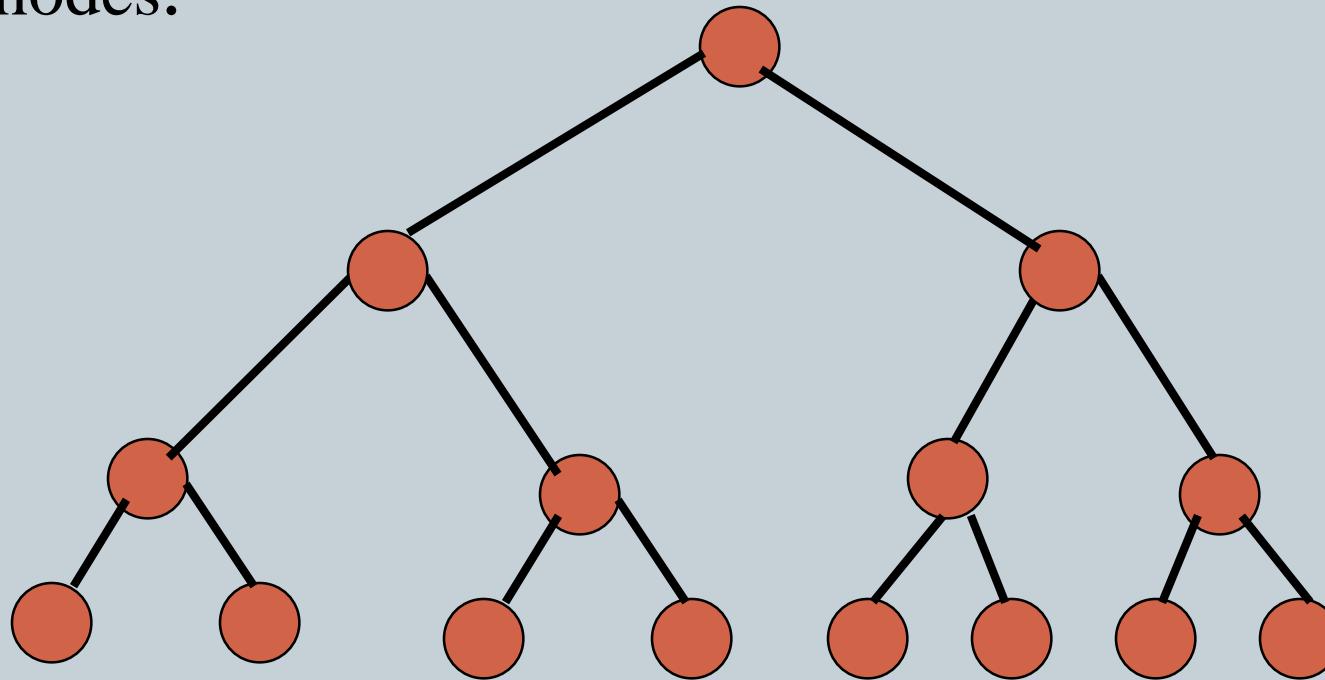


Maximum number of nodes  
$$2^0 + 2^1 + \dots + 2^h = 2^{h+1} - 1$$
$$= 2^{h+1} - 1$$

# Perfect Binary Tree



- A perfect binary tree of a given height  $h$  has  $2^{h+1} - 1$  nodes.



Q) Height=3 What are the MAX number of nodes?

(Here **a leaf is considered at height 0**)



Q) Height of a Binary tree is =4  
What are the MAX number of nodes possible?

- MAX Number of nodes in any binary tree (with respect to heights of its left and right subtree)

(Here **a leaf is considered at height 0**)



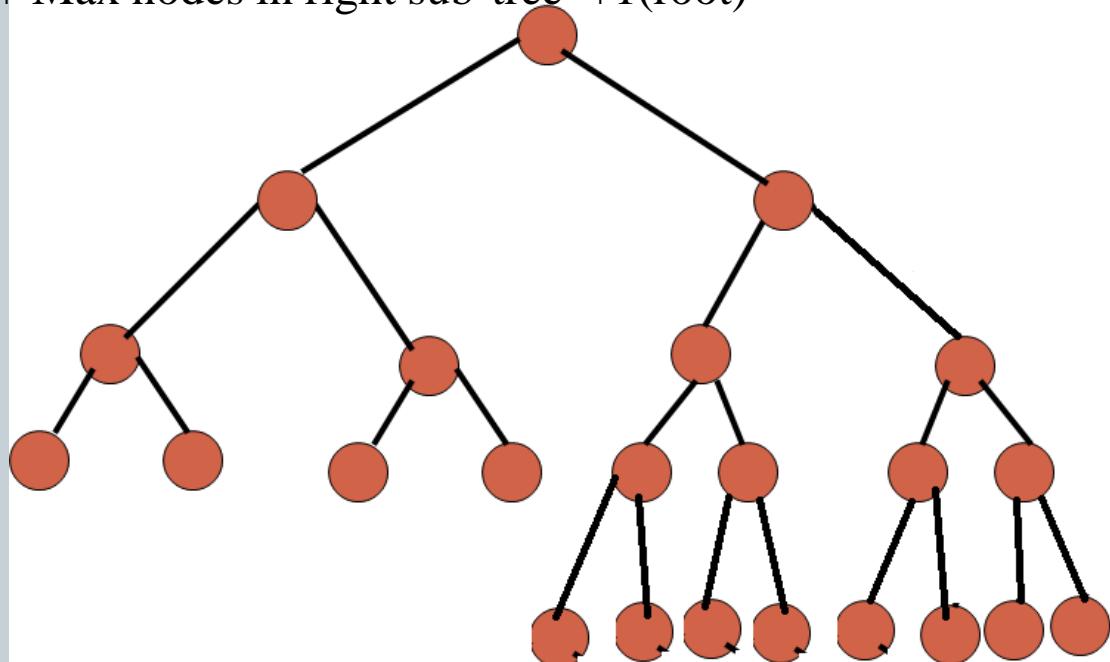
- $lh$ =height of left subtree
- $rh$ =height of right subtree :
- MAX number of nodes possible =  $(2^{lh+1} - 1) + (2^{rh+1} - 1) + 1$
- i.e. Max nodes in left sub-tree + Max nodes in right sub-tree +1(root)

Q)

- $lh$ =height of left subtree = 2
- $rh$ =height of right subtree = 3

What is MAX number of nodes possible?

- $(2^{lh+1} - 1) + (2^{rh+1} - 1) + 1$

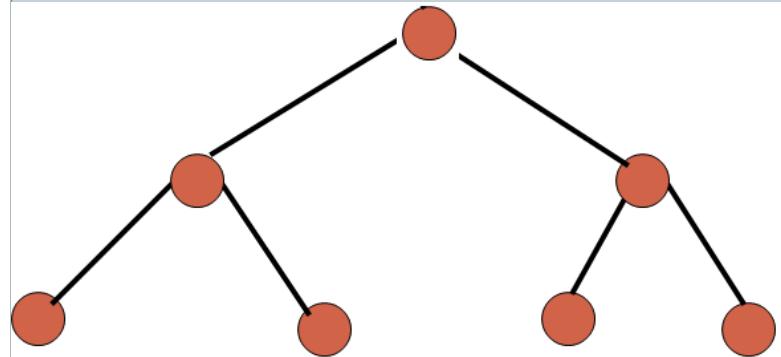


# Number Of Nodes( $n$ ) & Height( $h$ )

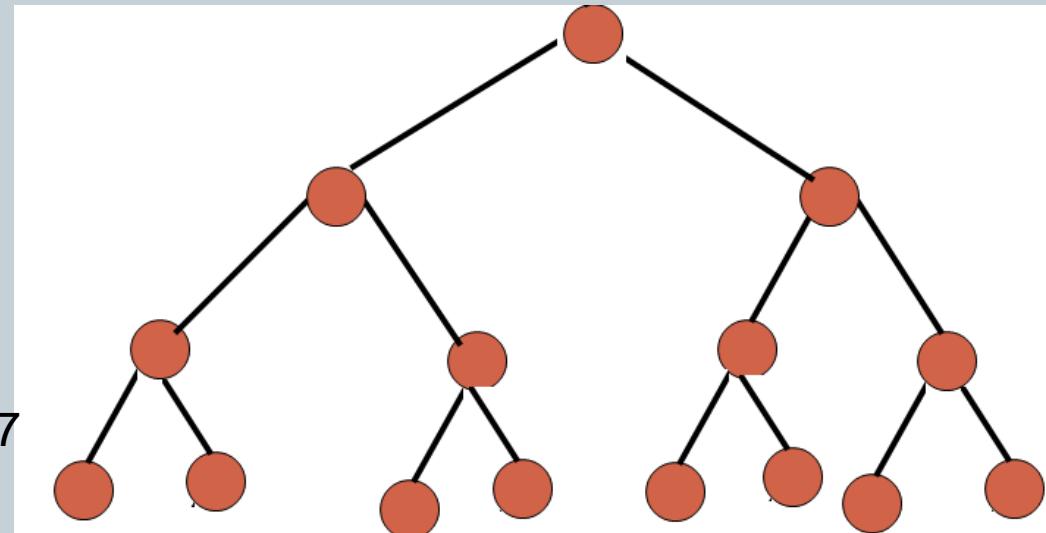


*Let  $n$  be the number of nodes in a complete binary tree whose height is  $h$*

- The **number** of nodes (nearly) **doubles** every time the **depth** increases by 1



3 internal nodes + 4 external nodes = 7



7 internal nodes + 8 external nodes = 15

# Number Of Nodes(n) & Height(h)



Let  $n$  be the number of nodes in a binary tree whose height is  $h$  (Here a leaf is considered at height 0)

**Number of nodes  $n$  will be between the min and max number of nodes : i.e**

$$\text{Min} = (h+1) \quad \text{and} \quad \text{Max} = 2^{h+1} - 1$$

$$(h+1) \leq n \leq 2^{h+1} - 1$$

OR

$$h < n < 2^{h+1}$$

- Ex. ( $h = 3$ )               $3 < n < 2^4(16)$

# Number Of Nodes(n) & Height(h)



Let  $n$  be the number of nodes in a binary tree whose height is  $h$  (Here **a leaf is considered at height 1**)

**Number of nodes  $n$  will be between the min and max number of nodes : i.e**

$$\text{Min}=(h) \text{ and } \text{Max}=2^h - 1$$

$$(h) \leq n \leq 2^h - 1$$

OR

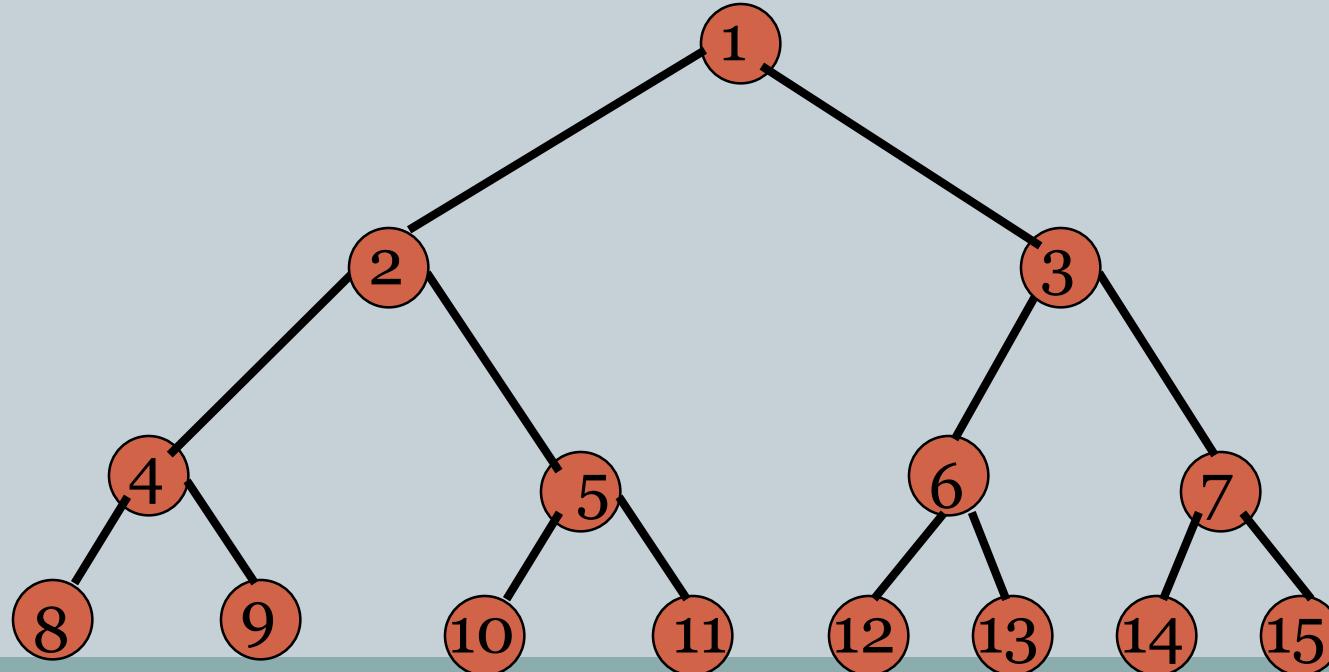
$$h < n < 2^h$$

- Ex. ( $h = 4$ )       $4 < n < 2^4(16)$

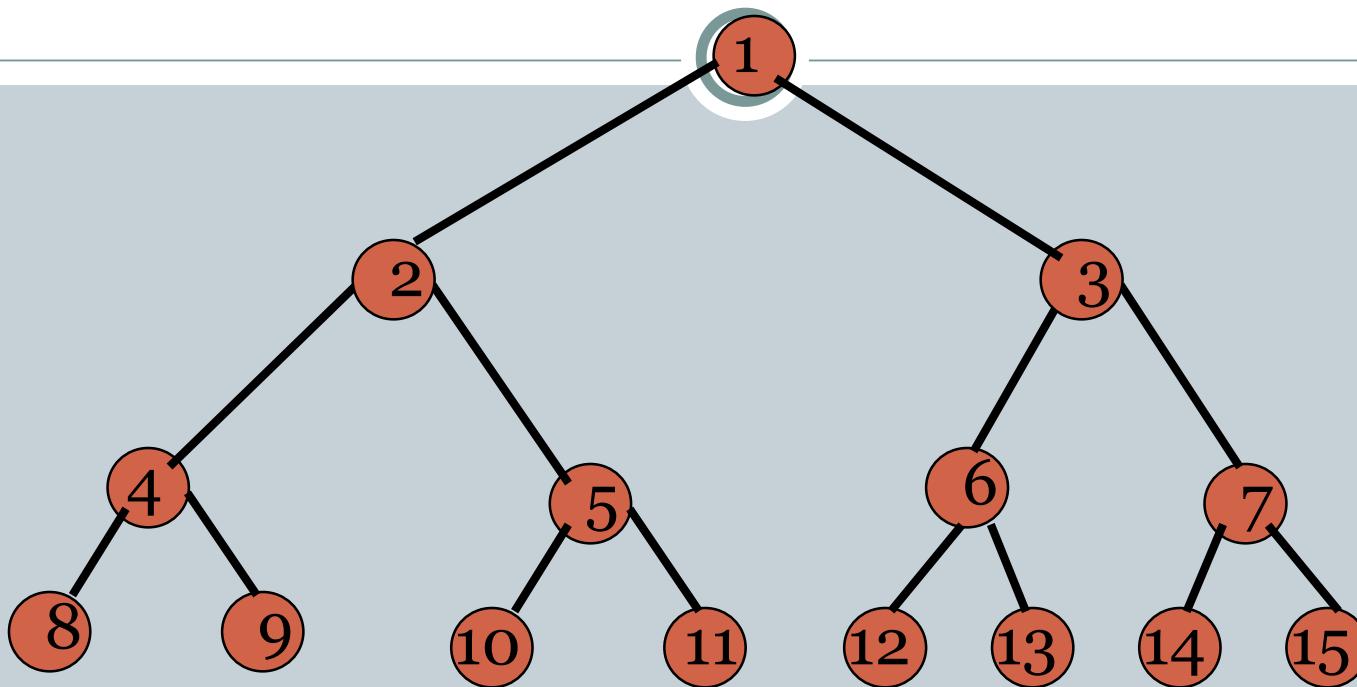
# Numbering Nodes In A Full Binary Tree



- Number the nodes 1 through  $2^{h+1} - 1$ .
- Number by levels from top to bottom.
- Within a level number from left to right.

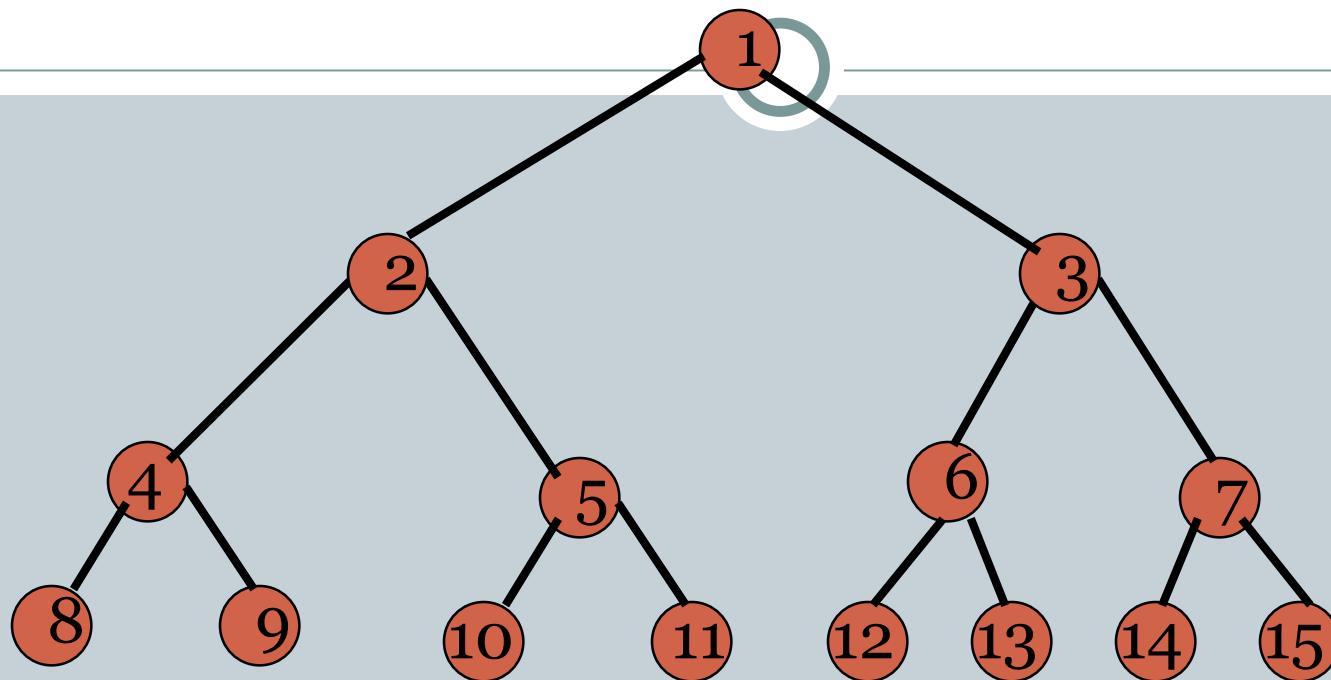


# Node Number Properties



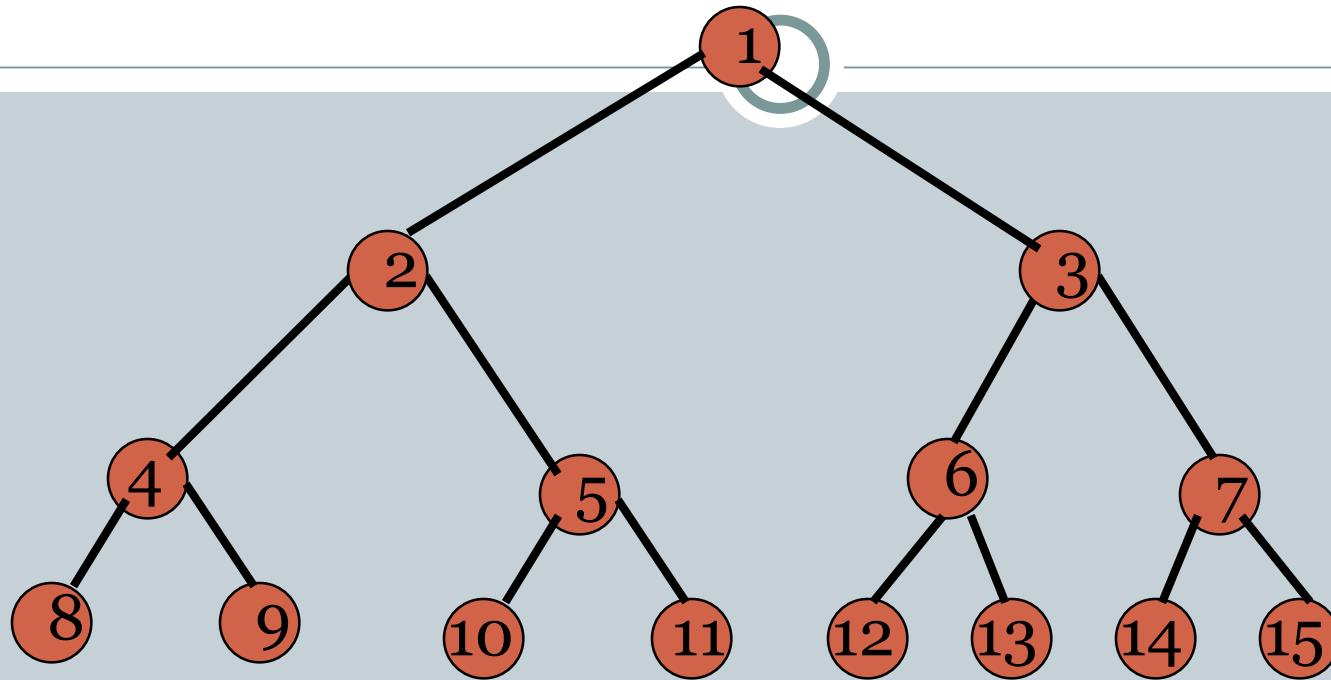
- Parent of node  $i$  is node  $i / 2$ , unless  $i = 1$ .
- Node  $1$  is the root and has no parent.

# Node Number Properties



- Left child of node  $i$  is node  $2i$ , unless  $2i > n$ , where  $n$  is the number of nodes.
- If  $2i > n$ , node  $i$  has no left child.

# Node Number Properties

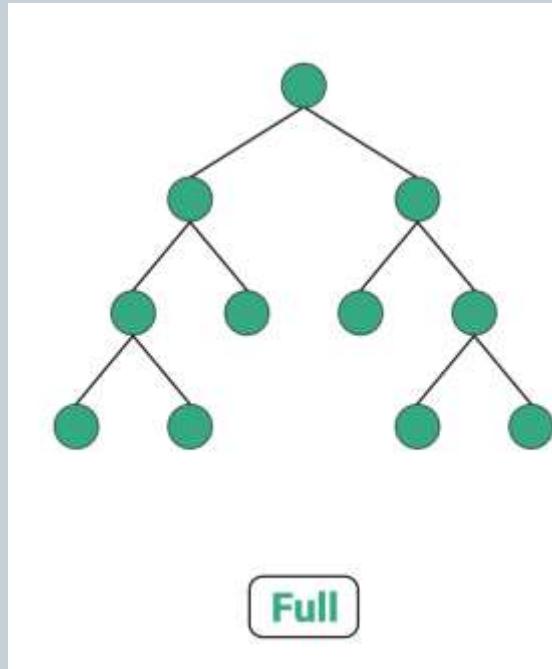


- Right child of node  $i$  is node  $2i+1$ , unless  $2i+1 > n$ , where  $n$  is the number of nodes.
- If  $2i+1 > n$ , node  $i$  has no right child.

# Full binary tree (2-tree)



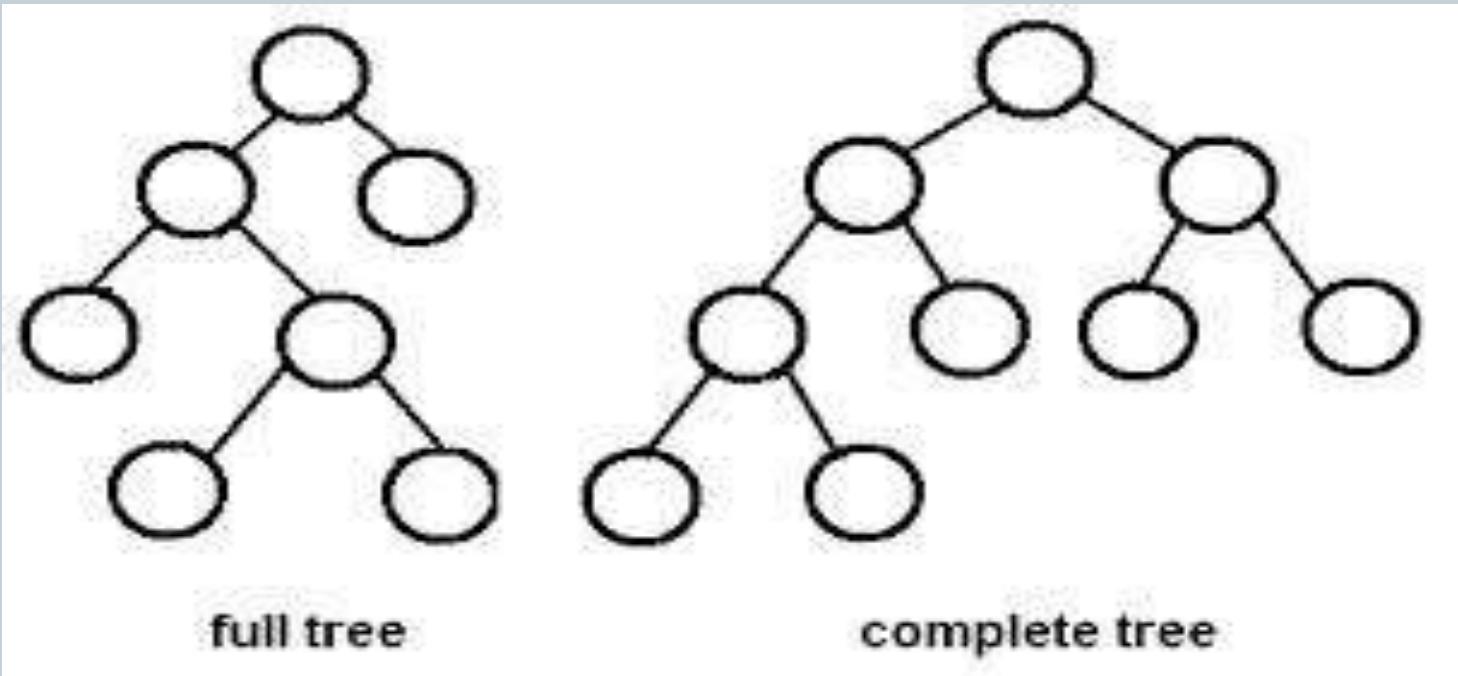
- A binary tree where every internal node has exactly two children (so every node has 0/2 children).



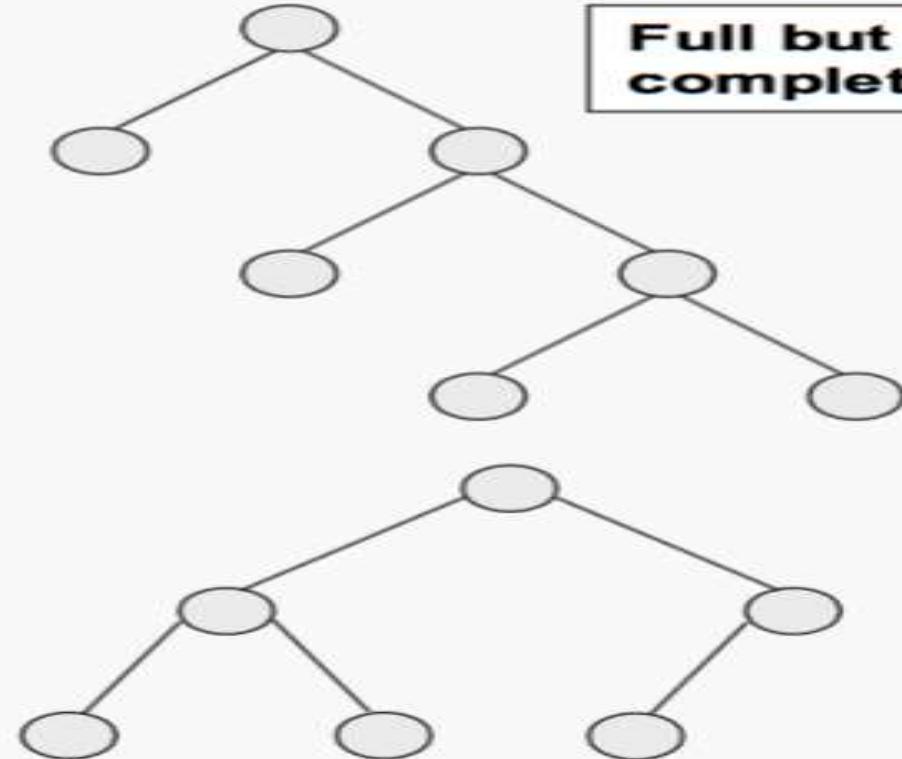
## Complete Binary Tree



- A binary tree  $T$  with  $n$  levels is Complete if all levels except possibly the last are completely full, and the last level has all its nodes to the left side.



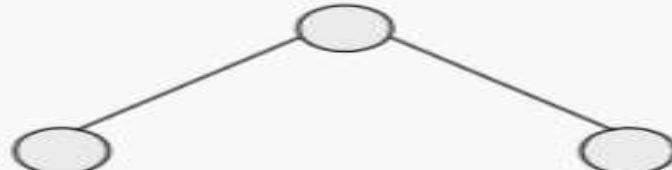
# Examples of Full and Complete Binary Tree



**Full but not complete.**



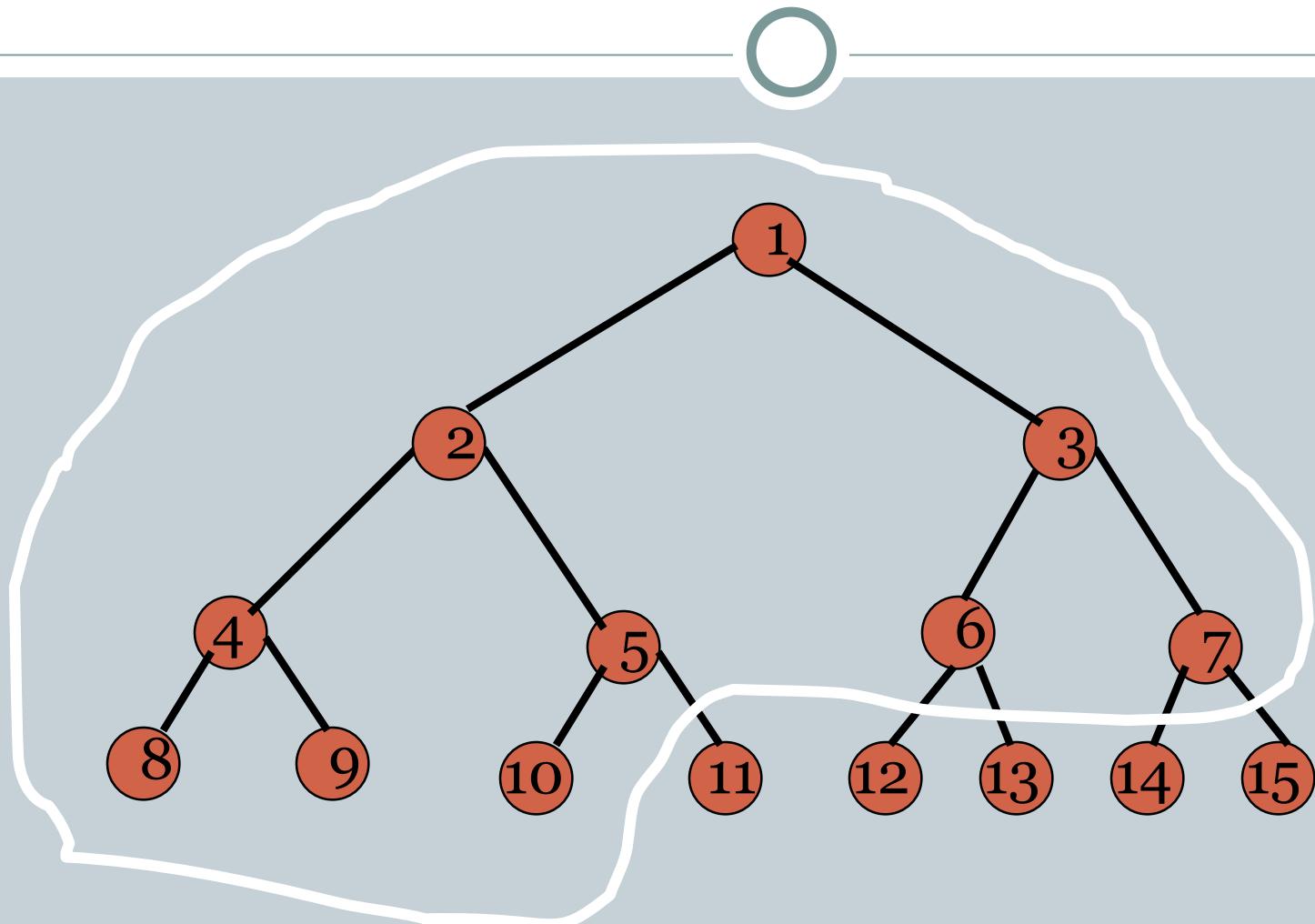
**Neither complete nor full.**



**Full and complete.**

**Complete but not full.**

# Example

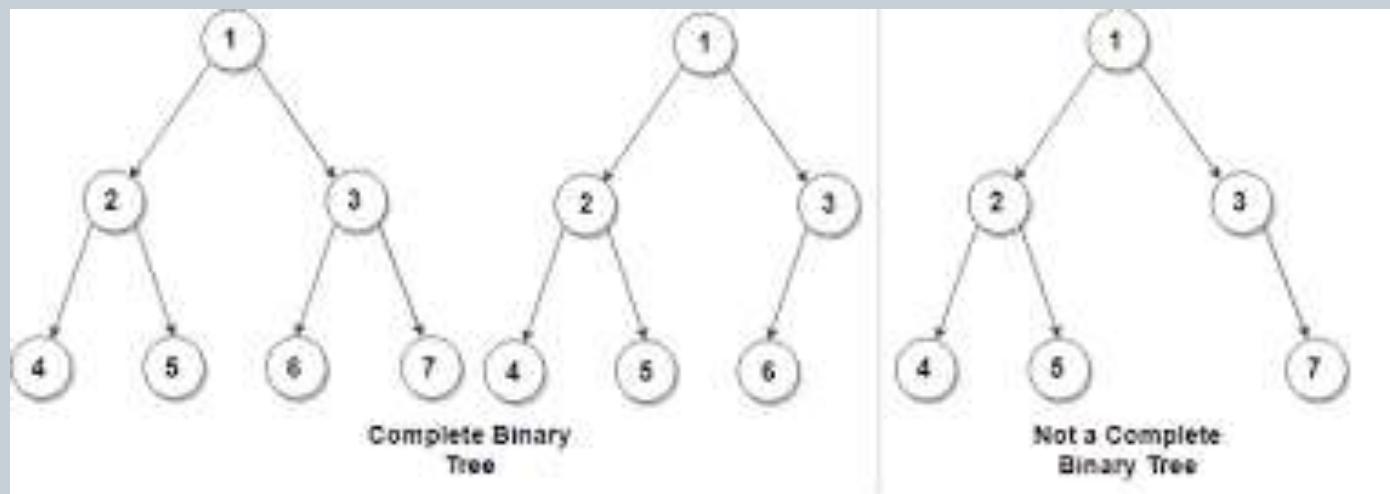


- Complete binary tree with **10** nodes.

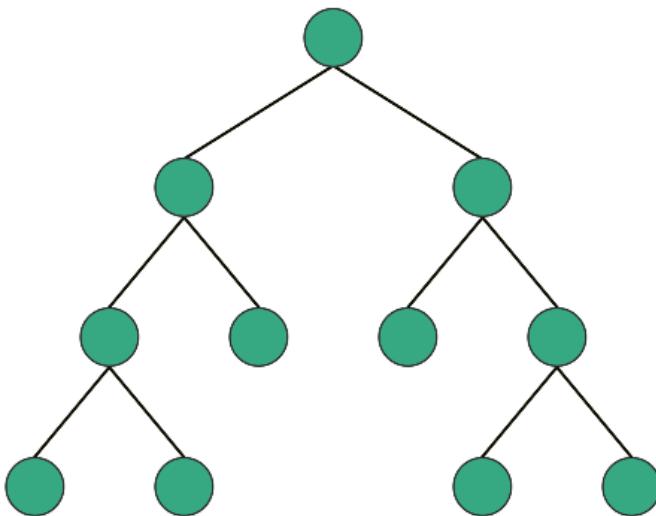
# Complete Binary Tree



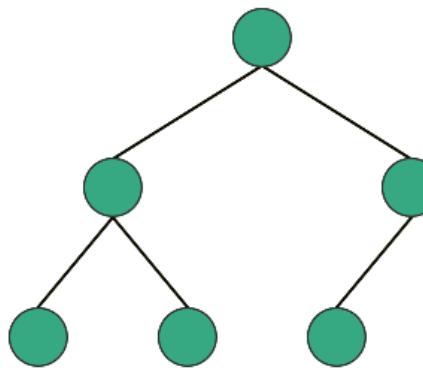
- A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right



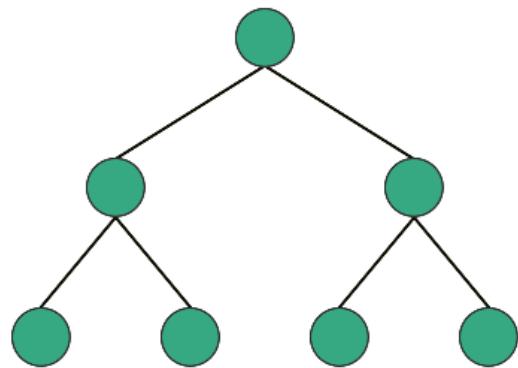
# Full, Complete and Perfect binary trees



Full



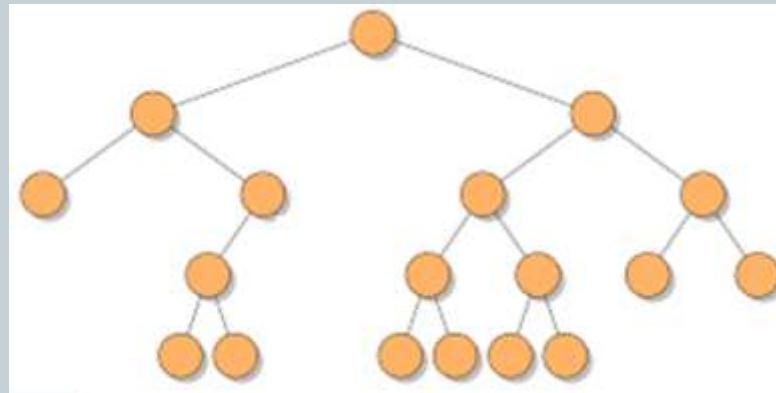
Complete



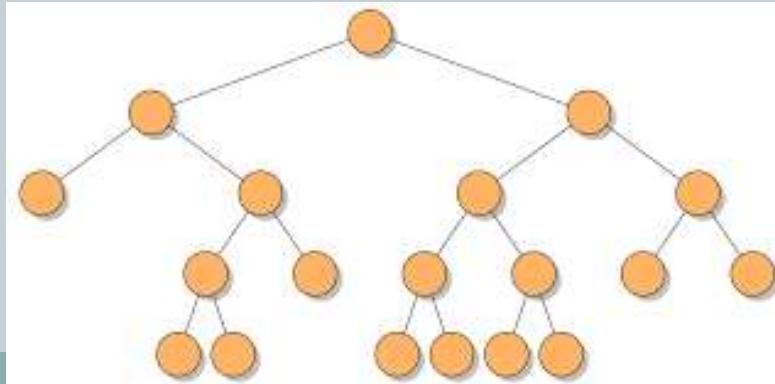
Perfect

# Which is a FULL binary tree?

A or B or Both or Neither



A



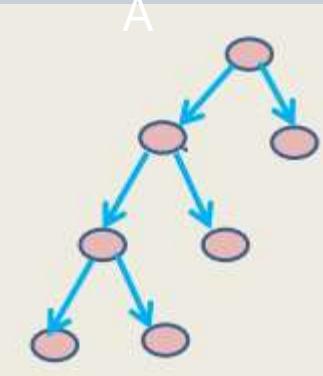
B

# Label the images A,B,C of Binary Trees with

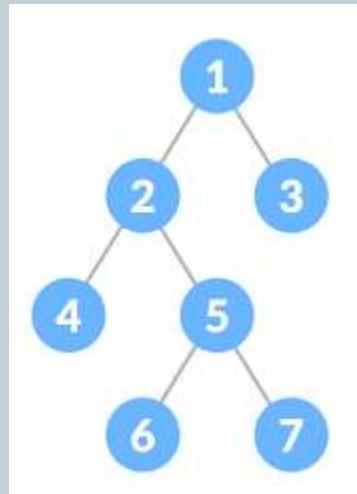
- Full / Complete



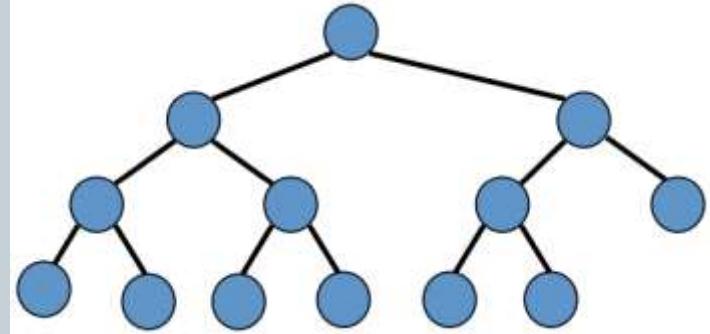
A



B



C

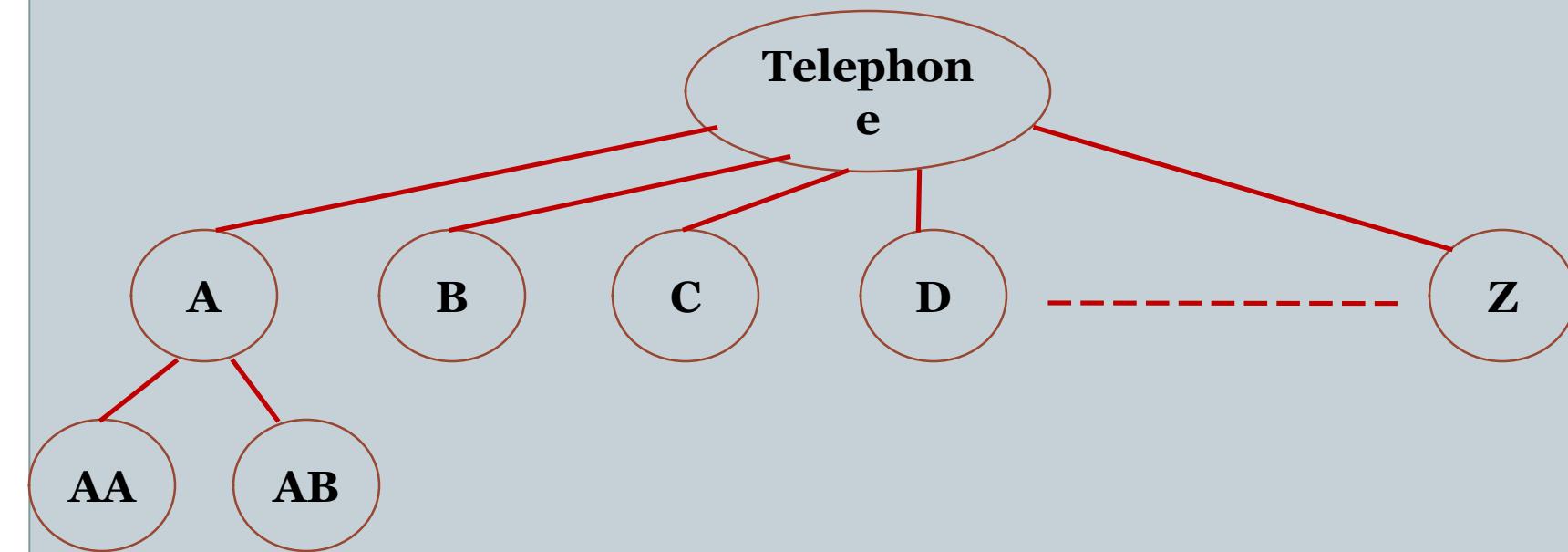




# Convert a General tree to Binary Tree

# General Tree

- A tree with unlimited out degree



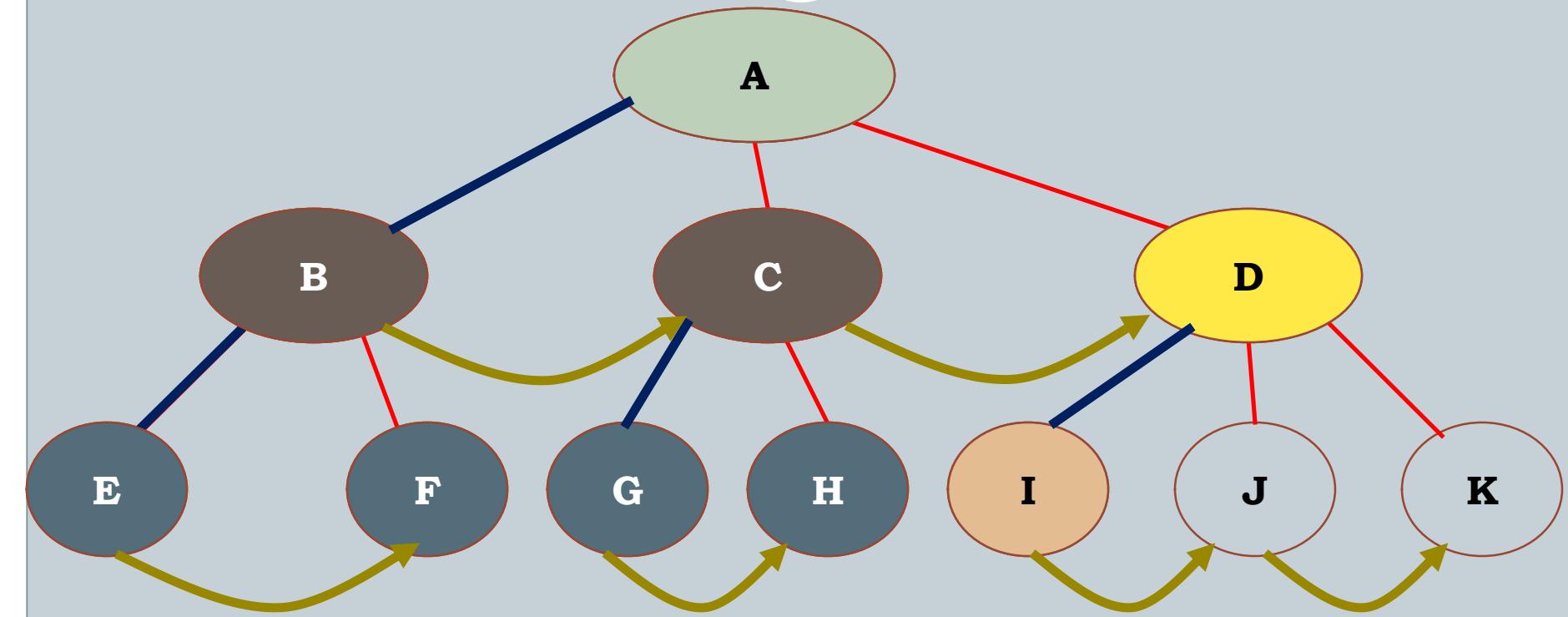
# Algorithm to convert General Tree to Binary

---

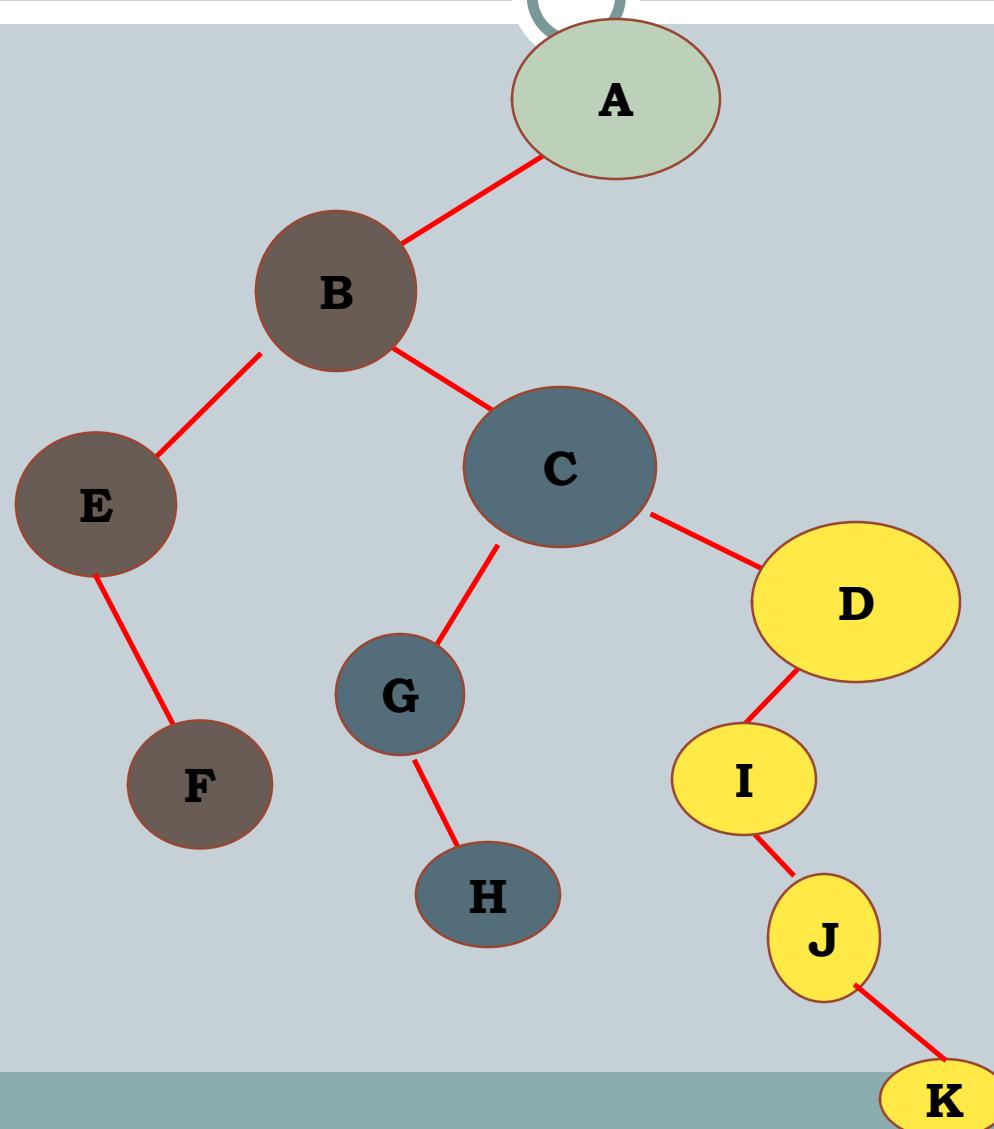


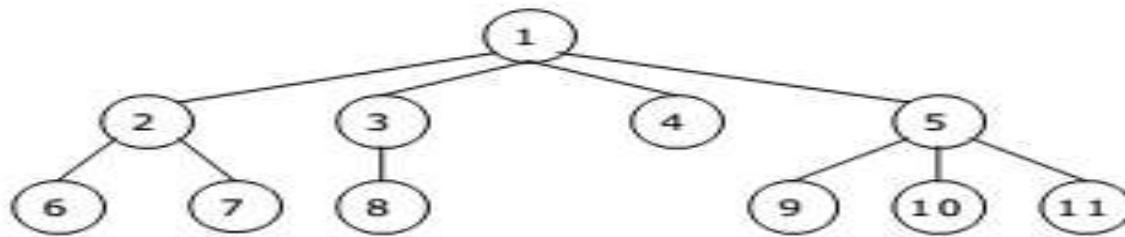
- Identify branches from parent to left most child
- Connect remaining siblings from left most child using branches for each sibling to its right
- Delete all unneeded branches

# Example Tree Conversion General to Binary



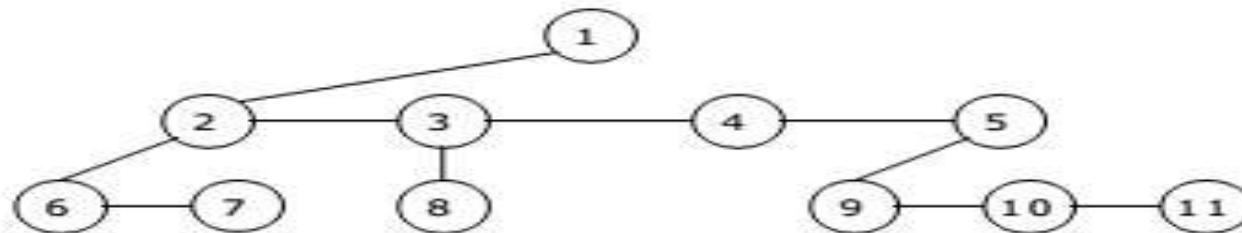
# Example Tree Conversion General to Binary



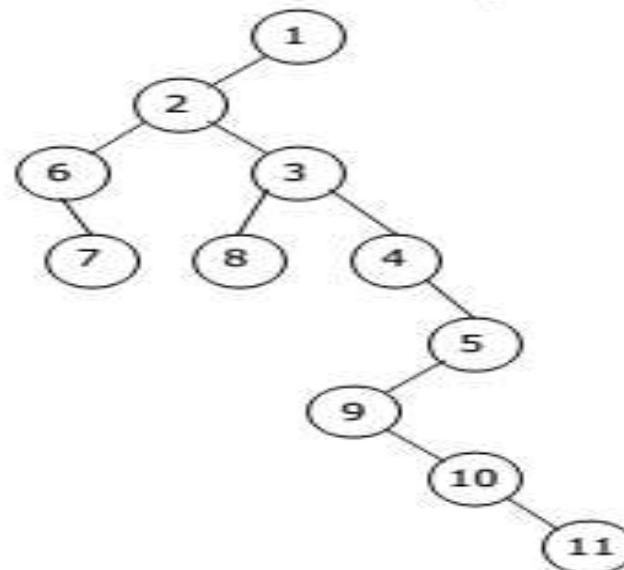


**Solution:**

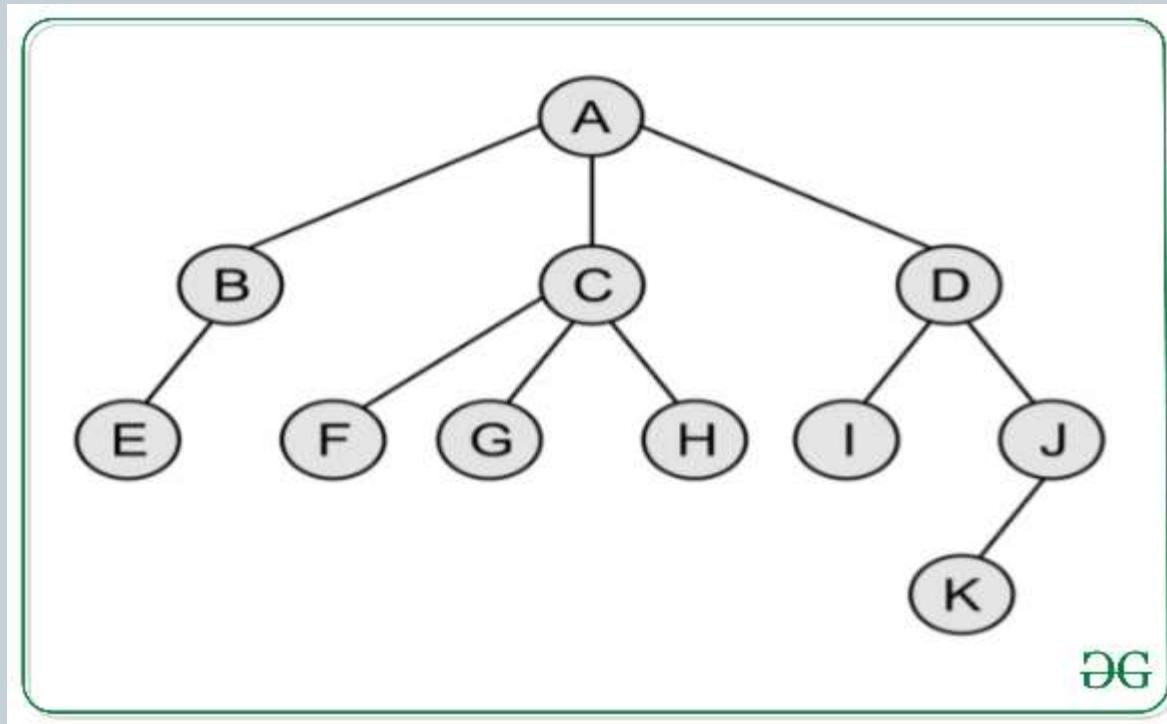
Stage 1 tree by using the above mentioned procedure is as follows:



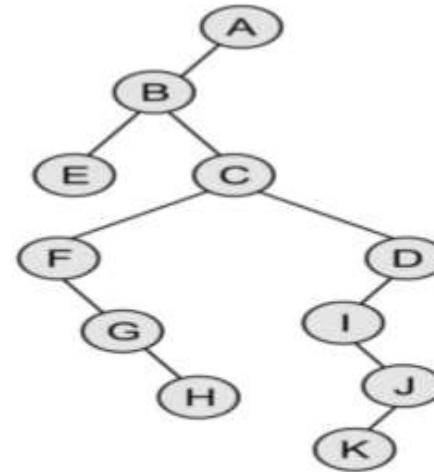
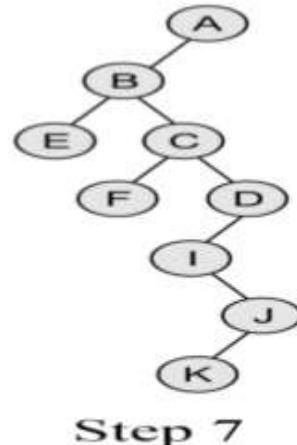
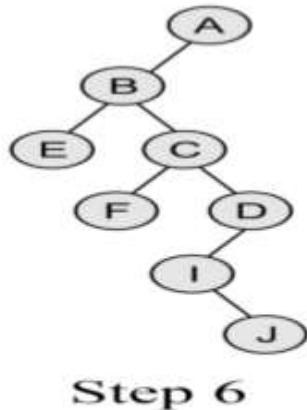
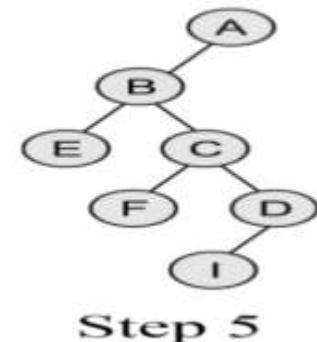
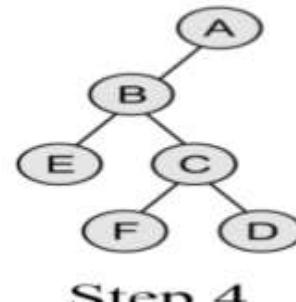
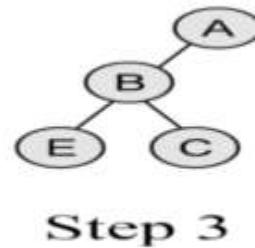
Stage 2 tree by using the above mentioned procedure is as follows:



# Convert this tree to binary tree

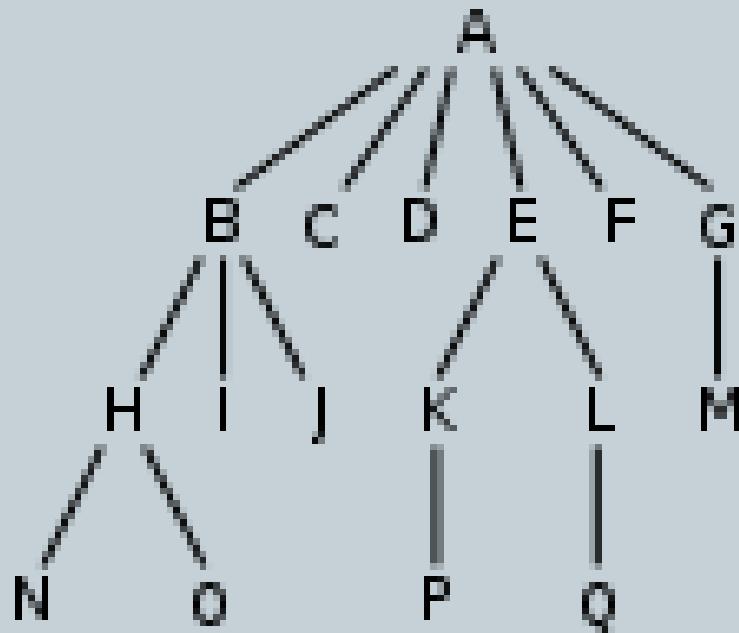


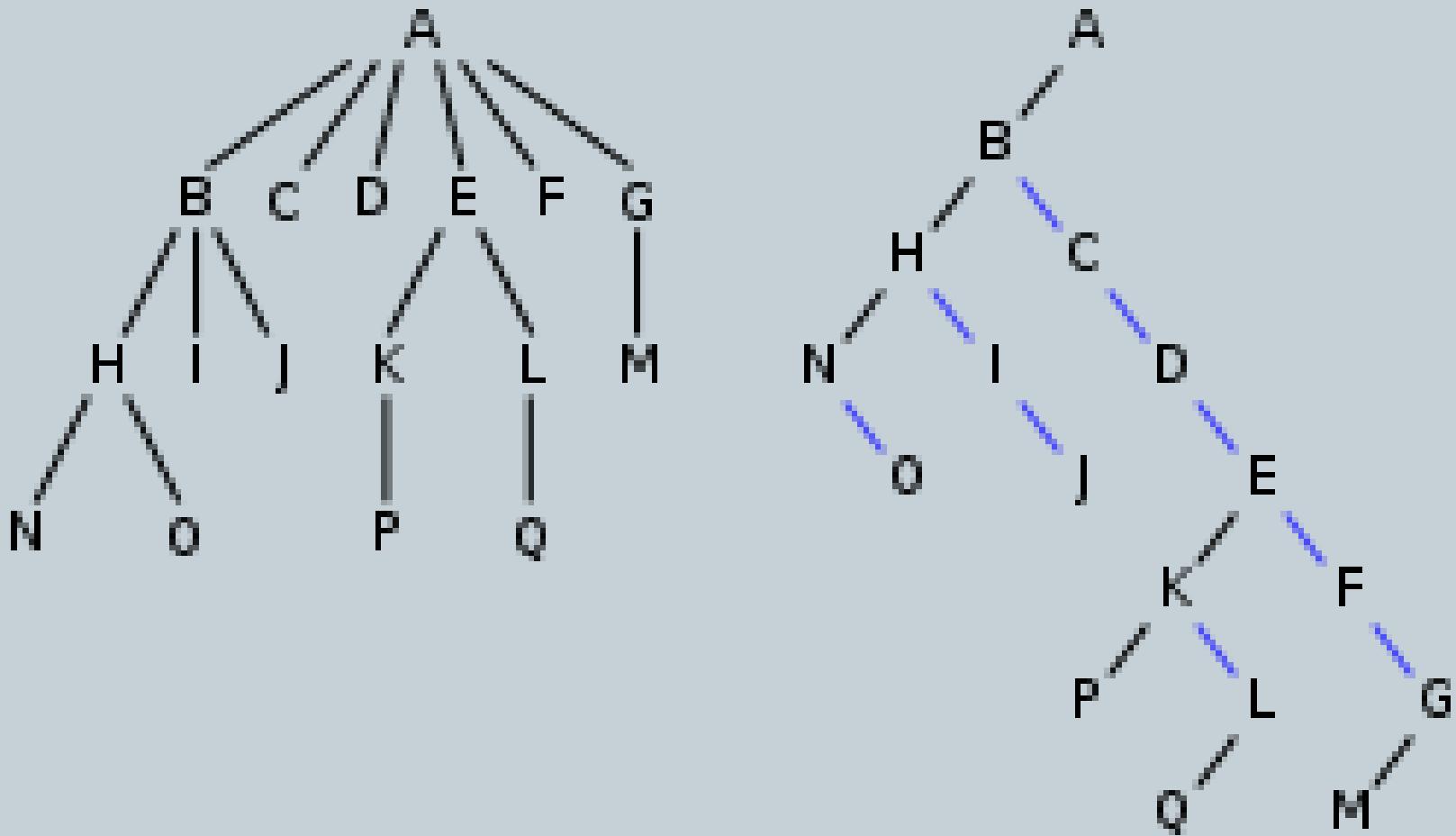
# Solution:



Step 8

# Convert this tree to binary tree





# NEW TOPIC (UNIT1:Trees)



- Syllabus
- Difference between linear and non-linear data structure
- Trees and Binary trees - basic terminology, representation using linked organization
- Binary tree- properties, converting tree to binary tree
- Binary tree traversals recursive and non-recursive
- Level wise Depth first and Breadth first.
- Binary Search Tree (BST), BST operations

# List of Some Binary Tree Operations

- Create, Insert and Delete Node
- Determine the height.
- Determine the number of nodes.
- Make a clone.
- Determine if two binary trees are clones.
- Display the binary tree.
- Evaluate the arithmetic expression represented by a binary tree.
- Obtain the in order form of an expression.
- Obtain the pre order form of an expression.
- Obtain the post order form of an expression.

# Binary Tree Traversal



- Many binary tree operations are done by performing a **traversal** of the binary tree.
- In a traversal, each element of the binary tree is **visited** exactly once.
- During the **visit** of an element, all action (make a clone, display, evaluate the operator, etc.) with respect to this element is taken.

# Tree Traversal

- Two main methods:
  - Depth first Traversal
  - Breadth first Traversal
- Three methods of Depth first Traversal
  - Preorder
  - Postorder
  - Inorder
- PREorder:
  - visit the root
  - Traverse left subtree
  - Traverse right subtree

# Tree Traversal



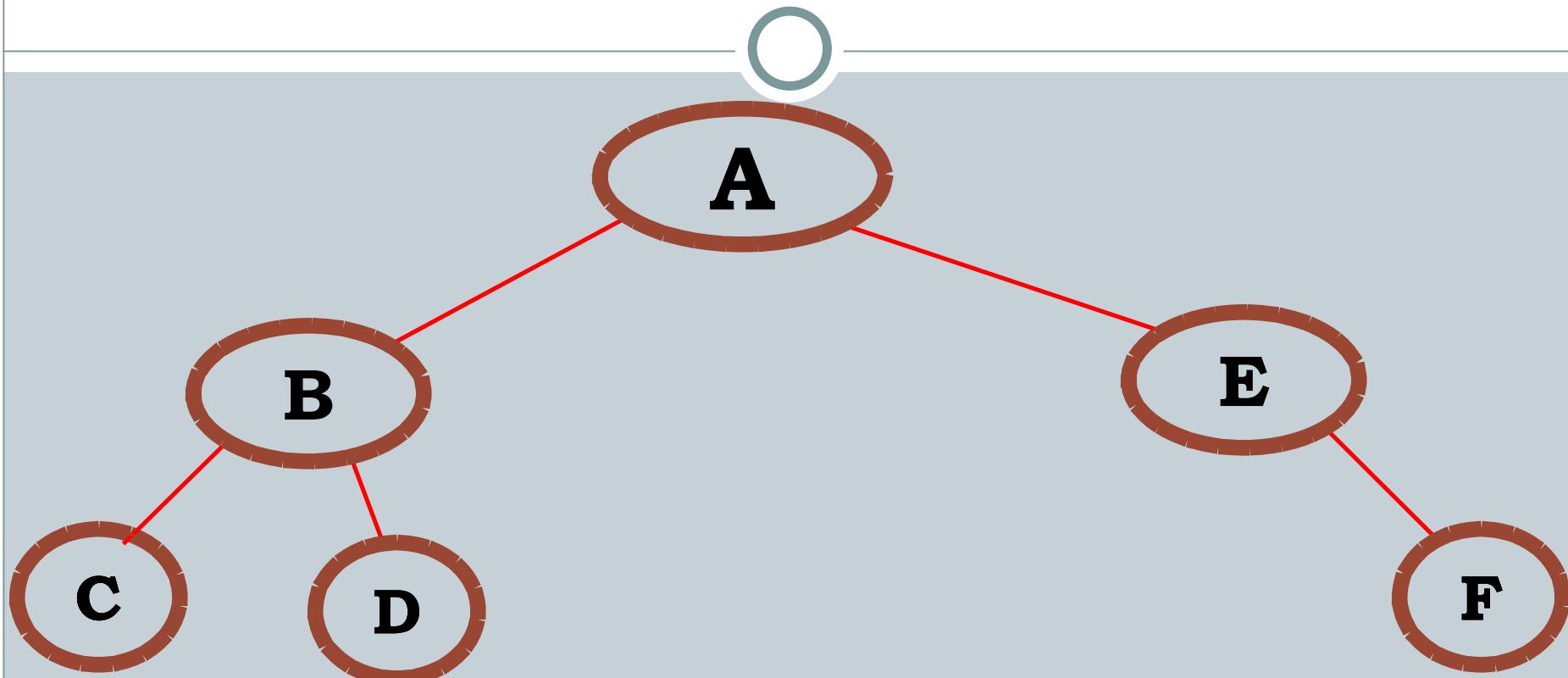
- **POSTorder**

- Traverse left subtree
- Traverse right subtree
- visit the root

- **Inorder**

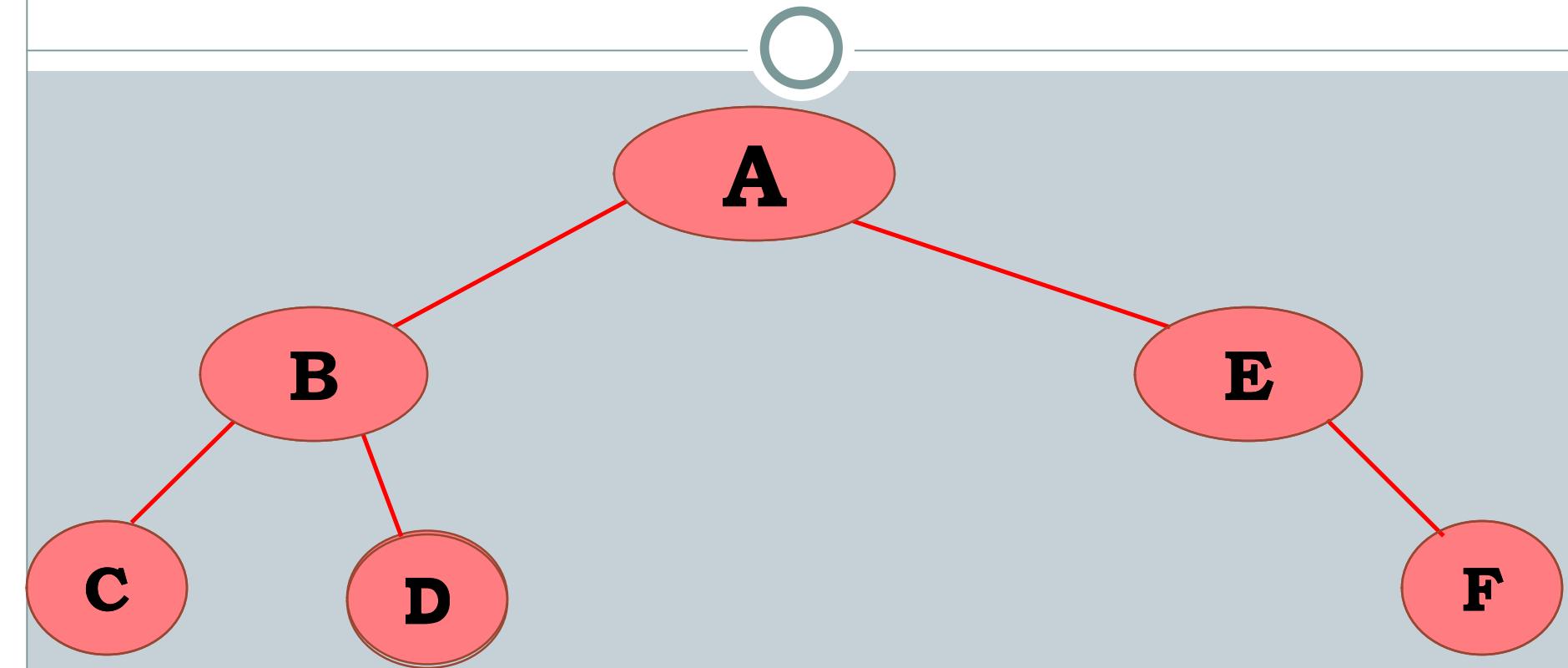
- Traverse left subtree
- visit the root
- Traverse right subtree

# In-Order Traversal



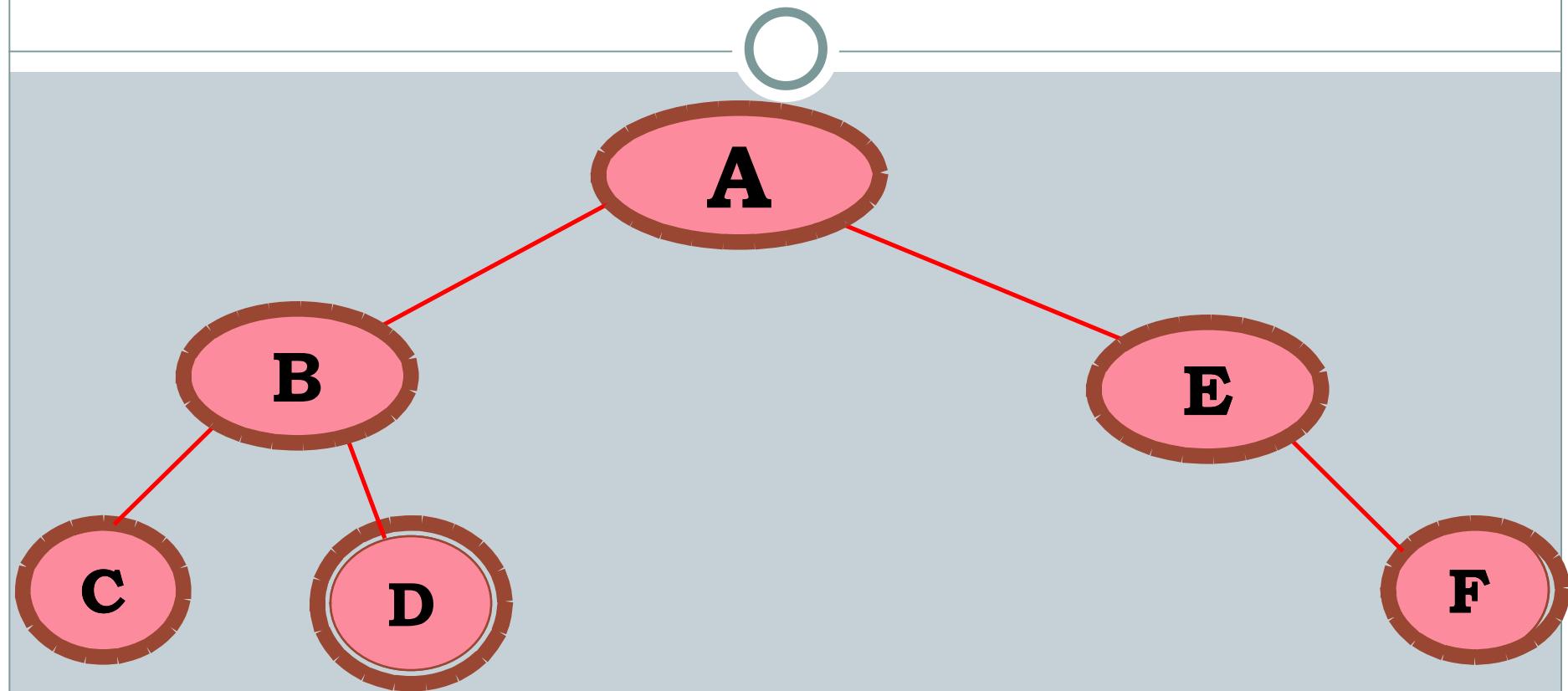
C   B   D   A   E   F

# Pre-Order Traversal



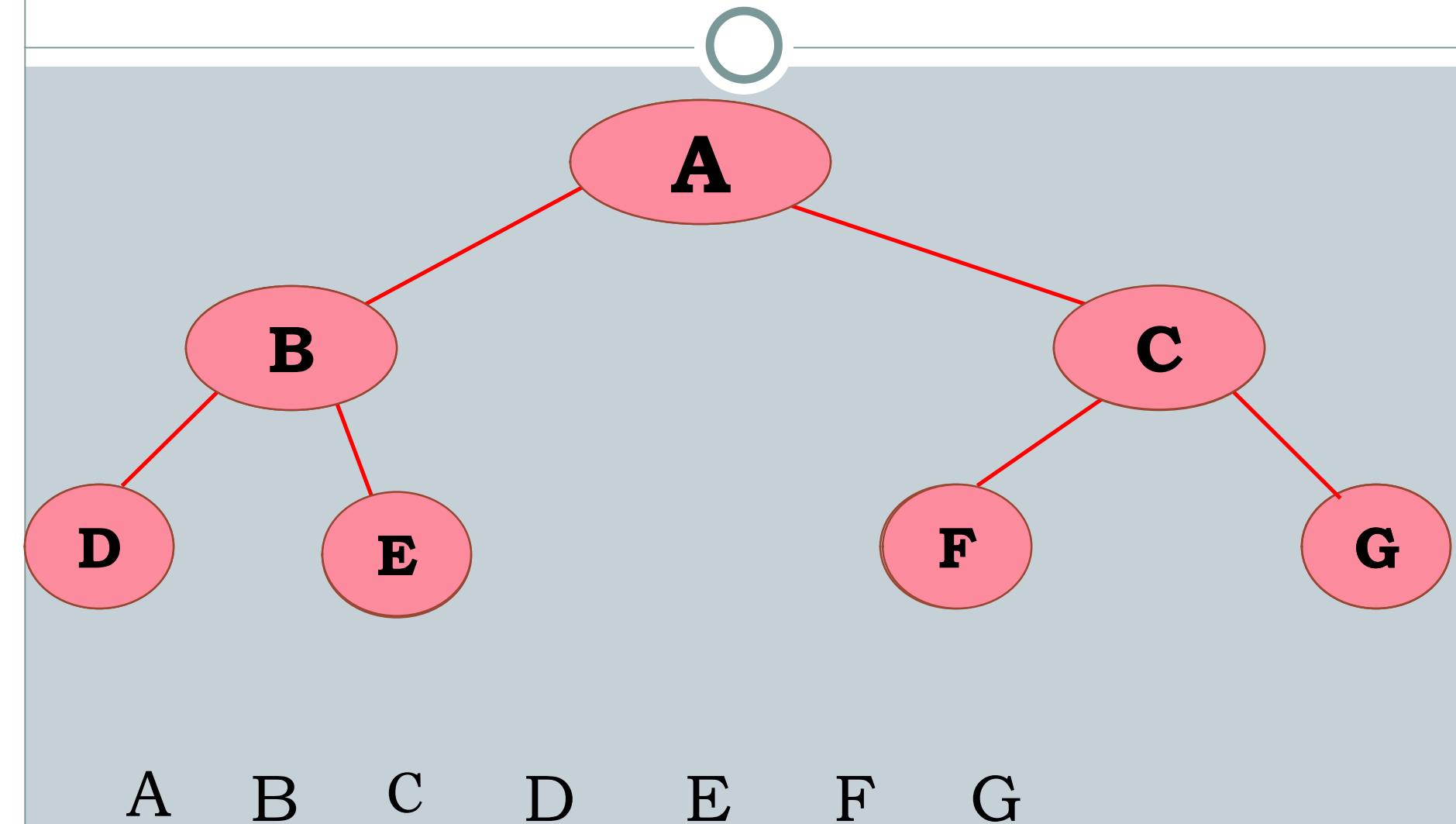
A   B   C   D   E   F

# Post-Order Traversal

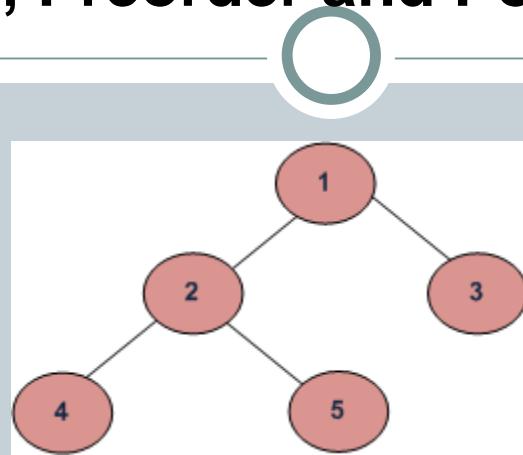


C D B F E A

# Breadth first Traversal



# Find In order, Preorder and Post Order traversals:

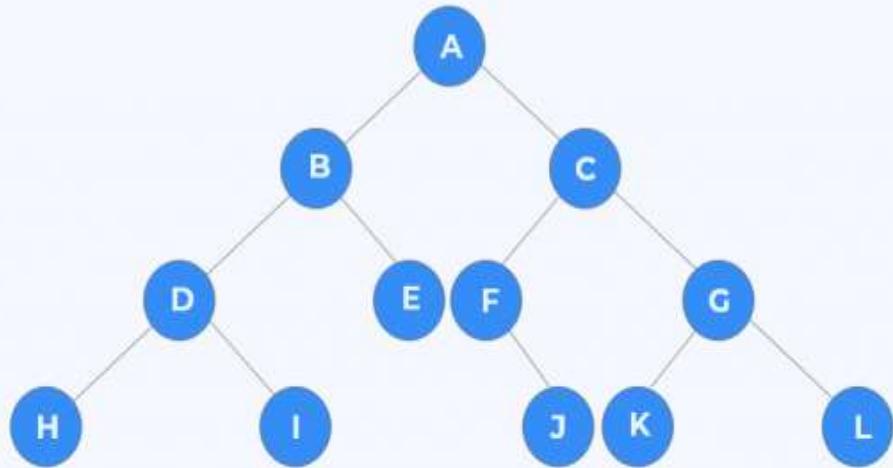


Depth First Traversals:

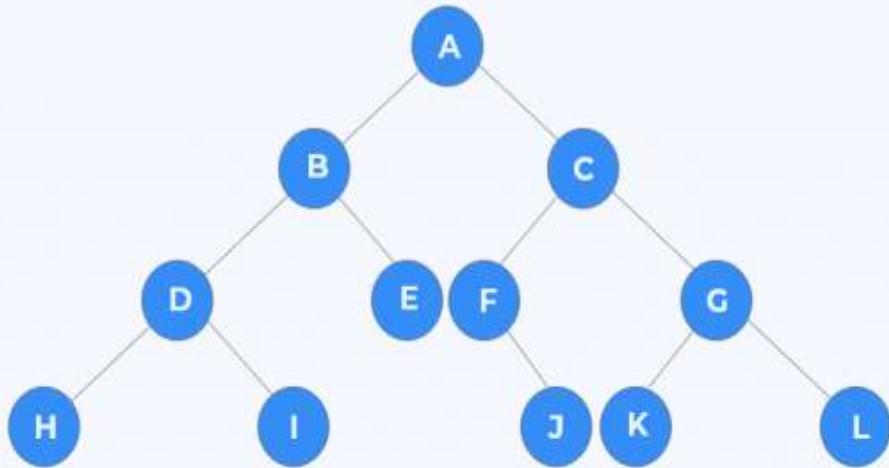
- (a) Inorder (Left, Root, Right) : 4 2 5 1 3
- (b) Preorder (Root, Left, Right) : 1 2 4 5 3
- (c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5

# Inorder Postorder Preorder Example 1



# Inorder Postorder Preorder Example 1



PreOrder:

A B D H I E C F J G K L

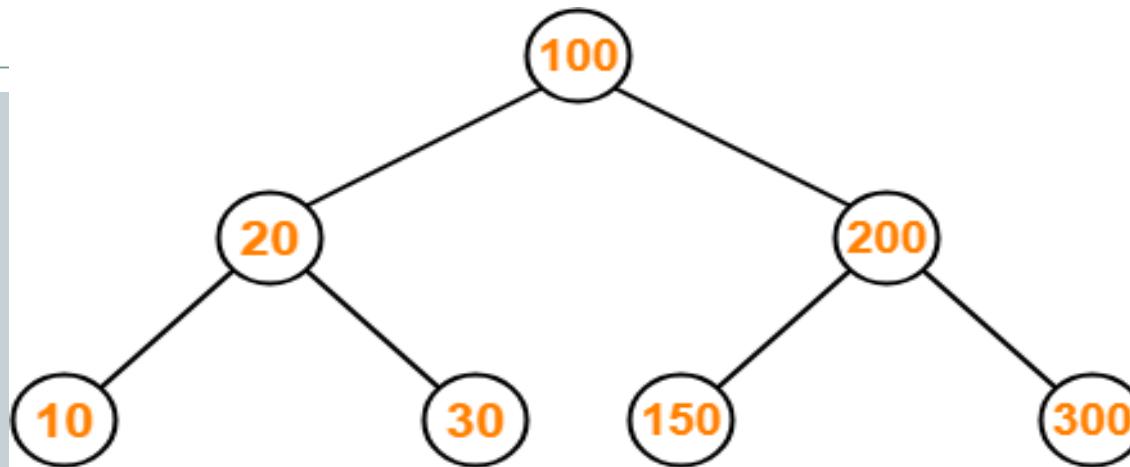
PostOrder:

H I D E B J F J L G C A

Inorder:

H D I B E A F J C K G L

# Find In order, Preorder and Post Order traversals:



Binary Search Tree

## Preorder Traversal-

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

## Inorder Traversal-

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

## Postorder Traversal-

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

# NEW TOPIC (UNIT1:Trees)



- **Syllabus**
- Difference between linear and non-liner data structure
- Trees and Binary trees - basic terminology, representation using linked organization
- Binary tree- properties, converting tree to binary tree
- Binary tree traversals recursive and non-recursive
- Level wise Depth first and Breadth first.
- **Binary Search Tree (BST), BST operations**

# Binary Search Trees



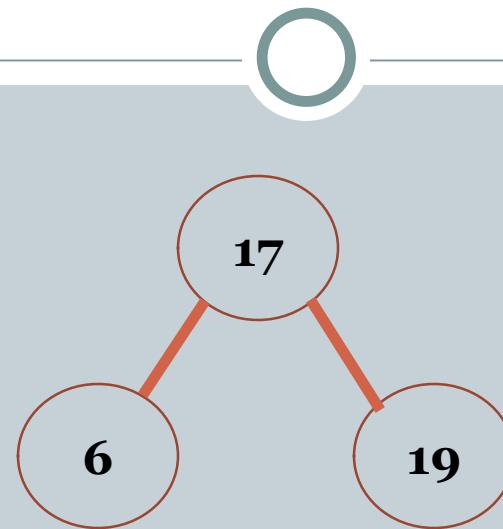
- Ordered data stored in Array will have efficient search algorithms but inefficient insertion and deletion
- Linked list storage will increase efficiency of insertion and deletion but search will always be sequential as you always have to start from ROOT node

# Properties of Binary Search Trees

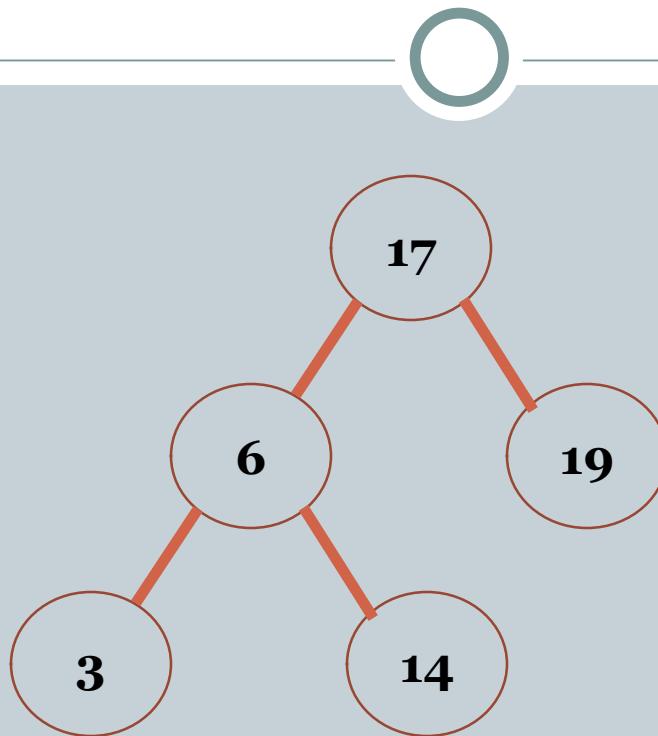


- All items in left sub tree are less than root
- All items in right sub tree are greater than or equal to root
- Each sub tree is itself a binary search tree

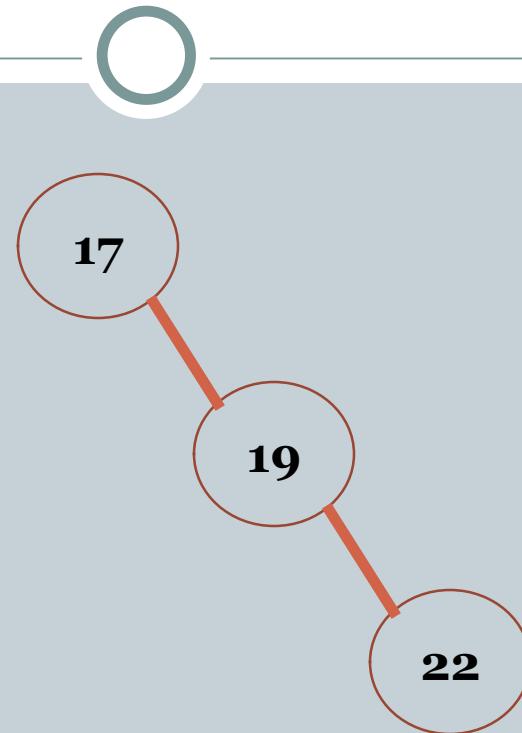
# Is this Binary Search Tree



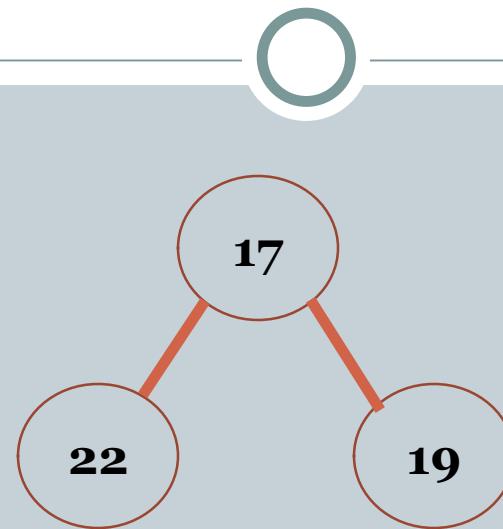
# Is this Binary Search Tree



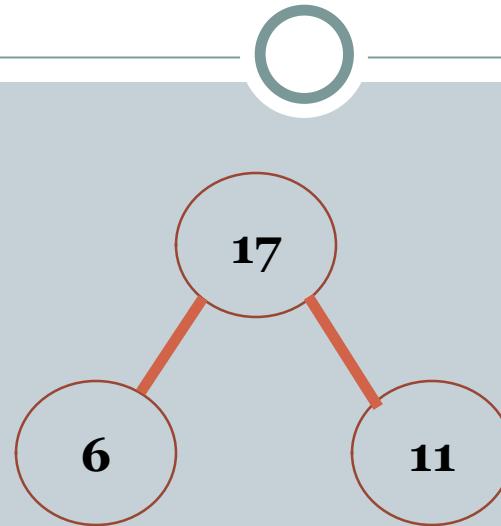
# Is this Binary Search Tree



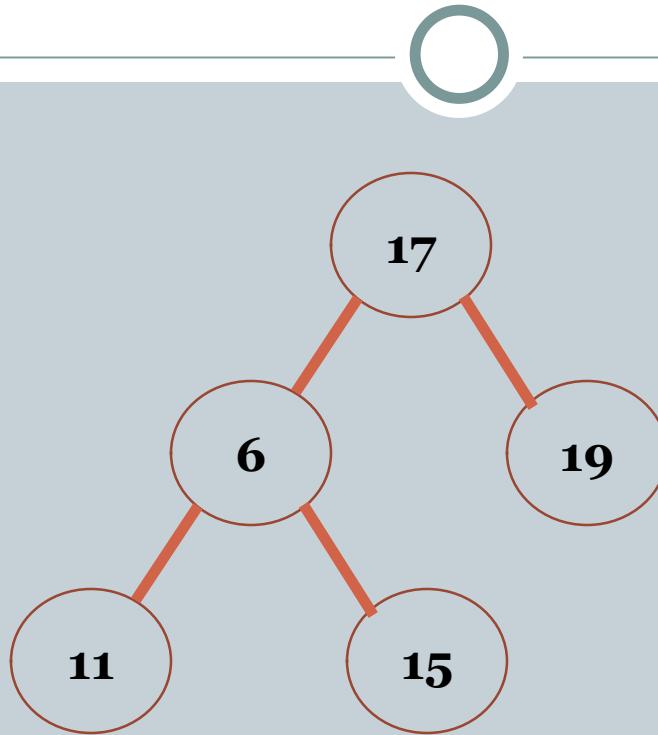
# Is this Binary Search Tree



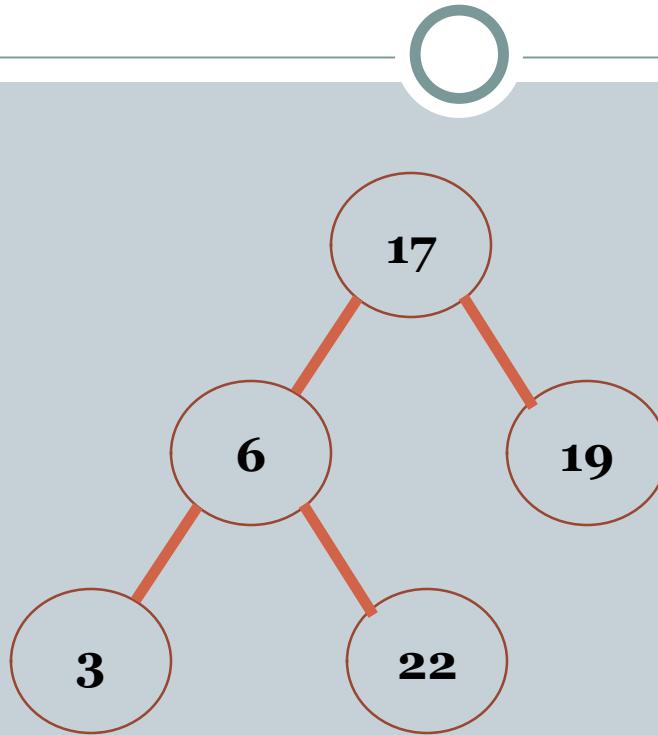
# Is this Binary Search Tree



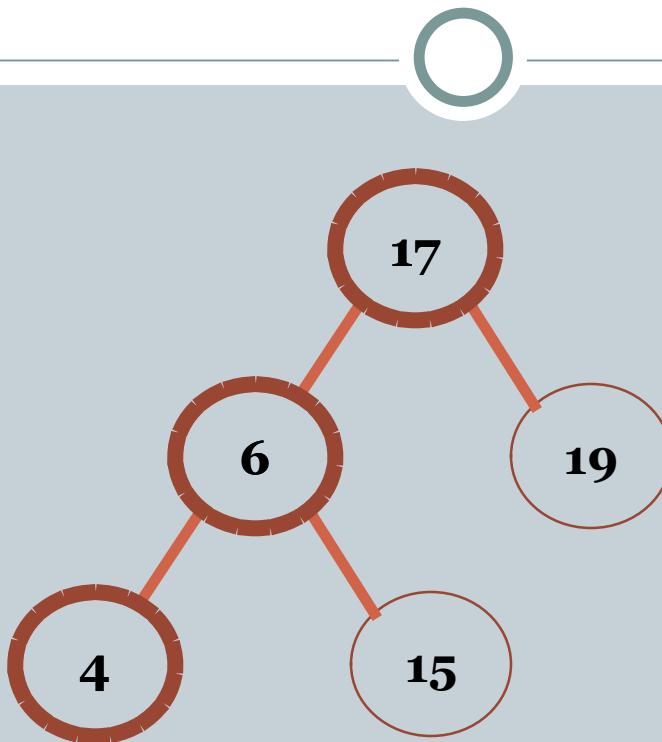
# Is this Binary Search Tree



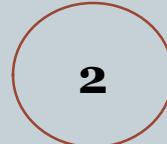
# Is this Binary Search Tree



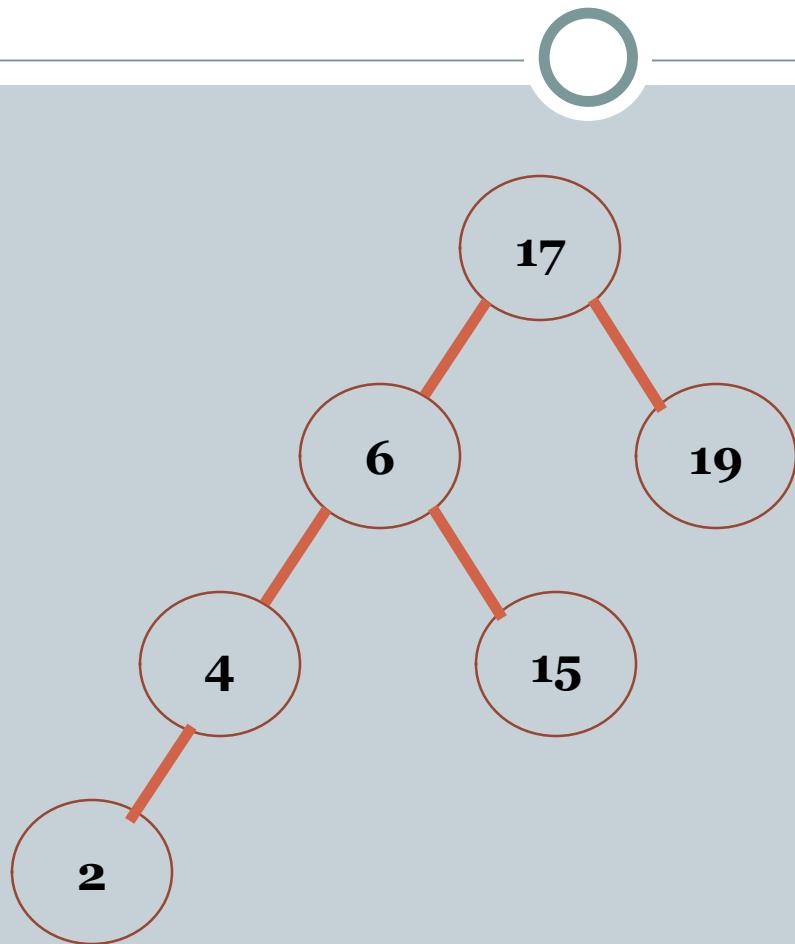
# Insertion in BST



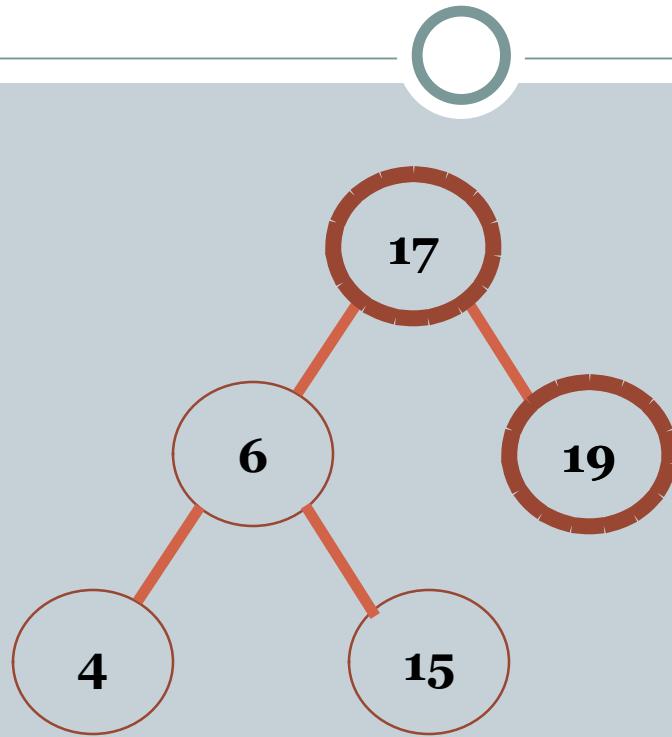
Add node -



# Insertion in BST



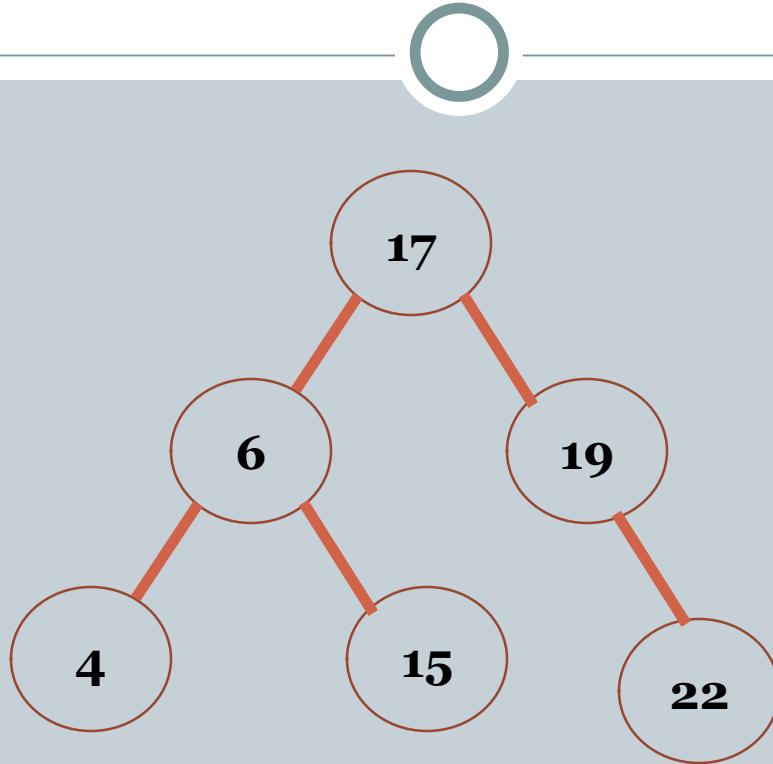
# Insertion in BST



Add node -



# Insertion in BST



Construct a Binary Search Tree by inserting the following sequence of numbers...

**10,12,5,4,20,8,7,15 and 13**



insert (10)



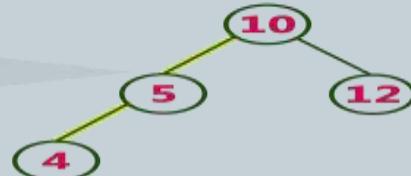
insert (12)



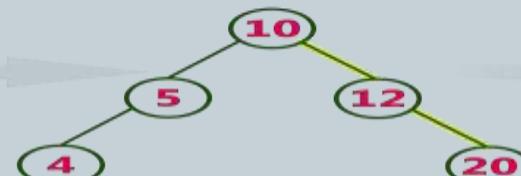
insert (5)



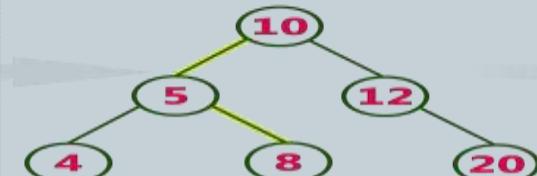
insert (4)



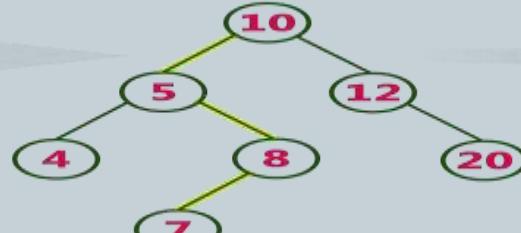
insert (20)



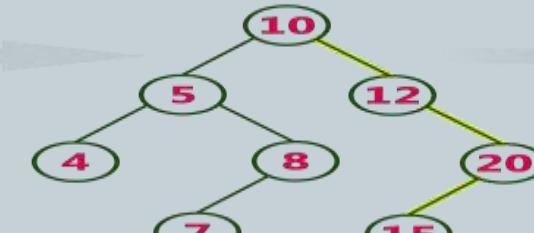
insert (8)



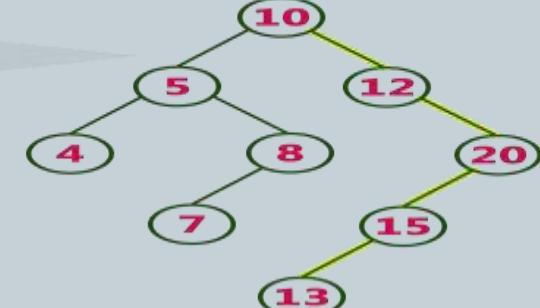
insert (7)



insert (15)



insert (13)

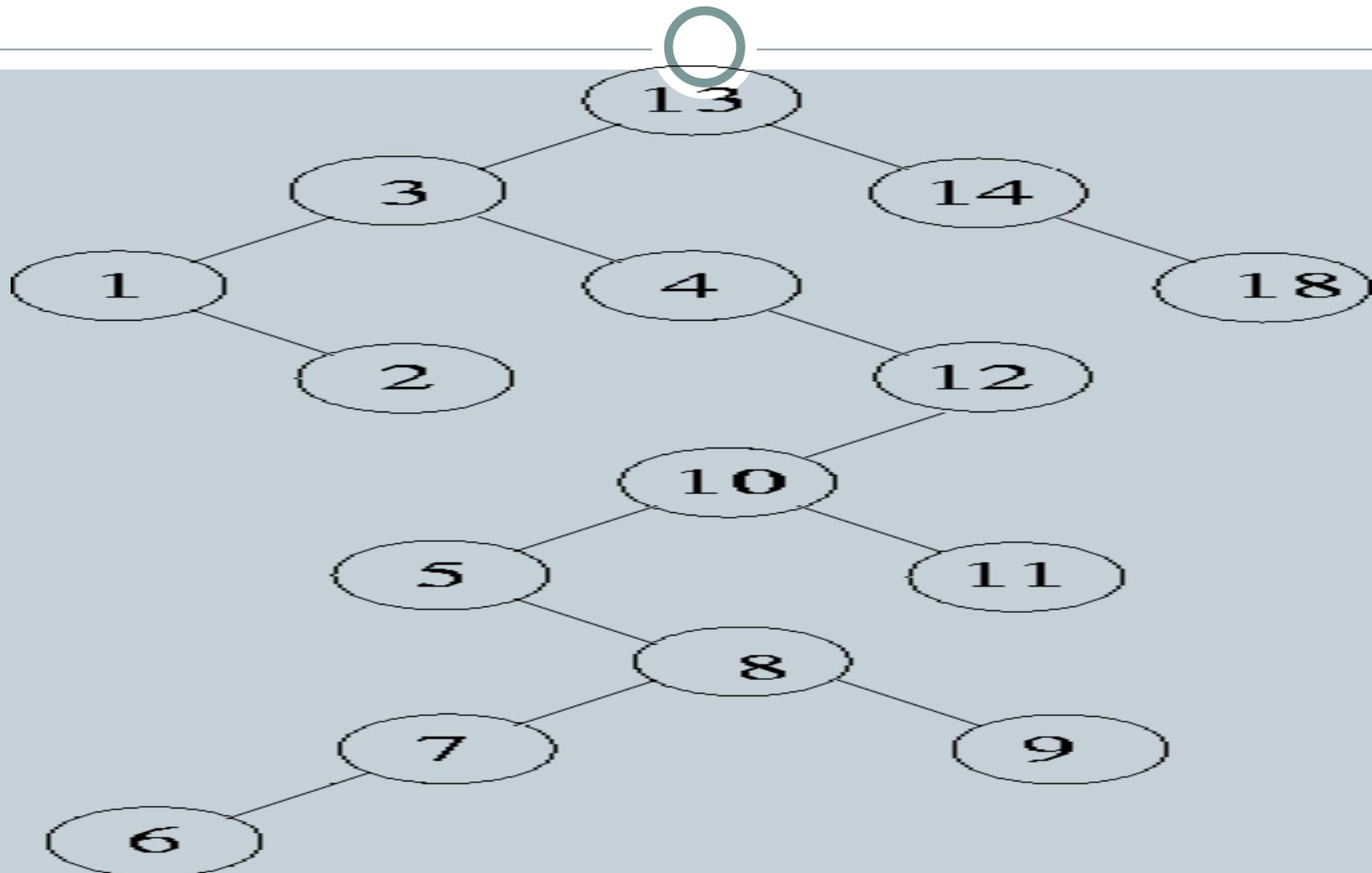


Construct a Binary Search Tree by inserting the following sequence of numbers...

13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



Construct a Binary Search Tree by inserting the following sequence of numbers...  
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



# Insertion (Recursive)



- Algorithm addBST (root, newnode)

If (empty)

    Set root to newnode

    Return newnode

End if

If (newnode < root)

    Return addBST (left sub tree, newnode)

Else

    Return addBST (right sub tree, newnode)

End if

End addBST

# Deleting from BST

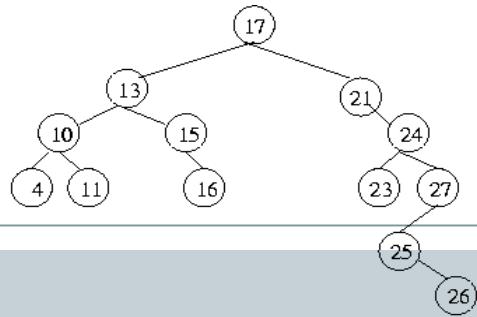


- If the node doesn't have children's then just delete the reference for this node from its parent and recycle the memory
- If the node has **only right sub tree** then  
Simply attach right subtree to delete's nodes parent
- If the node has **only left sub tree** then  
Simply attach left subtree to delete's nodes parent

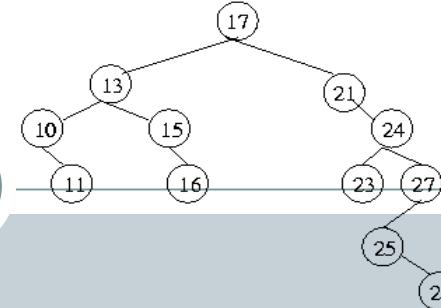
# Deleting from BST



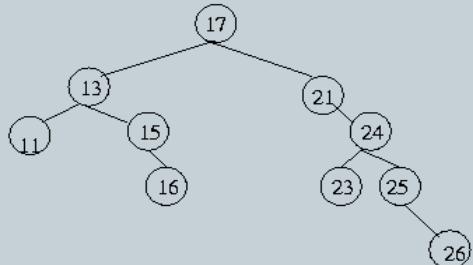
- If the node has **left and right** subtree then
  - Find **largest node from left subtree** and replace that with node we have to delete  
**OR**
  - Find **smallest node from right subtree** and replace that with node we have to delete



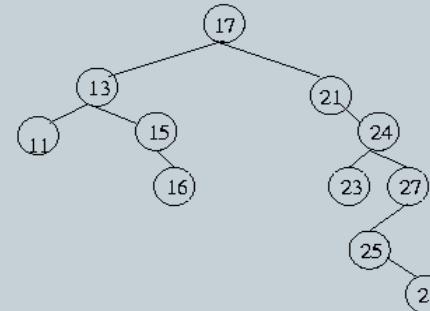
delete 4  
/\* delete leaf \*/



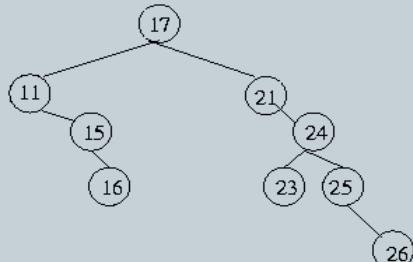
delete 10  
/\* delete a node with no left subtree \*/



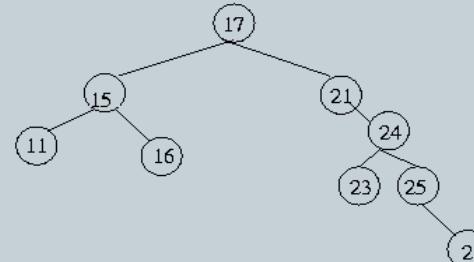
delete 27  
/\* delete node with no right subtree \*/



delete 13  
/\* delete node with both left and right subtrees \*/



**Method 1.**  
Find highest valued element among the descendants of left child



**Method 2**  
Find lowest valued element among the descendants of right child

# Removal in BST - Pseudocode



## **method remove (key)**

I **if** the tree is empty return false  
II Attempt to locate the node containing the target using the binary search algorithm  
    **if** the target is not found return false  
    **else** the target is found, so remove its node:  
        Case 1: **if** the node has 2 empty subtrees  
            replace the link in the parent with null

Case 2: **if** the node has a left and a right subtree

- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

# Removal in BST - Pseudocode



Case 3: if the node has no left child

- link the parent of the node to the right (non-empty) subtree

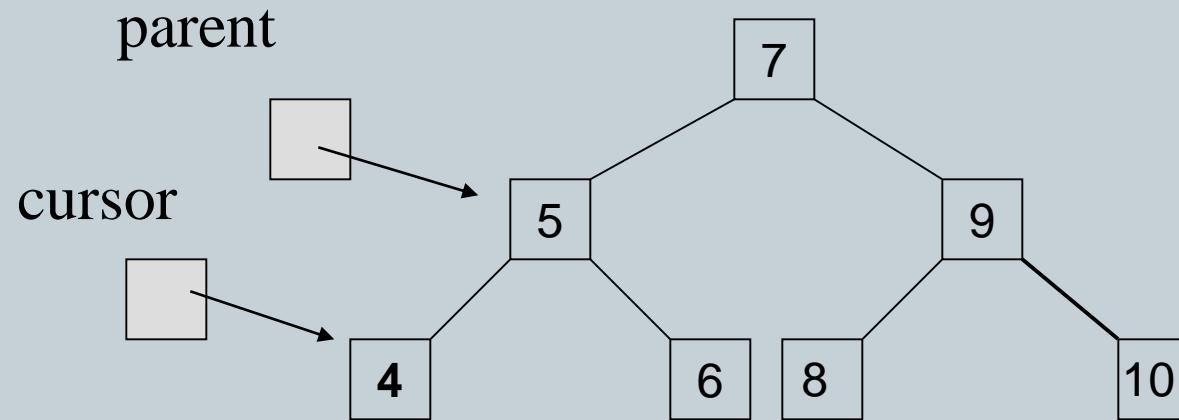
Case 4: if the node has no right child

- link the parent of the target to the left (non-empty) subtree

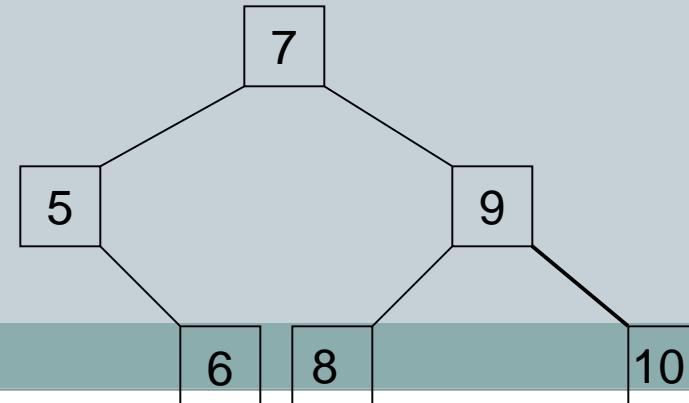
# Removal in BST: Example



Case 1: removing a node with 2 EMPTY SUBTREES



Removing 4 replace the link in the parent with null



# Removal in BST: Example

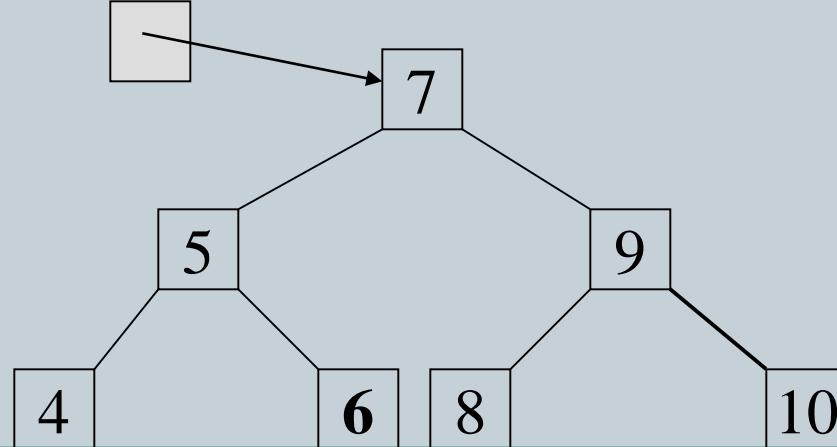


## Case 2: removing a node with 2 SUBTREES

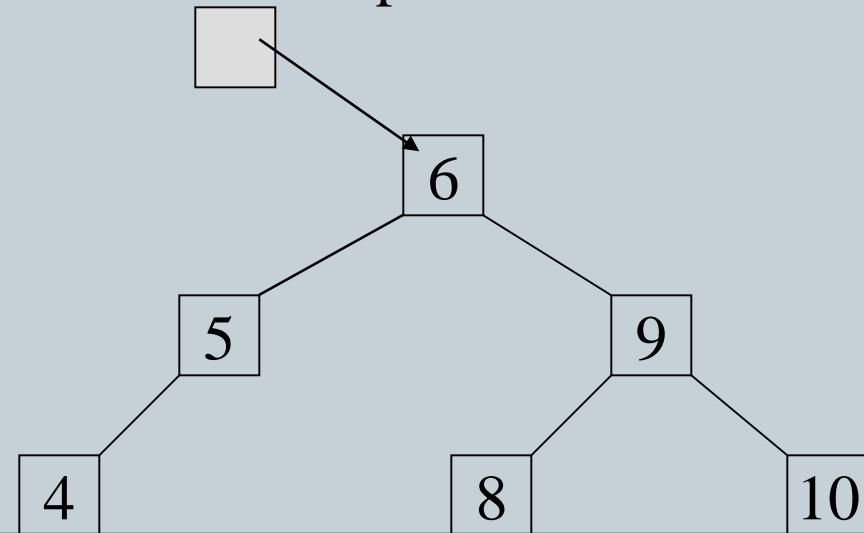
- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

*Removing 7*

cursor



cursor



What other element  
can be used as  
replacement?

# Removal in BST: Example

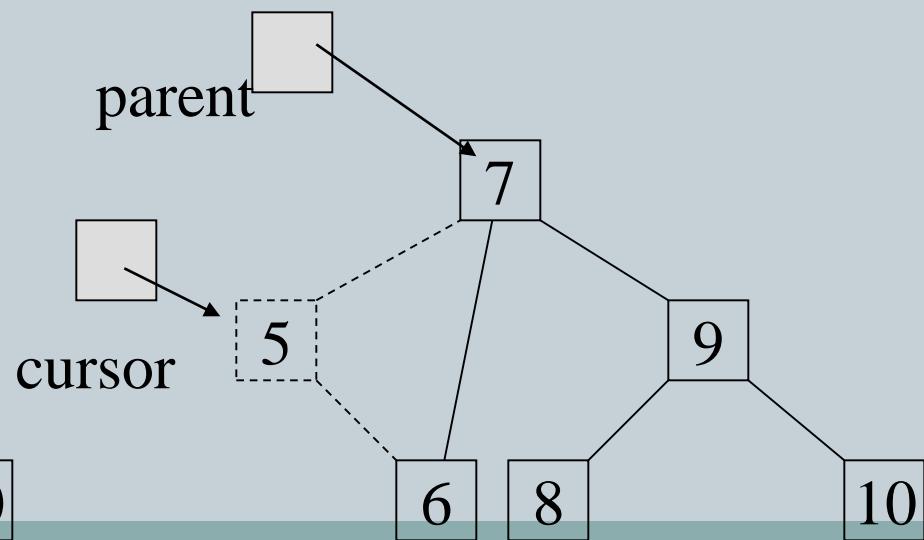
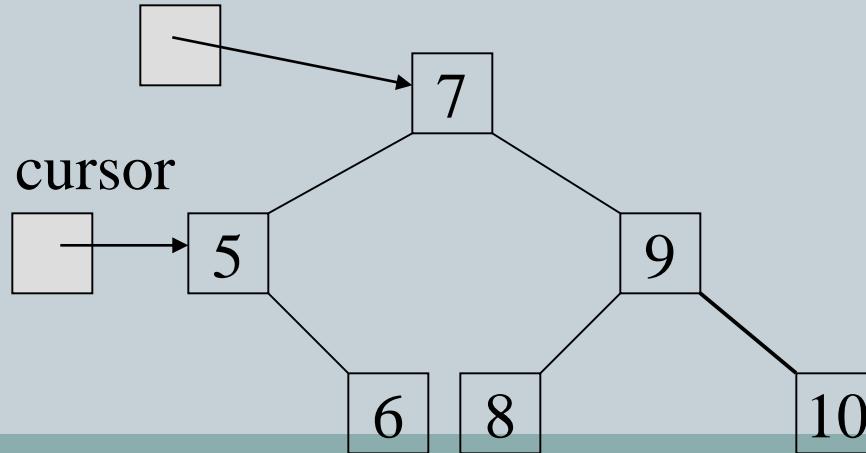


## Case 3: removing a node with 1 EMPTY SUBTREE

the node has no left child:

link the parent of the node to the right (non-empty) subtree

parent



# Removal in BST: Example



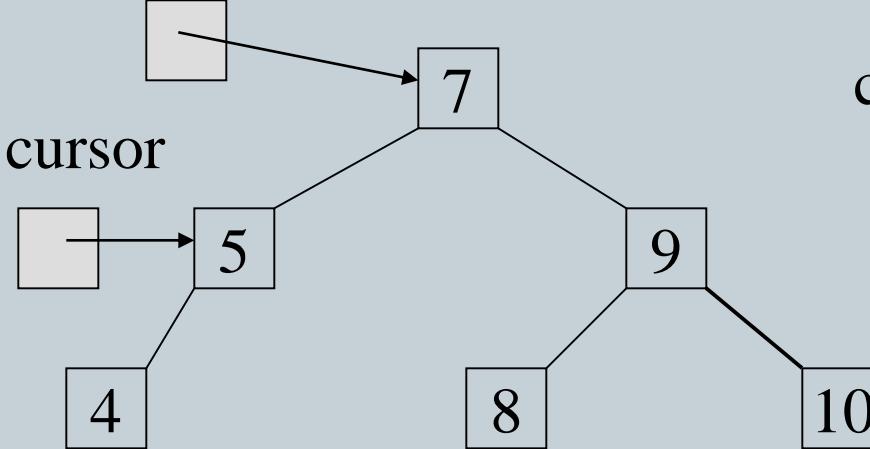
**Case 4**: removing a node with 1 EMPTY SUBTREE

the node has no right child:

link the parent of the node to the left (non-empty) subtree

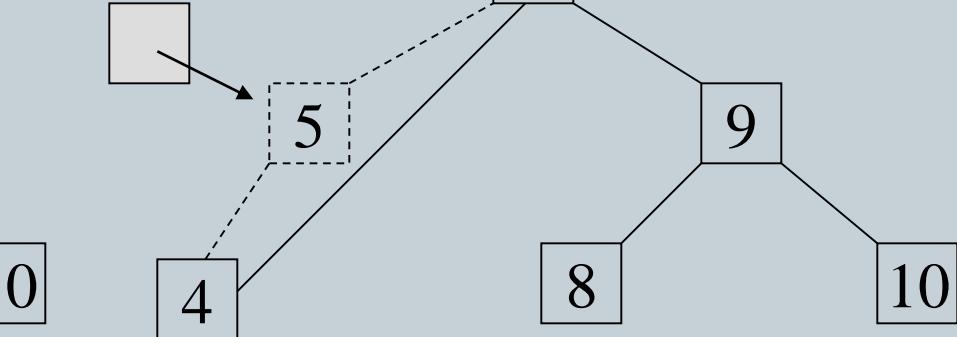
Removing 5

parent



parent

cursor





## Program For implementation of Binary Search tree

1. Creation of BST
2. Insertion In BST
3. Display BST
4. Search Element form BST,
5. Find MIN, MAX
6. Delete a node

# Inorder traversal without Recursion (Non-Recursive /Iterative):



## Iterative solution

For recursion, we use implicit stack. So here to convert recursive solution to iterative, we will use explicit stack.

Steps for iterative solution:

1. If root is null (BST empty so) Return
2. Create an empty stack S
3. Initialize currentNode as root and push currentNode in stack
4. If S==Empty AND currentNode ==NULL   Return (Work Done)
5. If currentNode is Not NULL
  1. Push the currentNode onto S and set currentNode = currentNode.left
6. ElseIf currentNode is NULL AND S is not empty then
  1. Pop the top node from stack S and print the poppedNode
  2. set currentNode = poppedNode.right
7. Go to step 4

# Implementation in Non Recursive Way:



```
public void inOrderIter(TreeNode root) {  
  
    if(root == null)  
        return;  
  
    Stack<TreeNode> s = new Stack<TreeNode>();  
    TreeNode currentNode = root;  
  
    while(!s.empty() || currentNode!=null){  
  
        if(currentNode!=null)  
        {  
            s.push(currentNode);  
            currentNode=currentNode.left;  
        }  
        else  
        {  
            TreeNode n=s.pop();  
            System.out.printf("%d ",n.data);  
            currentNode=n.right;  
        }  
    }  
}
```



# Thank You

# Advanced Data Structures





## Unit 2

# Advanced Trees

# **Advanced Trees : Contents**

---



- Threaded binary tree - concept
- In order traversal of in-order threaded binary tree
- AVL Trees
- Indexing and Multiway Trees
- Types of search tree, B-Tree, B+Tree, Trie Tree

# Threaded Binary Tree



- Need of it and Concept:
- Inorder traversal of a Binary tree can either be done using recursion or making use of a auxiliary stack.
- The idea of **threaded** binary trees is to make **inorder traversal faster** and do it without stack and without recursion.
- A binary tree is made threaded by making **all right child pointers** that would normally be NULL point to the **inorder successor of the node** (if it exists). Right Thread points to Successor node in INORDER traversal.
- **Left thread** to its INORDER **Predecessor**

<b>o</b>	<b>left</b>	<b>B</b>	<b>right</b>	<b>o</b>
----------	-------------	----------	--------------	----------

# Two types of threaded binary trees.



- **Single Threaded:** Where a NULL right pointers is made to point to the inorder successor (if successor exists)  
**Double Threaded:** Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively.
- The **predecessor** threads are useful for reverse inorder traversal and postorder traversal.
- The threads are also useful for **fast accessing ancestors** of a node.

# Threaded Binary Tree



- By doing this threading we avoid the recursive method of traversing a Tree, which makes use of stacks and consumes a lot of memory and time. The node structure for a **threaded binary tree** varies a bit and its like this -

```
class Node
{
    int value;
    Node left;
    Node right;
    boolean leftThread;
    boolean rightThread;
}
```

Which statements are TRUE for Threaded Binary Tree

- a) It points to the successor of the current node.
- b) Boolean field, rightThread, indicates if there is a right child.
- c) There will be no Right Child Thread.



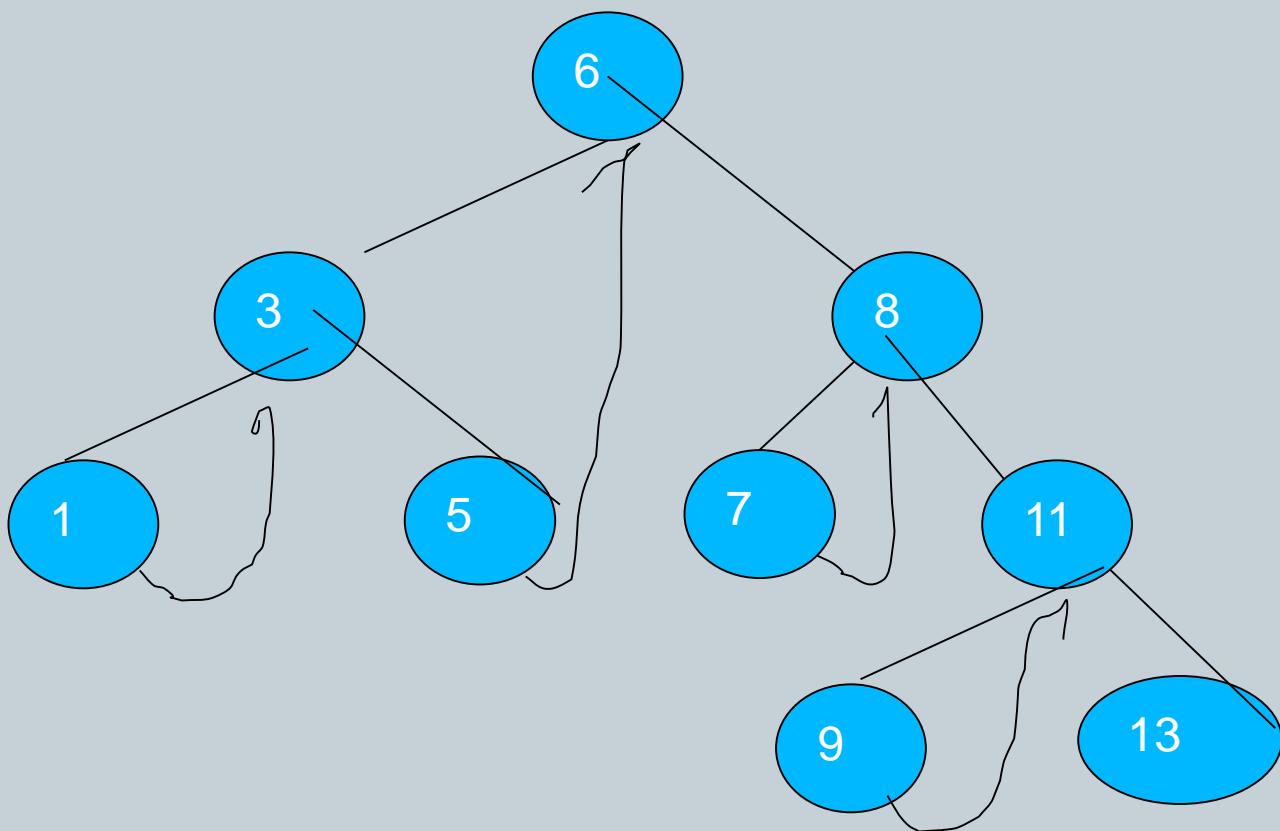
- Binary trees have a **lot of wasted space**: the leaf nodes each have 2 null pointers
- We can use these pointers to help us in **inorder** traversals
- We have the pointers reference to the next node in an inorder traversal; **called *threads***
- We need to know if a pointer is an **actual link (ref to a child)** or a **thread (ref to node in Inorder traversal)** , so we keep a **boolean** for each pointer

- Create threaded BST for given numbers:  
6, 3, 8, 1, 5, 7, 11, 9, 13

Find Inorder traversal sequence and make threads.

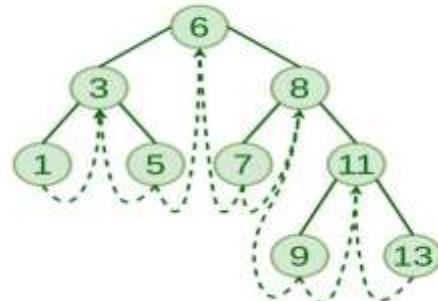
6, 3, 8, 1, 5, 7, 11, 9, 13

Inorder traversal - 1 3 5 6 7 8 9 11 13



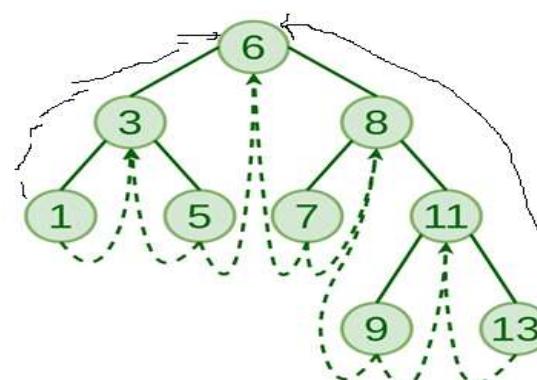
# Threaded Tree:

Double Threaded Binary Search Tree



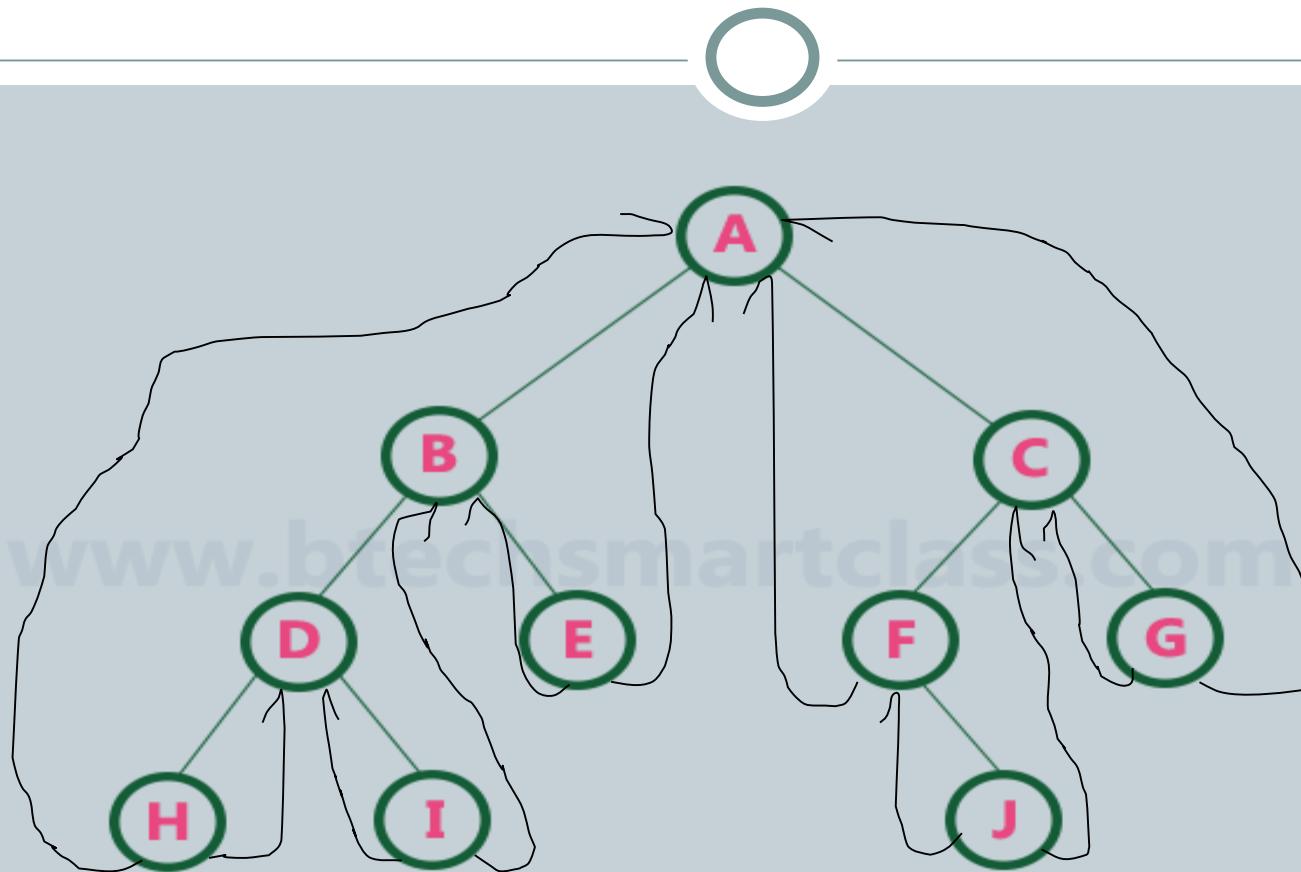
DG

Double Threaded Binary Search Tree



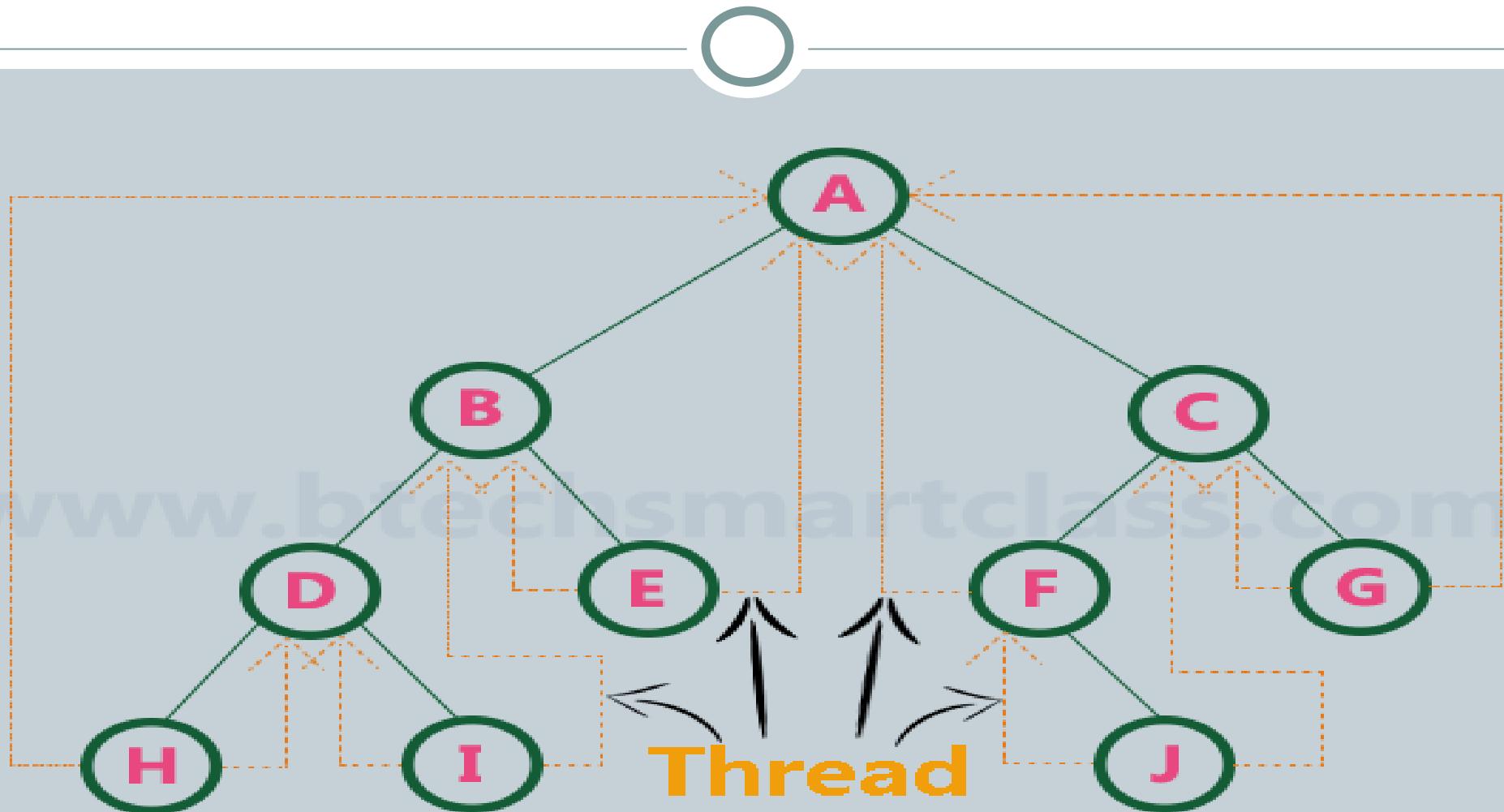
DG

# Convert given tree in threaded tree form



H D I B E A F J C G

# Solution:

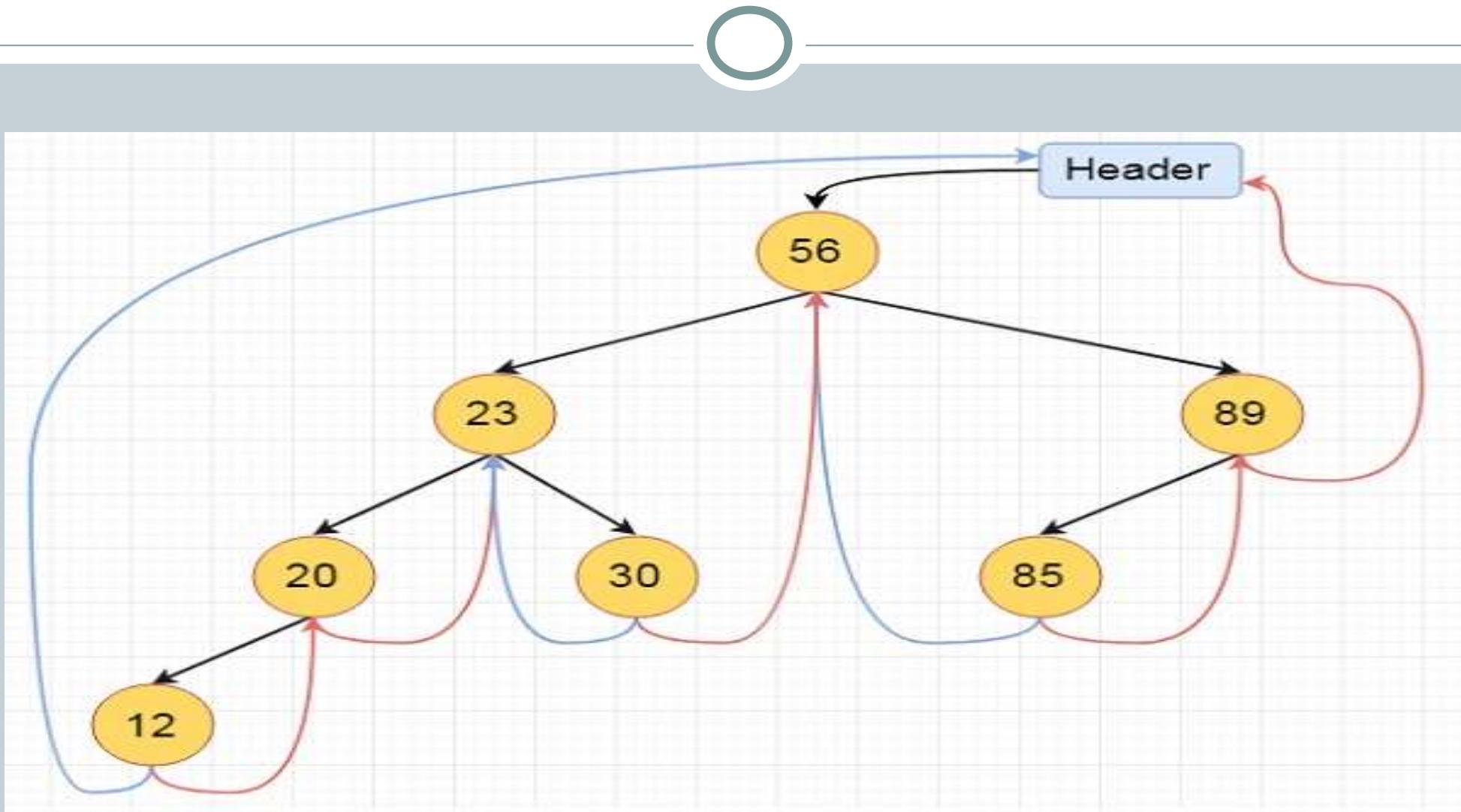


# Create threaded BST for given numbers.

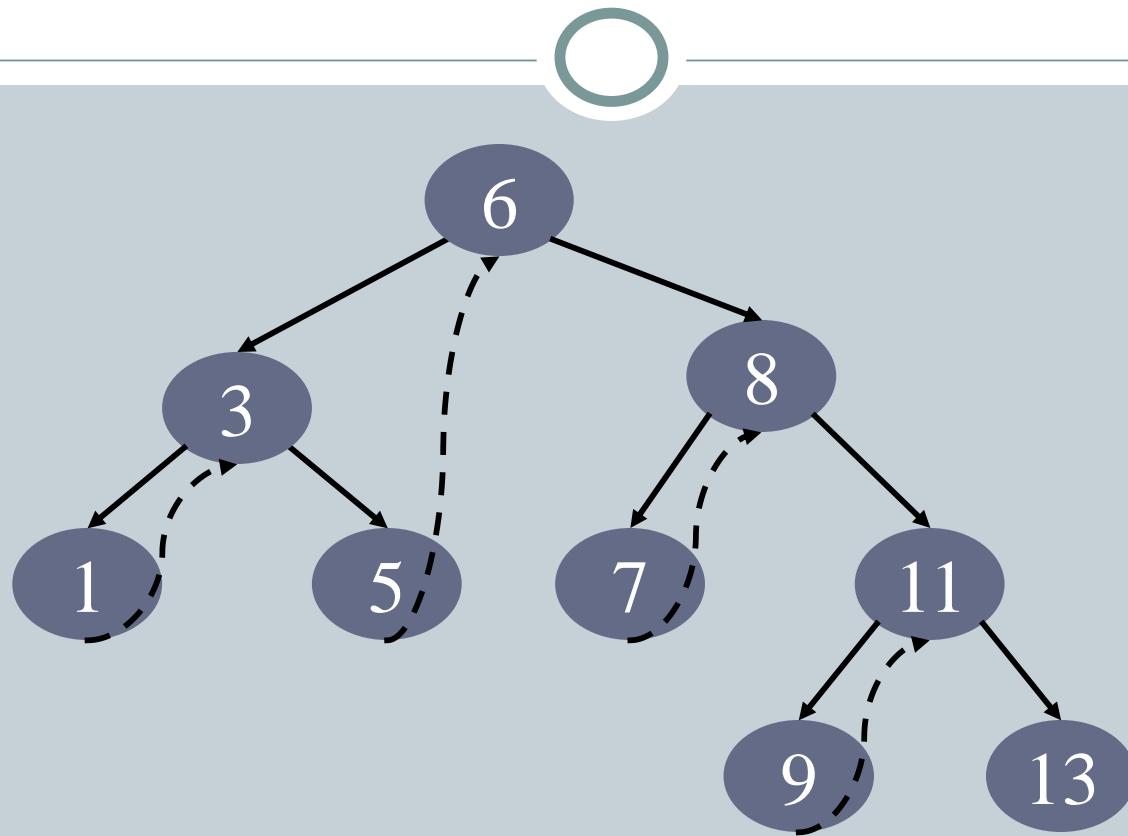
56, 23, 89, 20, 30, 85, 12

56, 23, 89, 20, 30, 85, 12    (inorder) 12, 20, 23, 30, 56, 85, 89

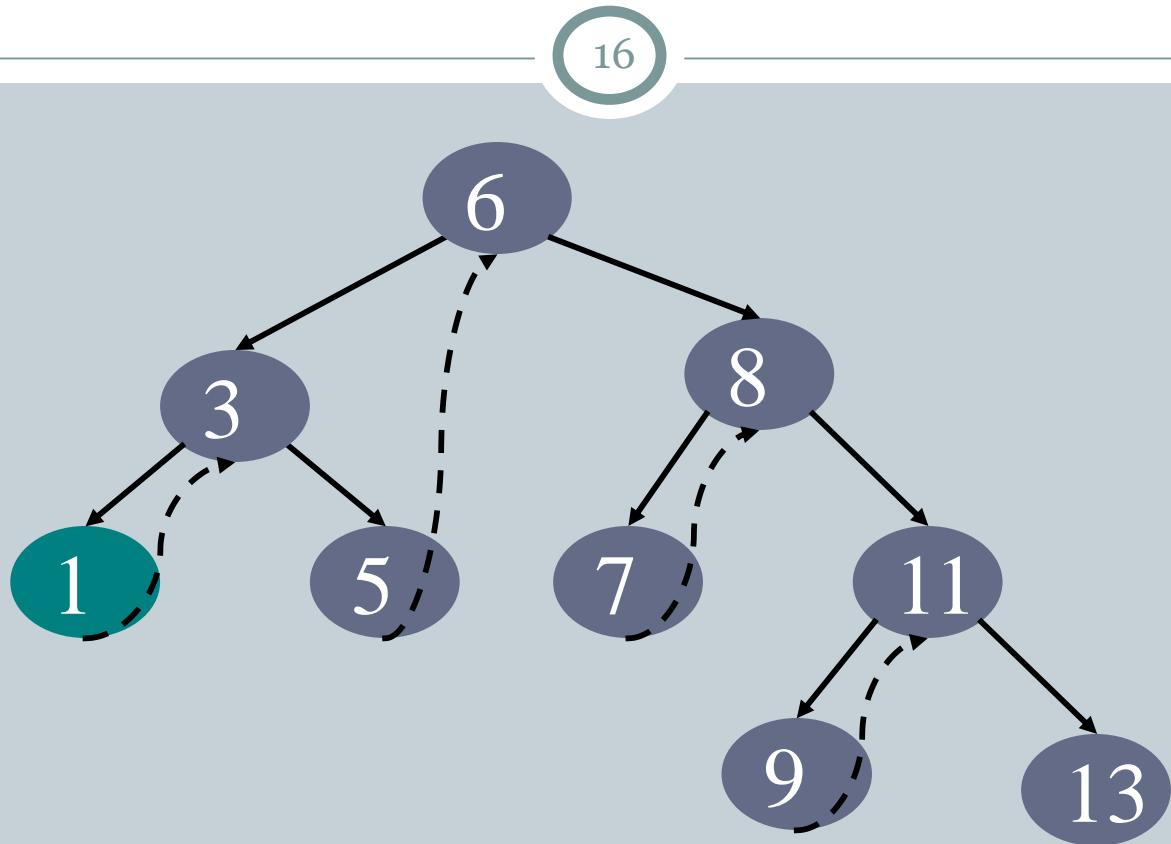
## Solution: for



# Threaded Binary Tree Example



# Threaded Tree Traversal

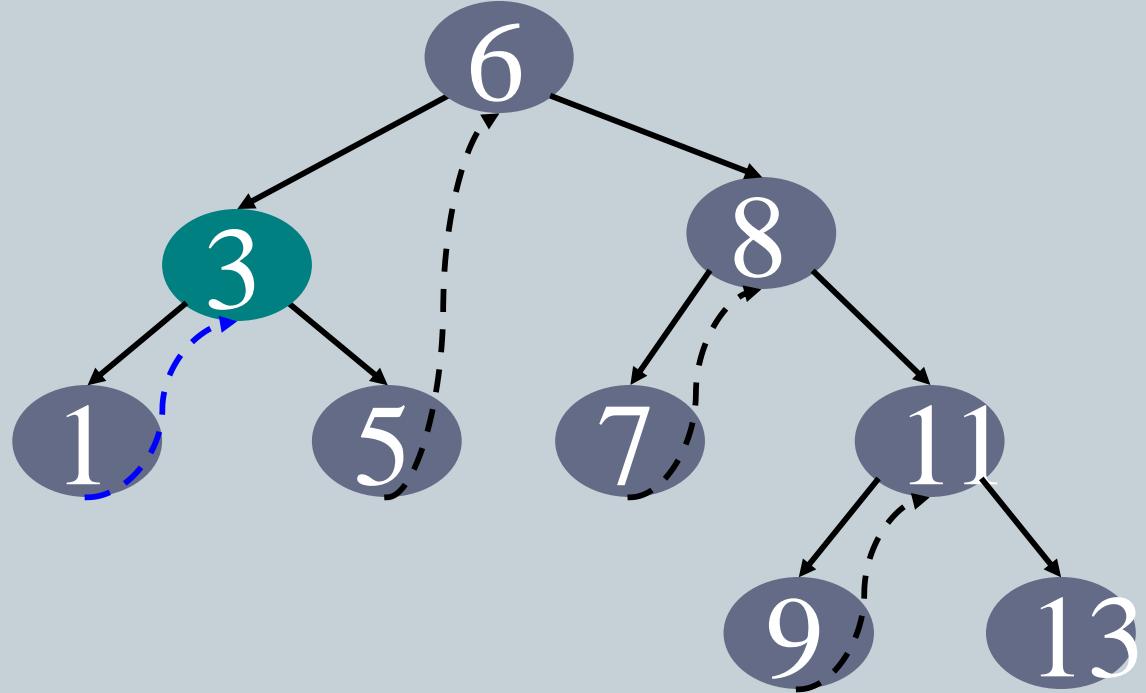


Output  
1

Start at leftmost node, print it

# Threaded Tree Traversal

17

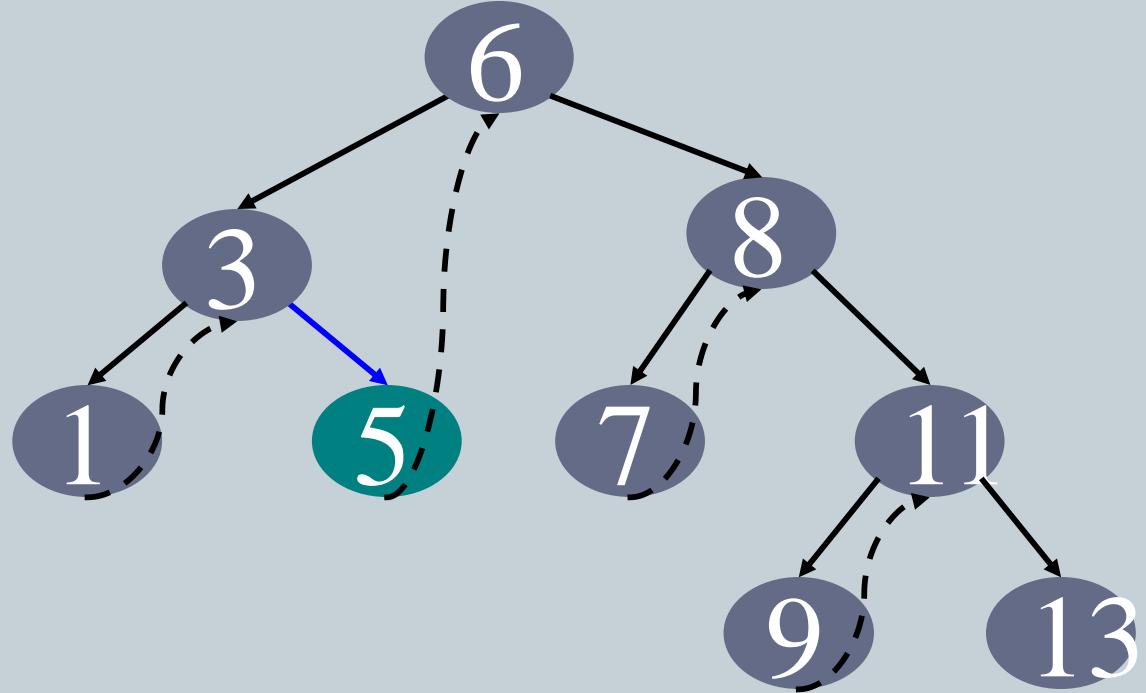


Output  
1  
3

Follow thread to right, print node

# Threaded Tree Traversal

18

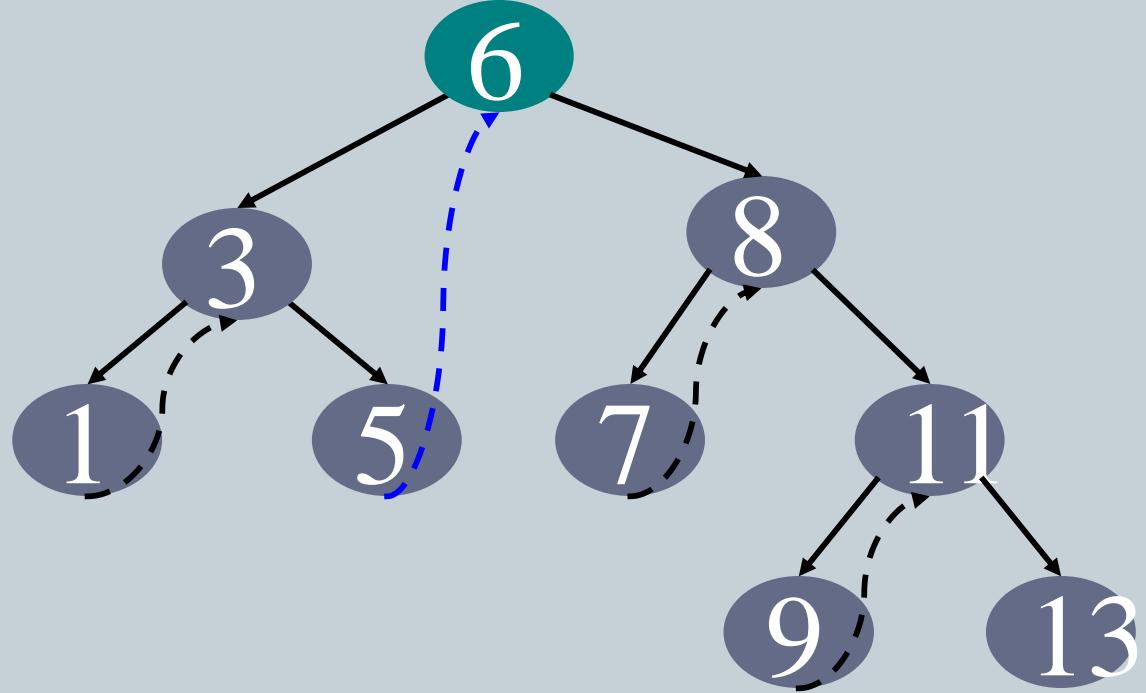


Output  
1  
3  
5

Follow link to right, go to leftmost node and print

# Threaded Tree Traversal

19

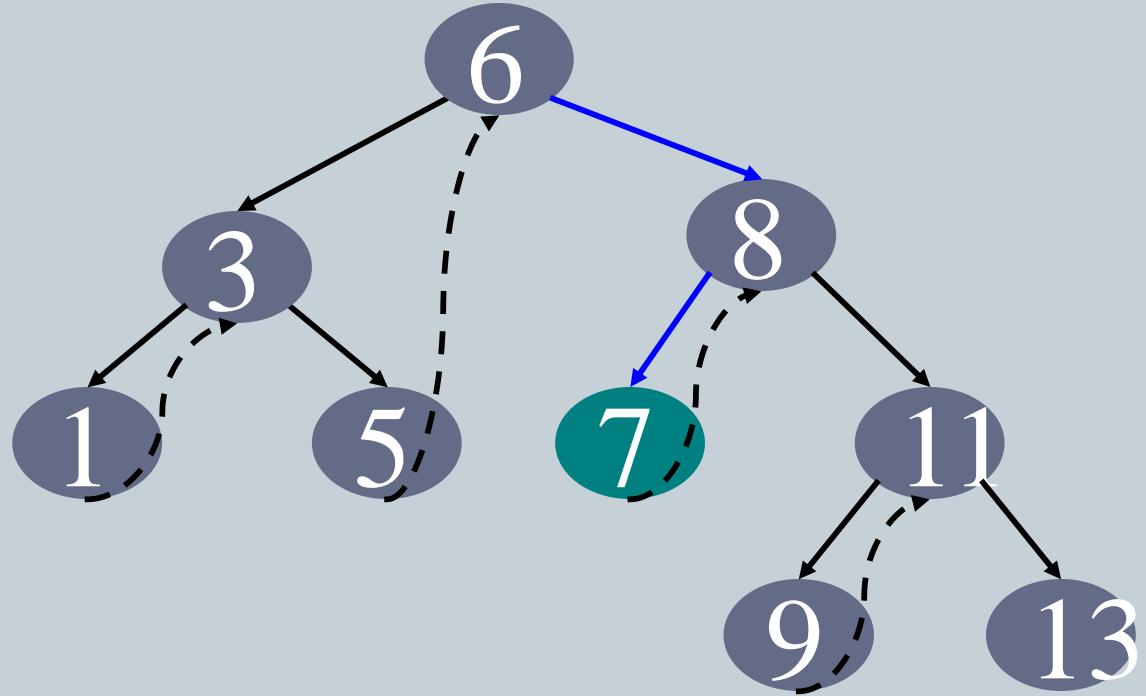


Output  
1  
3  
5  
6

Follow thread to right, print node

# Threaded Tree Traversal

20



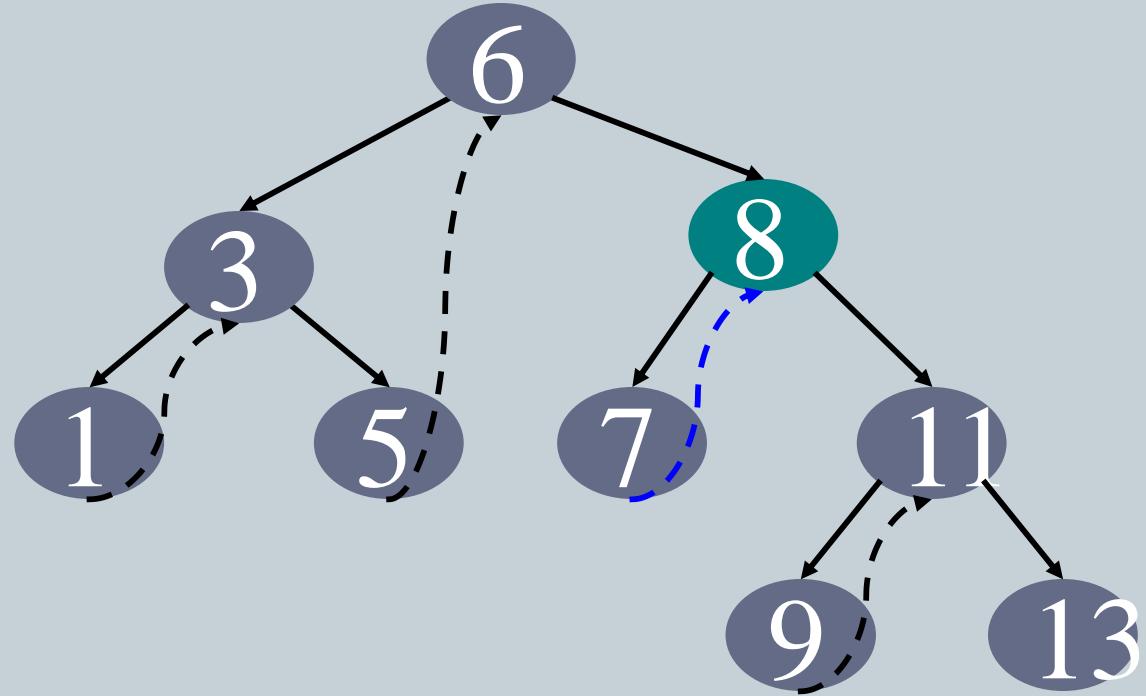
Output

1  
3  
5  
6  
7

Follow link to right, go to leftmost  
node and print

# Threaded Tree Traversal

21



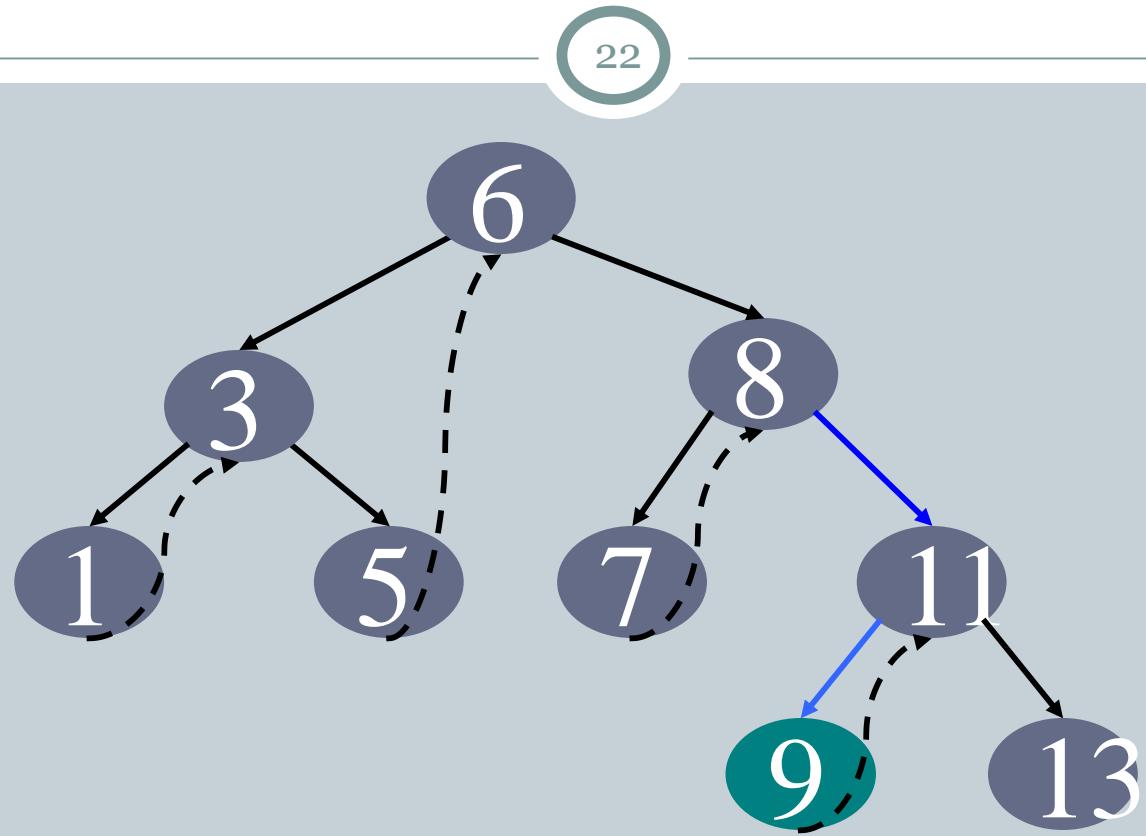
Output

1  
3  
5  
6  
7  
8

Follow thread to right, print node

# Threaded Tree Traversal

22

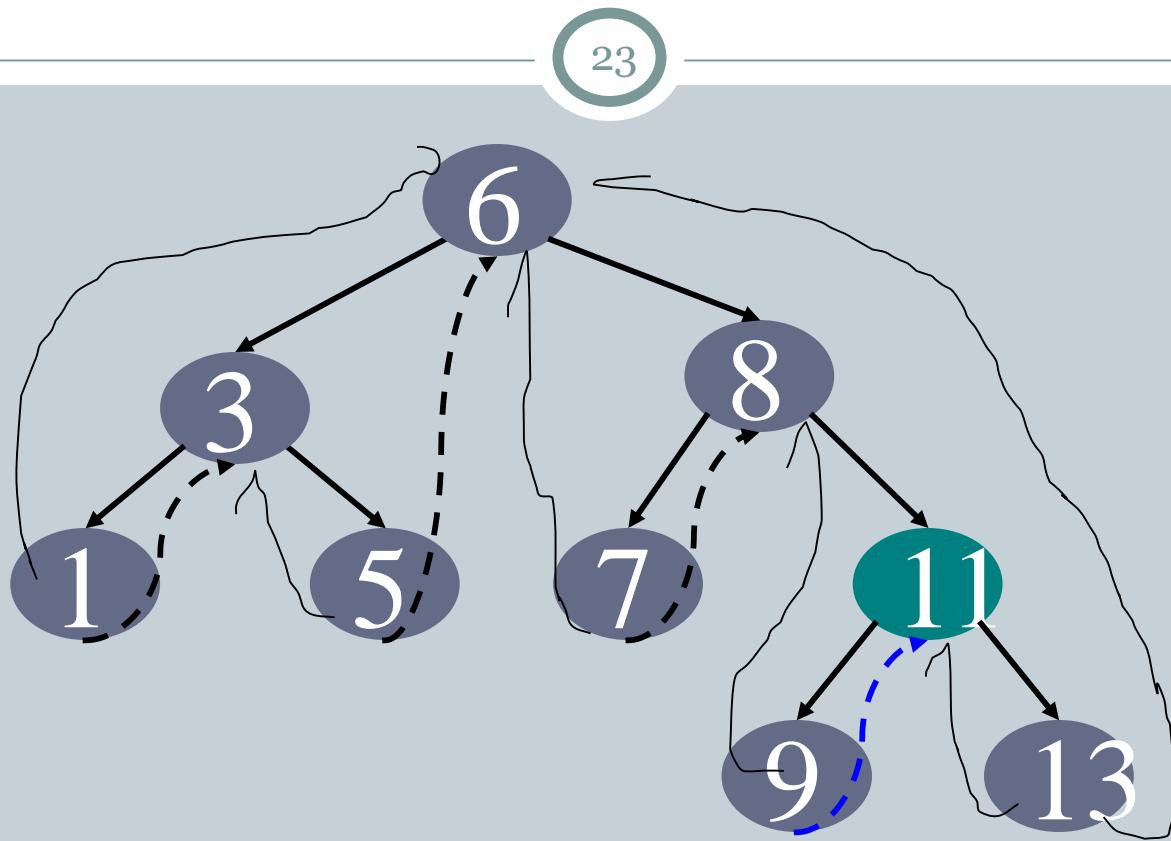


Output

1  
3  
5  
6  
7  
8  
9

Follow link to right, go to leftmost node and print

# Threaded Tree Traversal



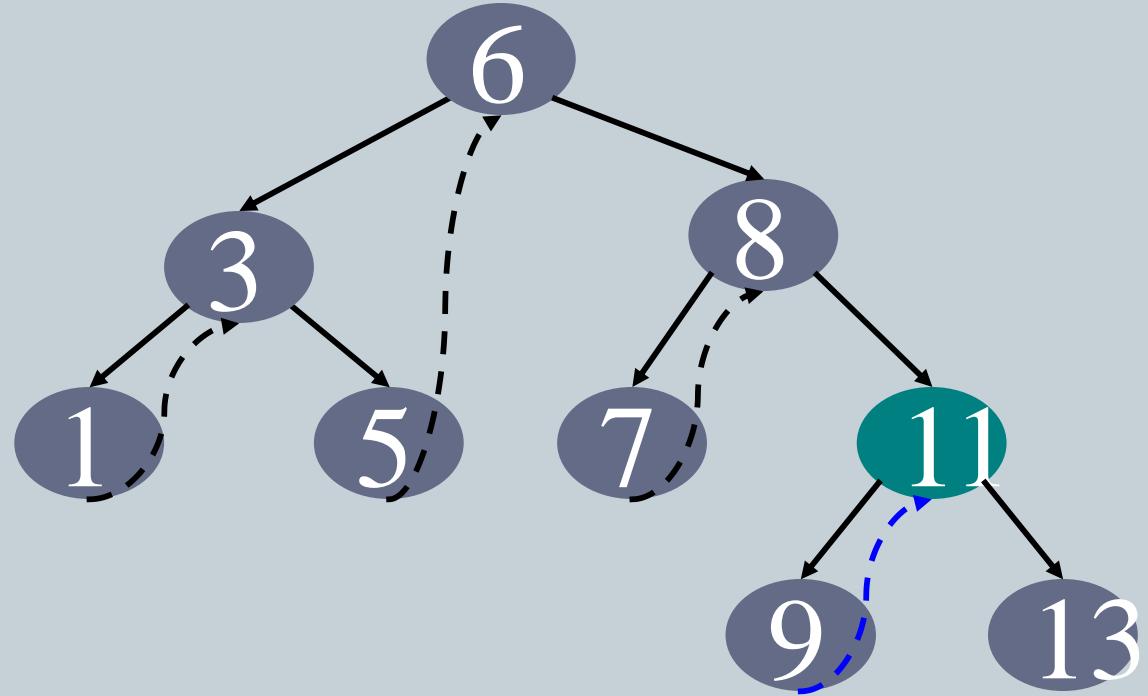
Output

1  
3  
5  
6  
7  
8  
9  
11

Follow thread to right, print node

# Threaded Tree Traversal

24



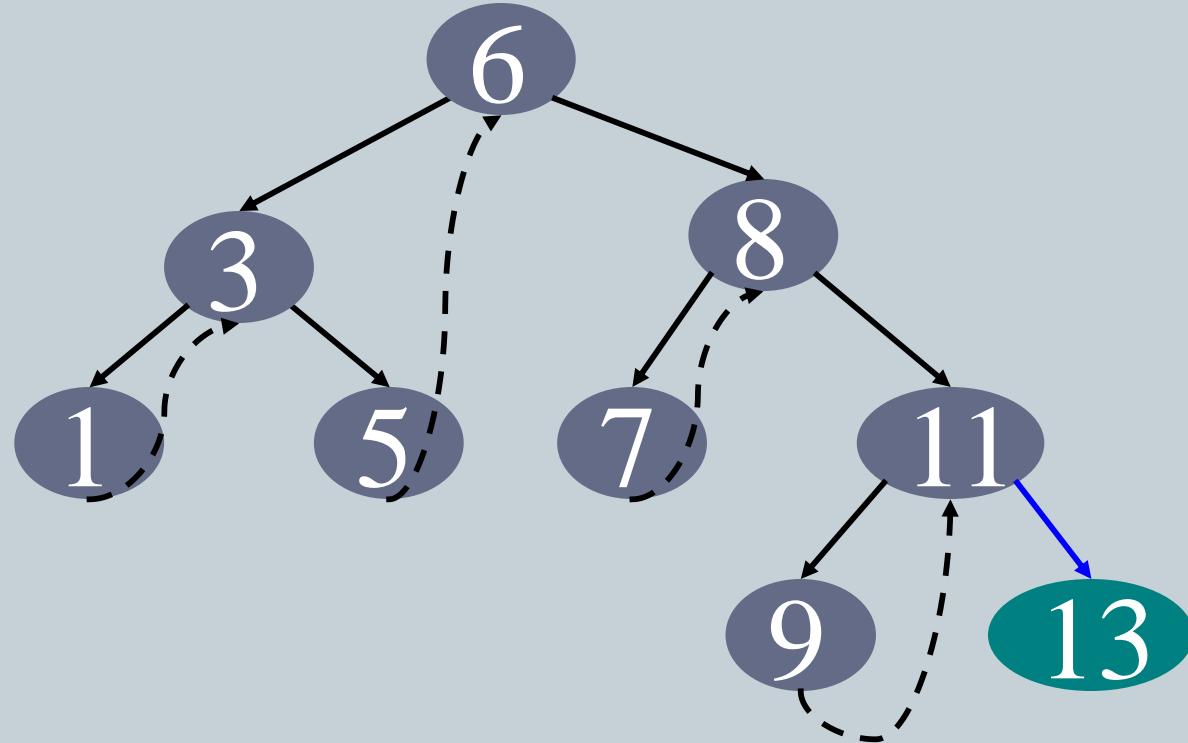
Output

1  
3  
5  
6  
7  
8  
9  
11

Follow thread to right, print node

# Threaded Tree Traversal

25



Output

1  
3  
5  
6  
7  
8  
9  
11  
13

Follow link to right, go to leftmost node and print

# Algorithm of Threaded Inorder

LeftThread	left	9
False	NULL	

```
void inOrder(struct Node *root)
{
    struct Node *cur = leftmost(root);
    while (cur != NULL)
    {
        printf("%d ", cur->data);
        if (cur->rightThread)
            cur = cur->right;
        else // Else go to the leftmost child
             // in right subtree
            cur = leftmost(cur->right);
    }
}
```





## AVL TREE(HEIGHT BALANCED TREE)

# What should be Learnt?



- 1) What is an AVL TREE?
- 2) What is meaning of Height Balanced, Balance factor (-1, 0, 1)?
- 3) How/ When does an AVL becomes Unbalanced
- 4) 4 cases of Unbalance
- 5) Result of balance of unbalanced AVL in each of the 4 cases  
(using a thumb rule)
- 6) How to Implement the Required Result in each of the 4 cases



- 1) What is an AVL TREE?
- 2) What is meaning of Height Balanced, Balance factor?

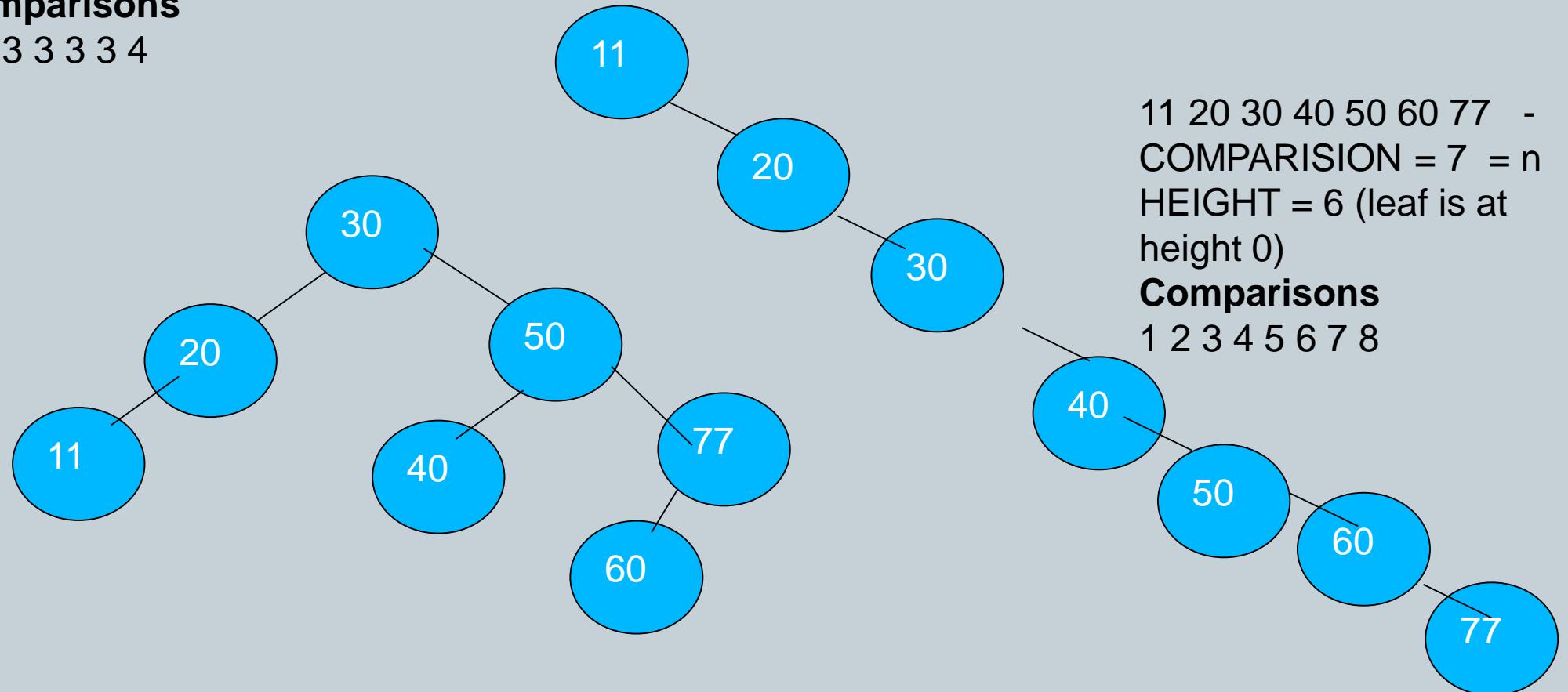
30 20 50 11 40 77 60

HEIGHT = 3



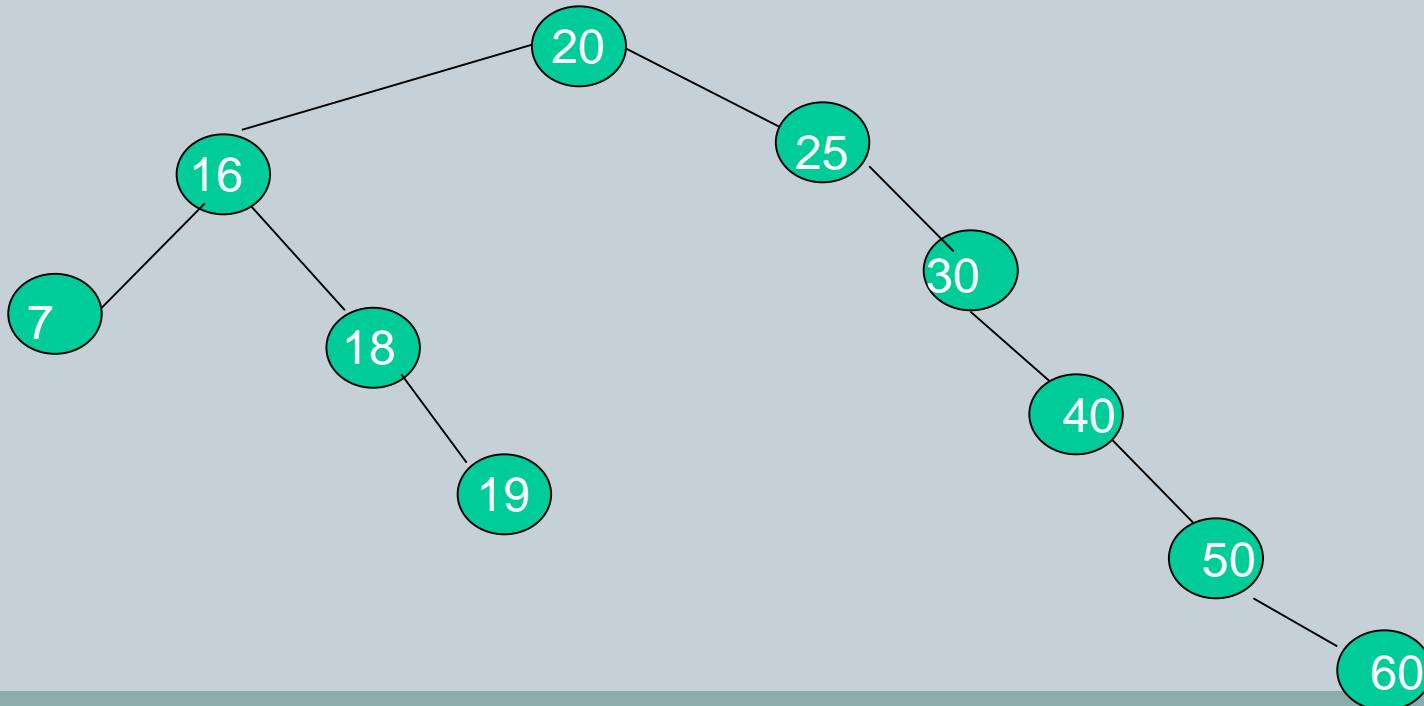
### Comparisons

1 2 3 3 3 3 4



- HEIGHT – (is BAD) is Costly in Binary Search TREE (for some applications)
  - So we need to BALANCE the B Search TREE
- BALANCED TREE : A tree with all nodes having difference in height of their LST and RST as = 1 / 0 / -1

- Many operations of trees depend on the **depth** (or Height) of the tree.
- We don't want trees with nodes which have **large height**
- This can be attained if both sub trees of each node **have roughly the same height.**



# How can we reduce the HEIGHT of a BST?



Ans: While creation of BST itself, we rotate (change positions) of the nodes so that it is AVL tree

- AVL tree is a binary search tree where the height of the two sub trees of a node **differs by at most one**

34

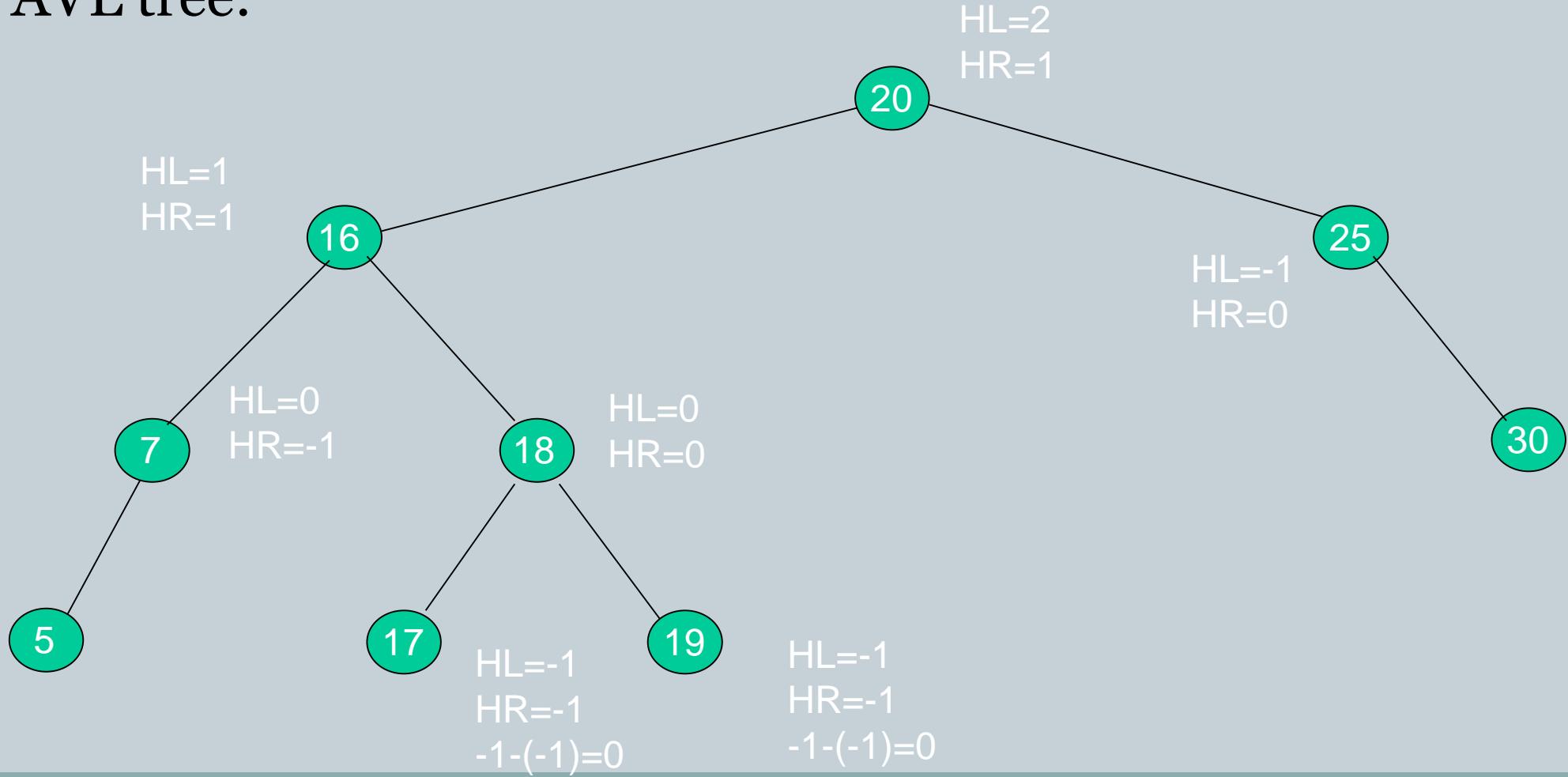
- For the tree to be **balanced** the **Balance factor** of tree should be **-1,0,1**
- The **Balance Factor** is calculated as **HL – HR**
- An **AVL tree** is one of the balanced binary search tree.
- Named after their inventors, **Adelson-Velskii** and **Landis**, they were the first **dynamically** balanced trees whose height differ by at most 1

# AVL Trees

(Adelson – Velskii – Landis) [Leaf is at Height 0]

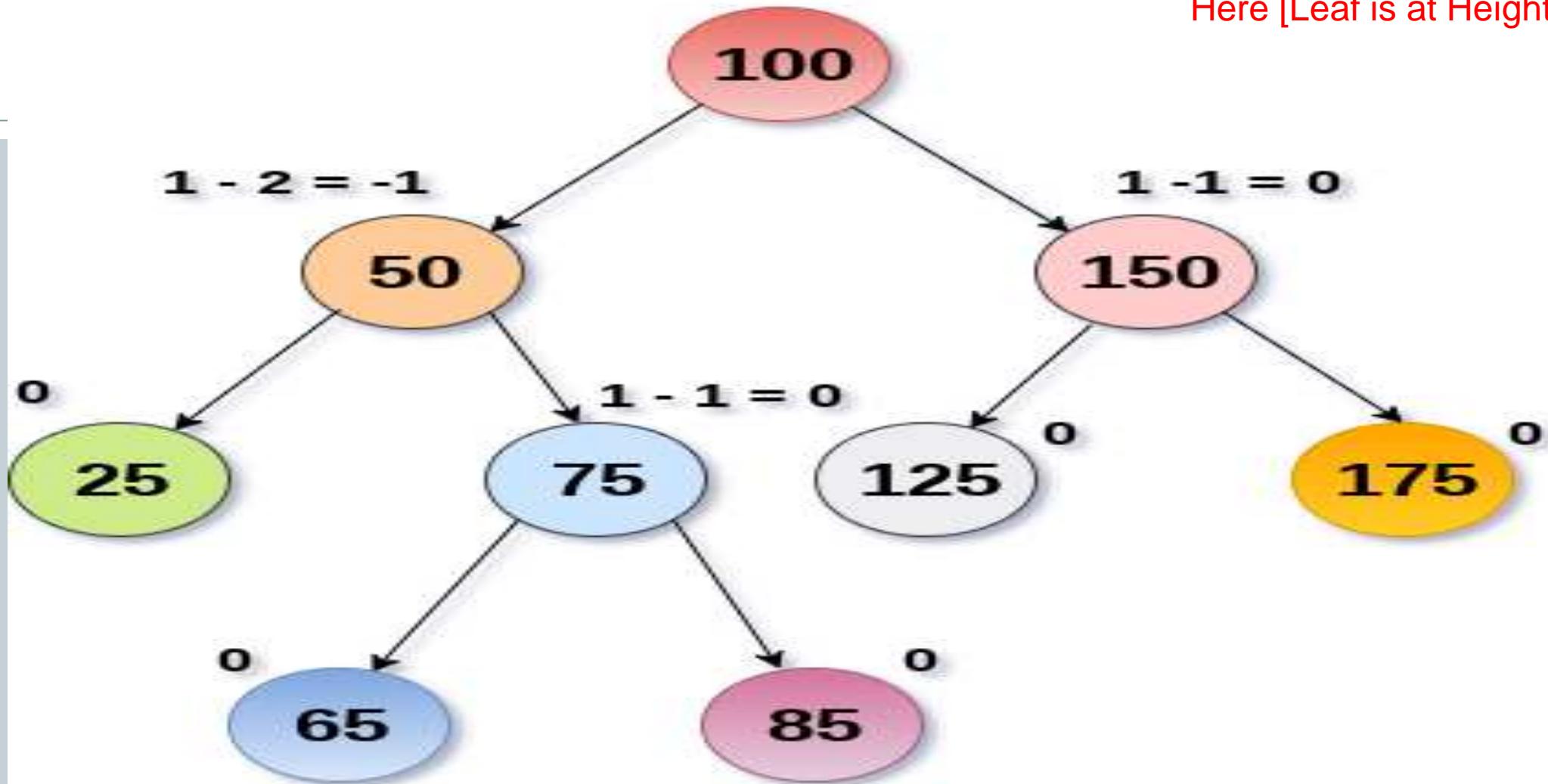
35

AVL tree:



$$3 - 2 = 1$$

Here [Leaf is at Height 1]



**AVL Tree**

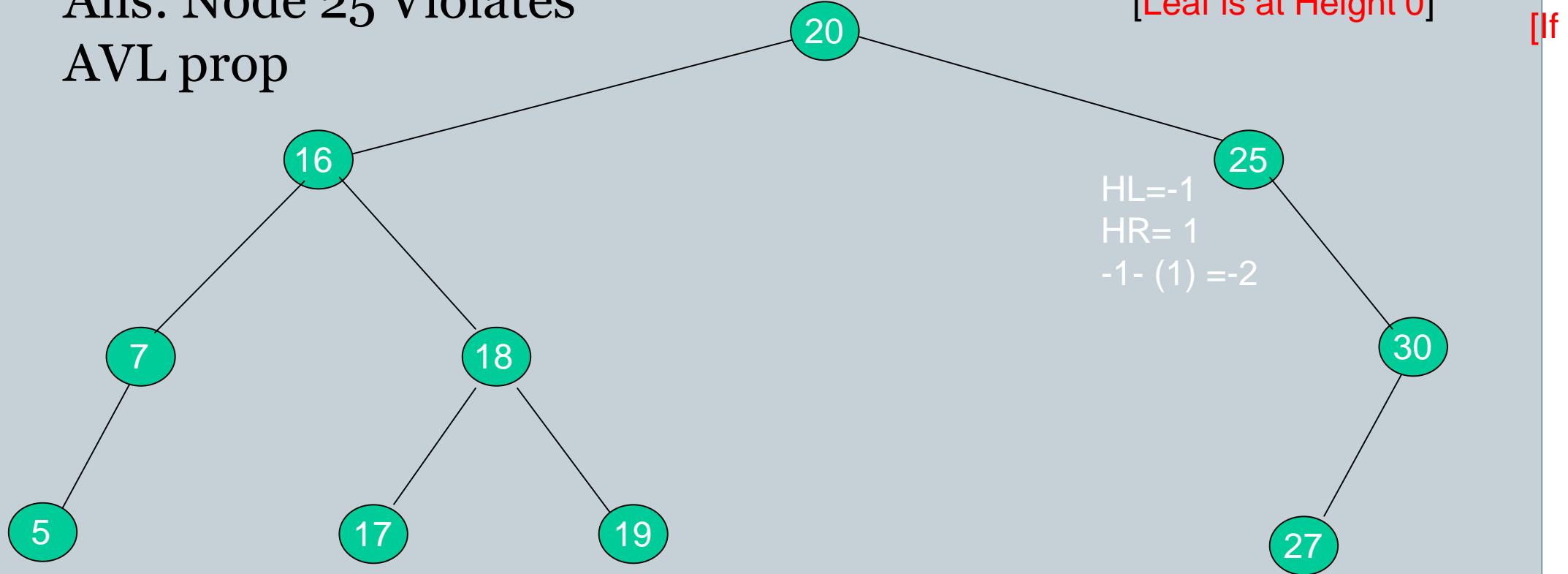
# AVL Trees

37

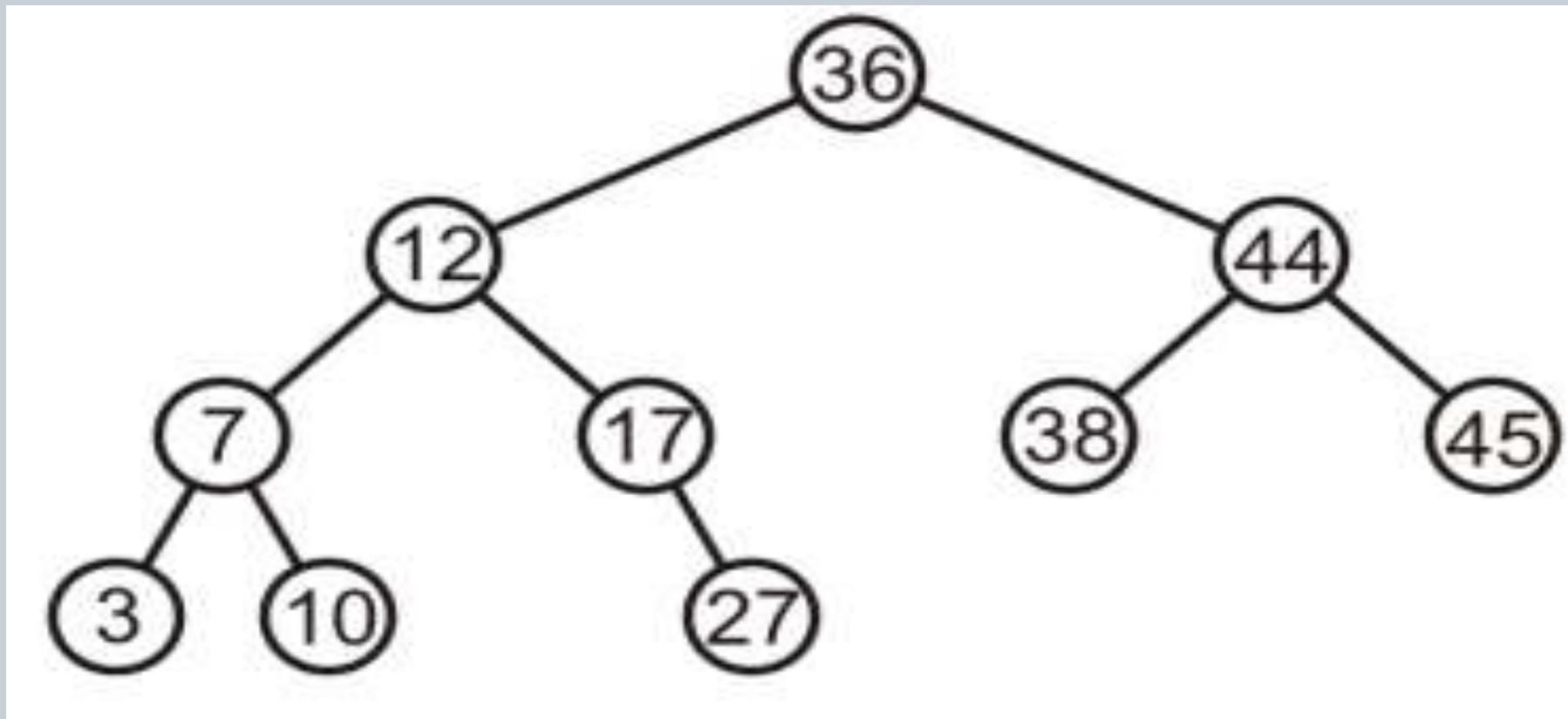
HL-HR > 1 ?  
Violates AVL prop

Not AVL tree: WHY?

Ans: Node 25 Violates  
AVL prop



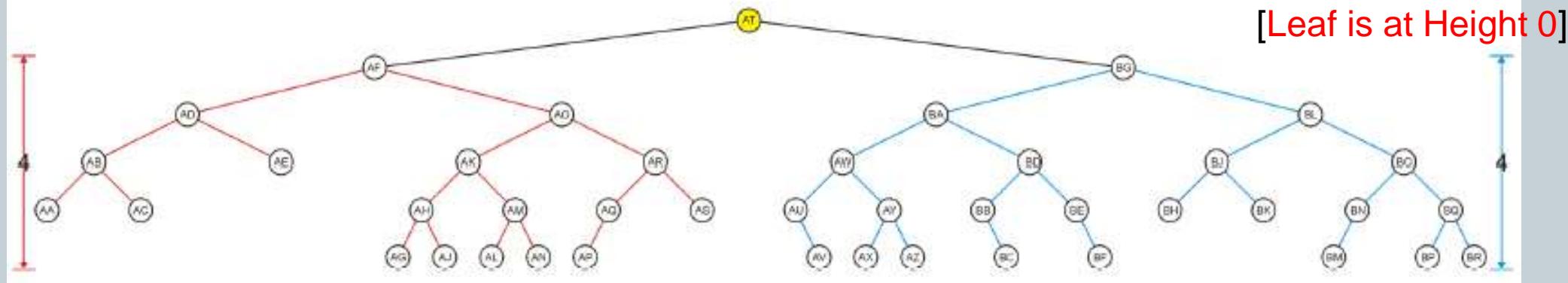
What is height of LST, RST of root? Is this AVL Tree?



# Is this AVL Tree?

The root node is AVL-balanced:

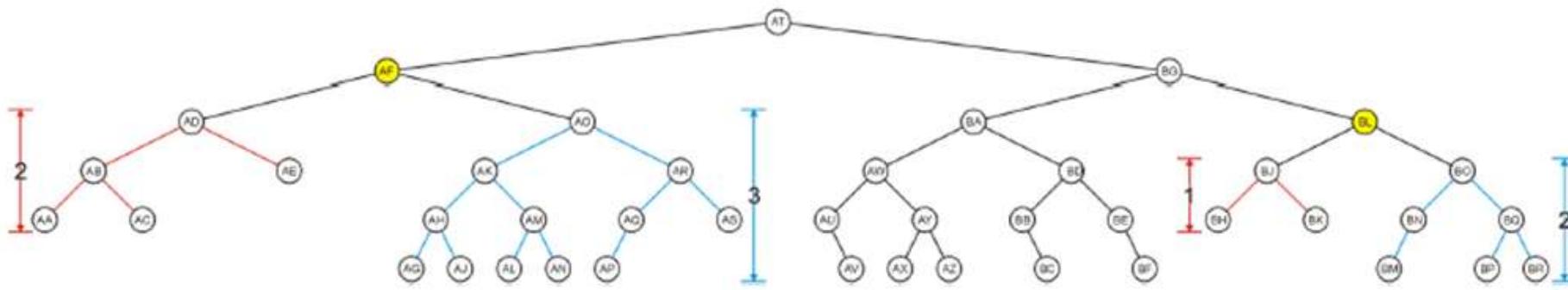
- Both sub-trees are of height 4:



[Leaf is at Height 0]

All other nodes are AVL balanced

- The sub-trees differ in height by at most one





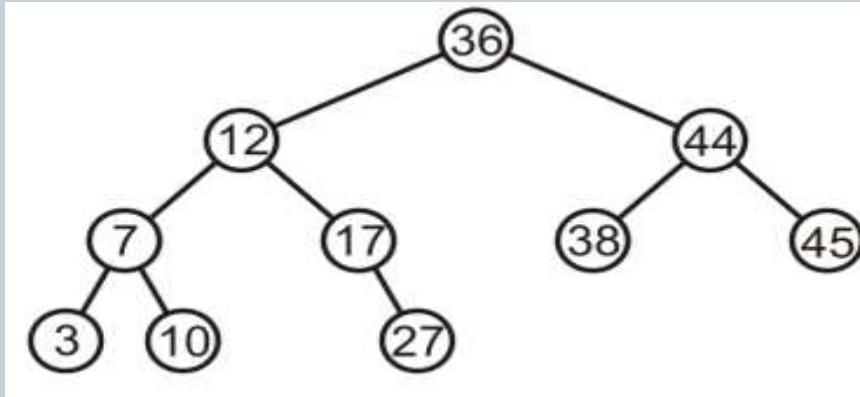
3) How an AVL becomes Unbalanced

Ans: Specific INSERTIONS

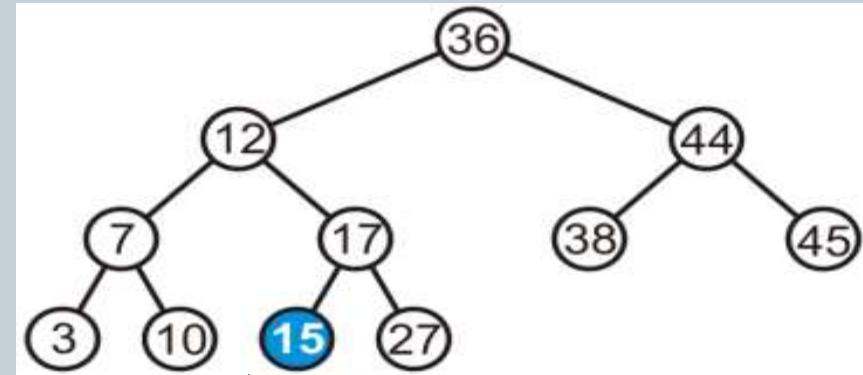
Inserting 15 in AVL (is the new one AVL?)  
Inserting 42 in AVL (is the new one AVL?)



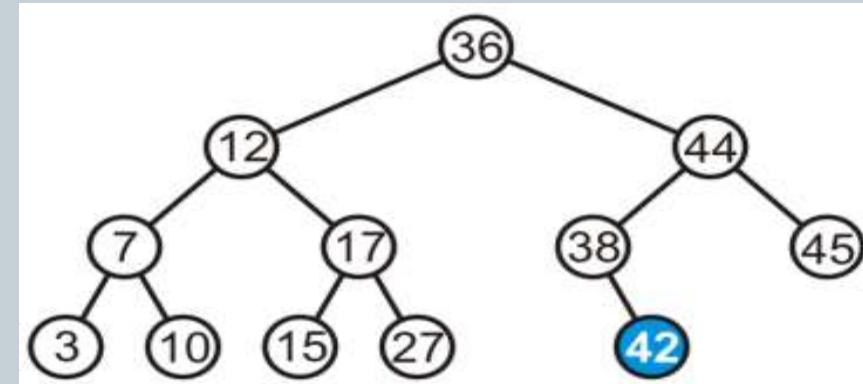
EXAMPLE of : Not Unbalanced after insertion



15



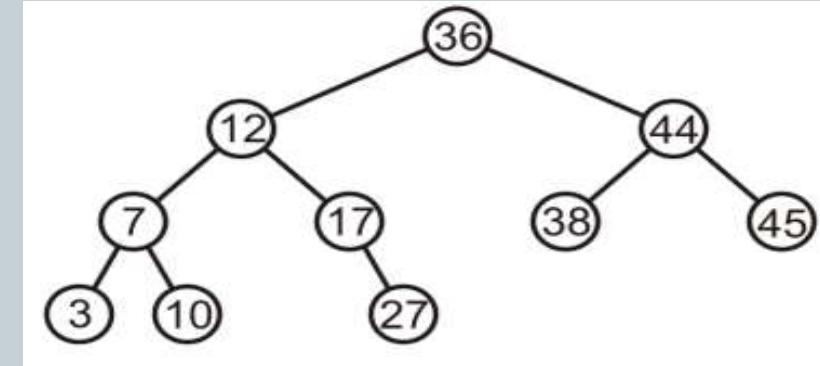
42



# When will an insertion cause an **imbalance**?

- The heights of the sub-trees must differ by 1 (e.g. LST (2) and RST(1) of 36)

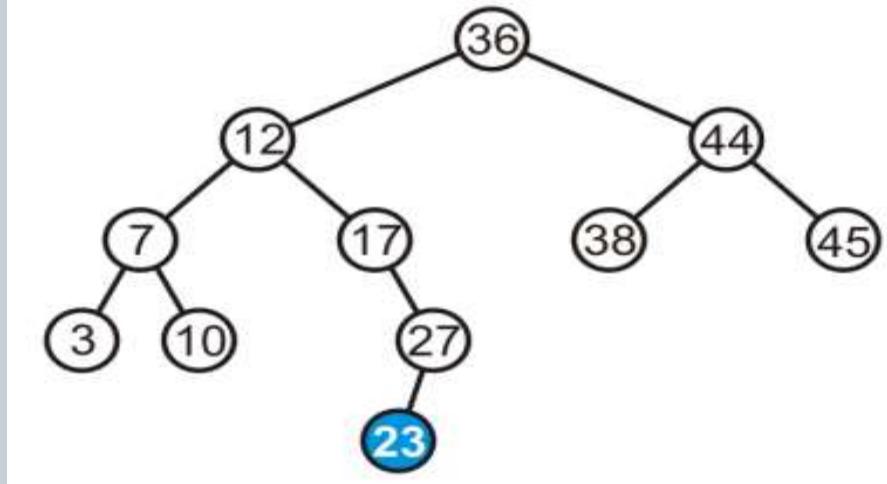
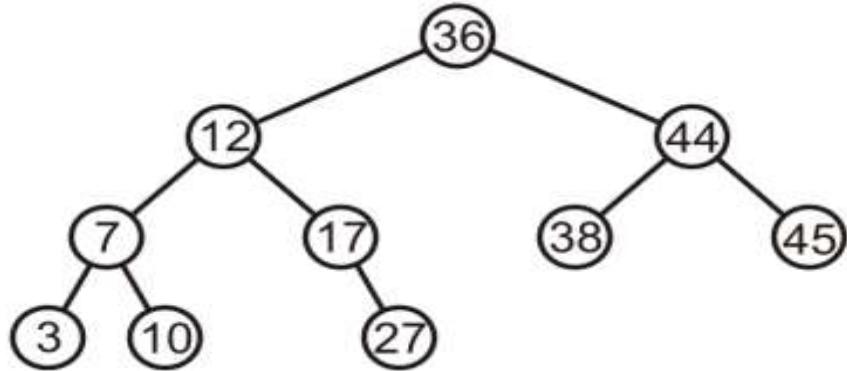
AND



- The **one** insertion must **increase the height of the deeper sub-tree** by 1
  - (here Deeper subtree is rooted at 12)

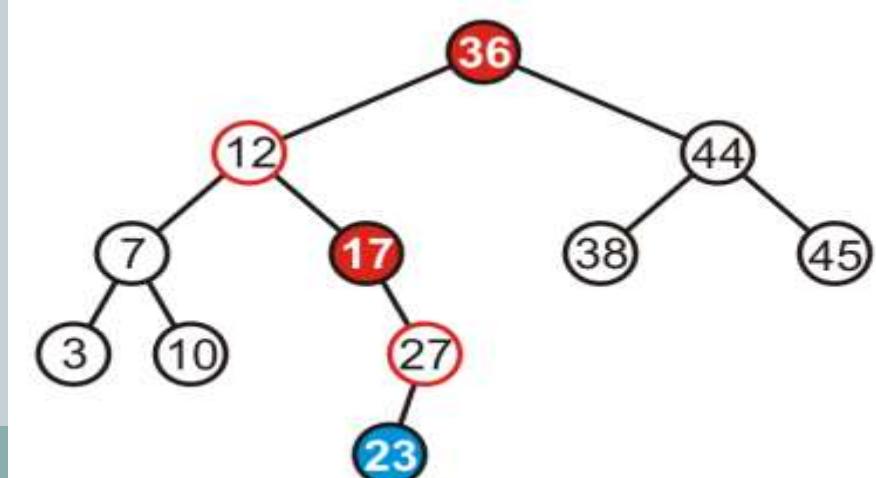
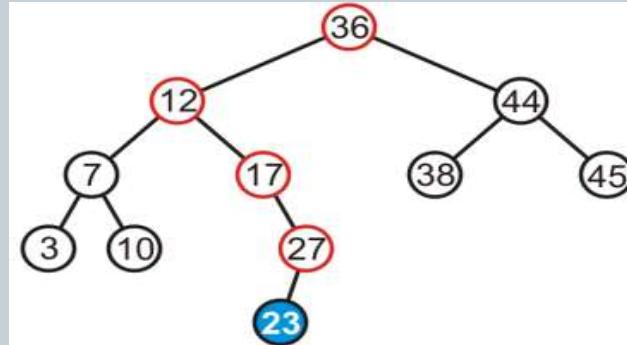
WHICH Insertions will cause an **imbalance**? (Ans: Samples : 2,4,8,11, (15??), 23,29)

## Inserting 23 in AVL (is the new one AVL?)



The heights of each of the sub-trees from here to the root are increased by one

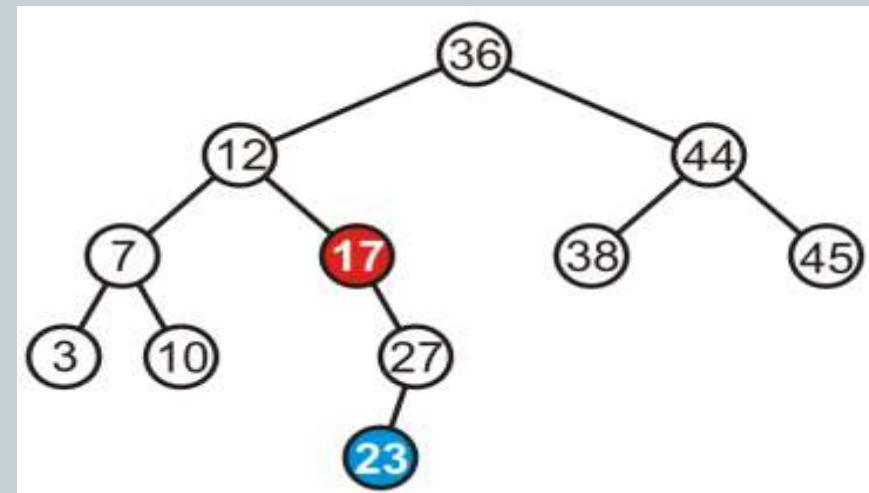
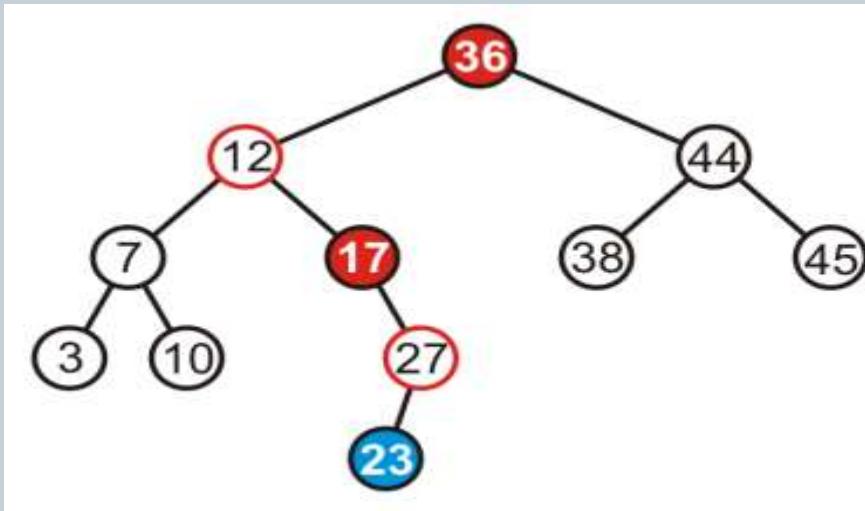
However, only two of the nodes are unbalanced: 17 and 36



# Maintaining Balance



- However, we only have to fix the imbalance at the **lowest node**





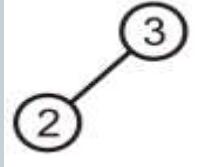
## 4) 4 cases of UNBALANCE

# 4 CASES of IMBALANCE

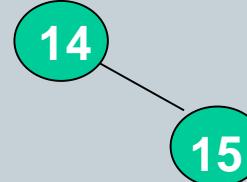
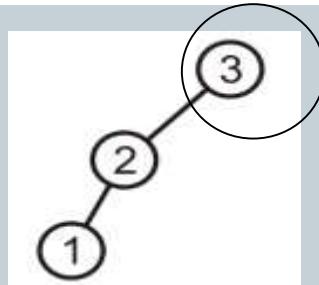


- (1) LL **case**(Left of Left Subtree)
- (2) RR **case**(Right of Right Subtree)
- (3) RL **case**(Right of Left Subtree)
- (4) LR **case**(Left of Right Subtree)

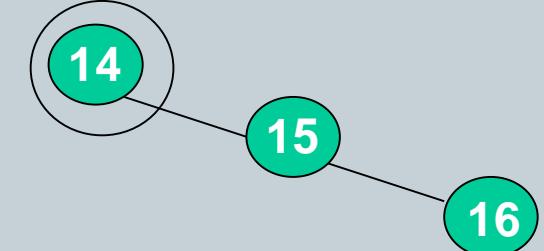
## 4 CASES of IMBALANCE



Insert 1

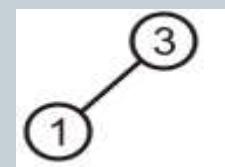


Insert 16

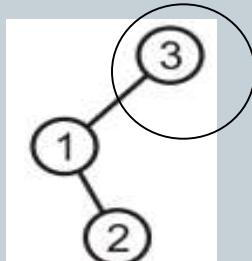


**CASE (1) LL case**(Left of Left Subtree)

**CASE (2) RR case**(Right of Right Subtree)

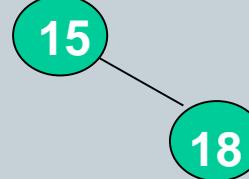


Insert 2

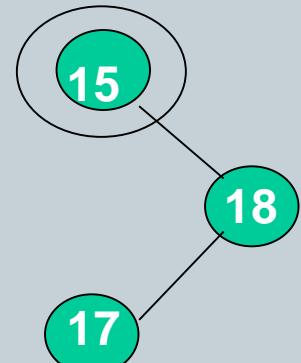


**CASE (3) RL case** (Right of Left Subtree)

ELBOW



Insert 17



**CASE (4) LR case** (Left of Right Subtree)

Note: RL is a Mirror case of LR, and similarly LR is a mirror case of RL

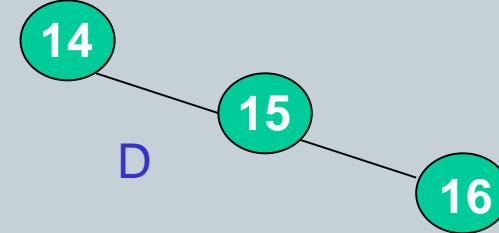
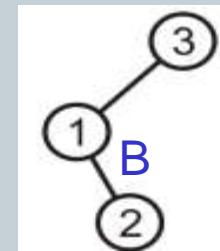
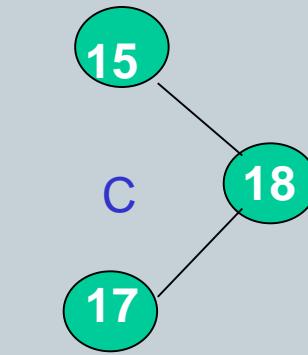
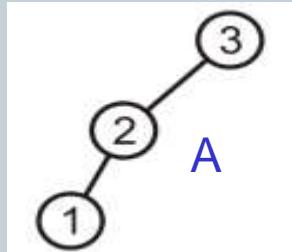


Recognize the 4 cases (LL, RR, RL, LR) in the unbalanced AVL (due to **one** insertion)

Q) Recognize which of the 4 cases apply in A, B, C , D

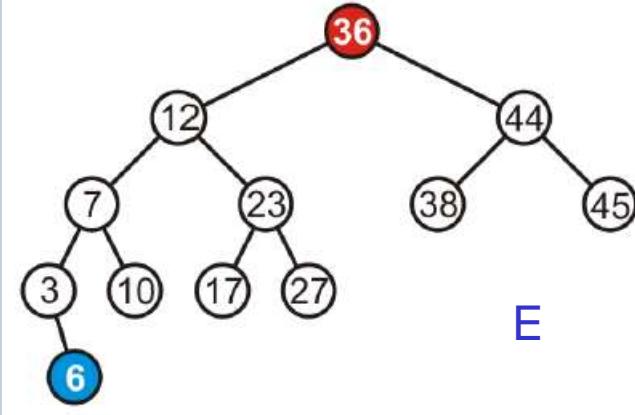


A- LL ,B- RL , C- LR , D-RR

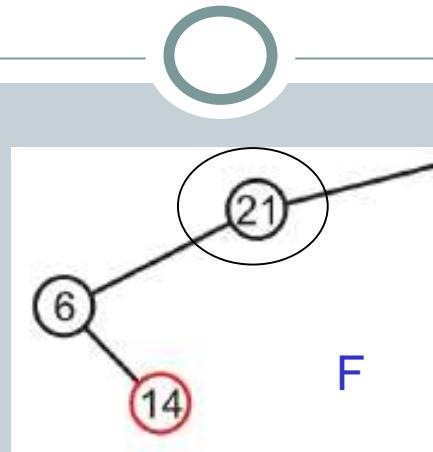


Q) Recognize which of the 4 (LL,RR,RL,LR case) cases apply in E, F, G, H, I

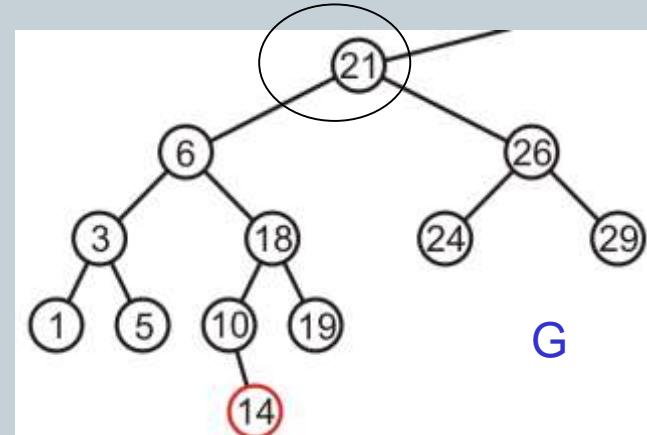
E = LL , F = RL



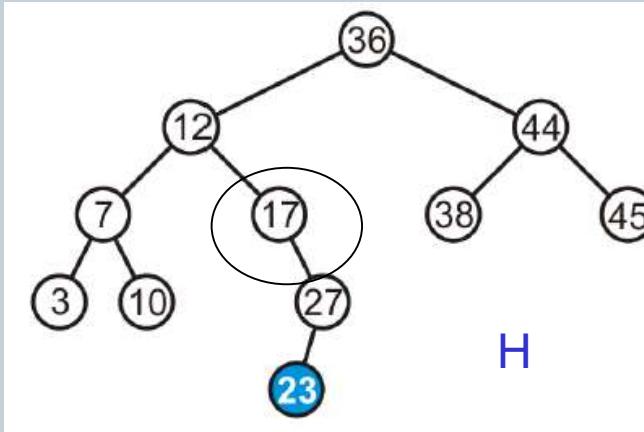
E



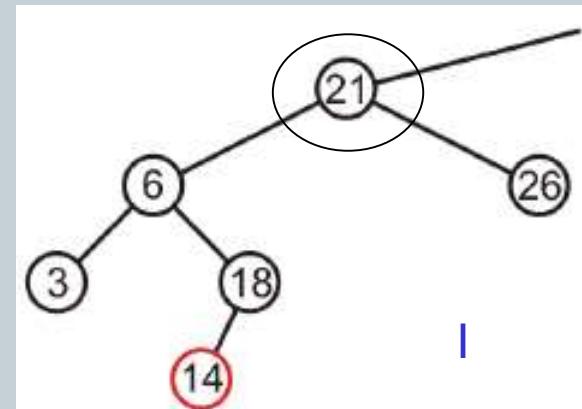
F



G

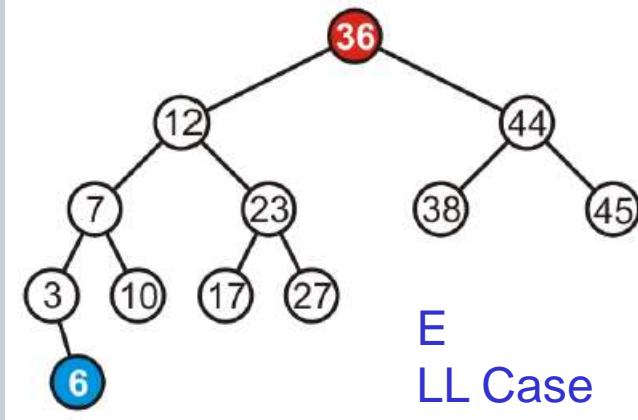


H

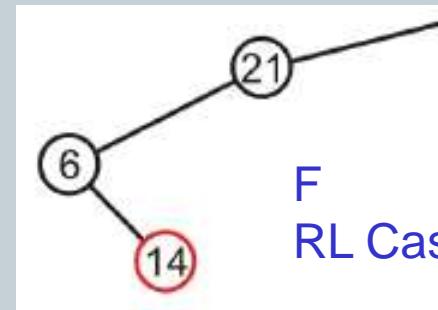


I

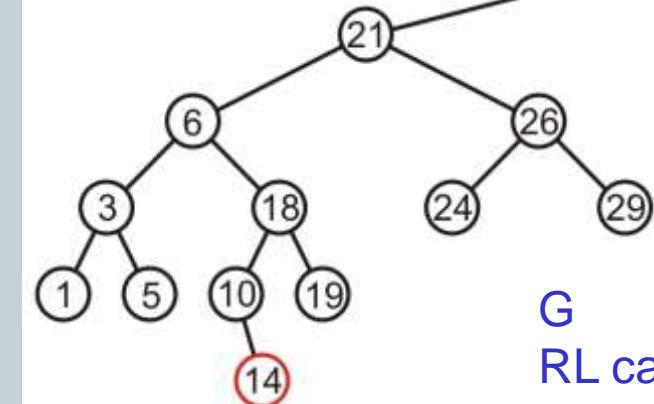
Q) Recognize which of the 4 cases apply in E, F, G, H, I



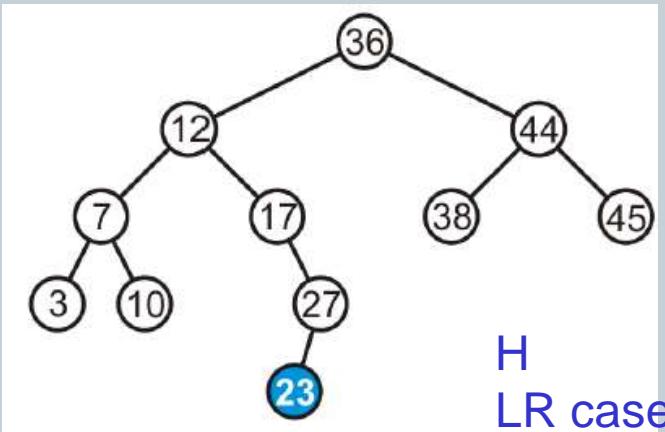
E  
LL Case



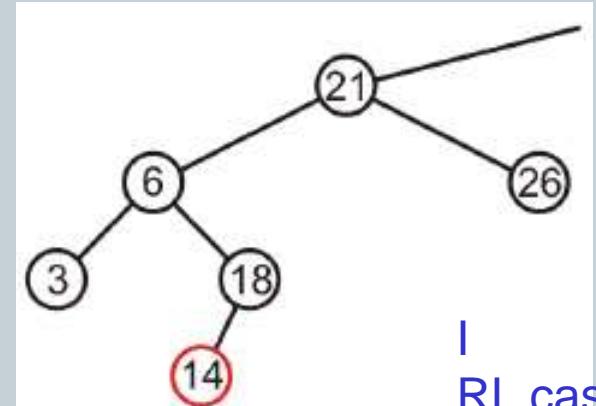
F  
RL Case



G  
RL case



H  
LR case

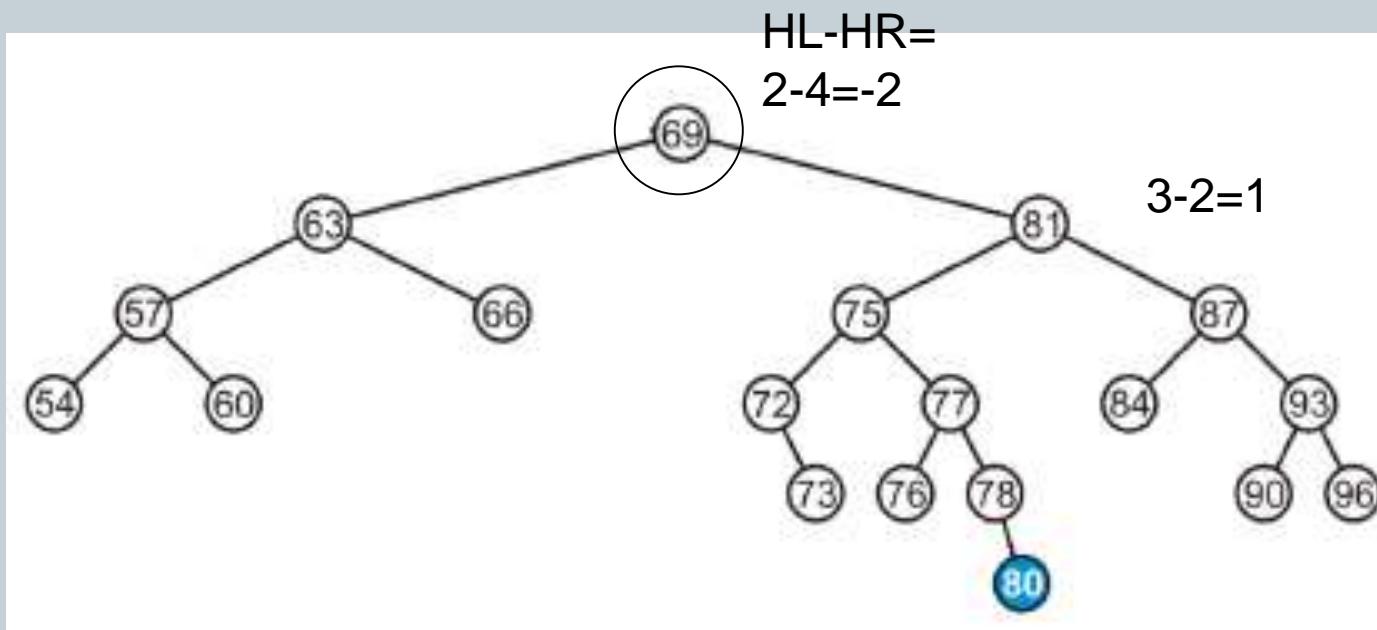


I  
RL case

E = LL , F = RL , G = RL , H = LR ,

Q) Recognize which of the 4 cases apply in J

Violated Node =  
Ans: J= LR?





5) Result of balance of unbalanced AVL in each  
of the 4 cases

# Result of Balancing in 4 cases (using some thumb rule)

In the 4 CASES:

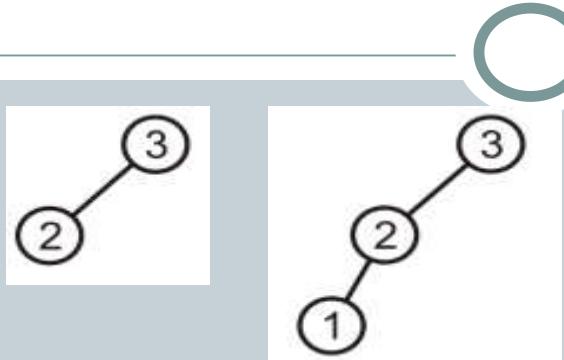
- (1) LL (Left of Left Subtree) – ??
- (2) RR (Right of Right Subtree) – ??
- (3) RL (Right of Left Subtree) – ??
- (4) LR (Left of Right Subtree) – ??



# CASE 1 : LL-> LEFT OF LEFT ST (RIGHT ROTATION)

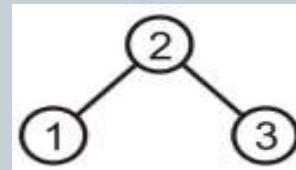
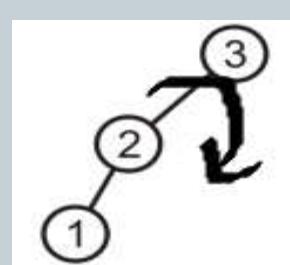
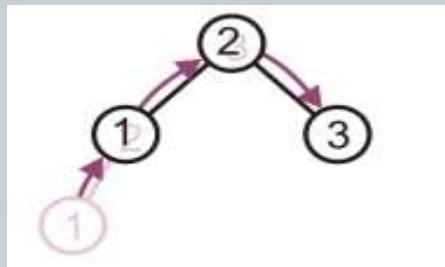
## How to correct imbalance? An Example (CASE 1: LL -> LEFT OF LEFT Case {RIGHT ROTATION})

- Add 1 now

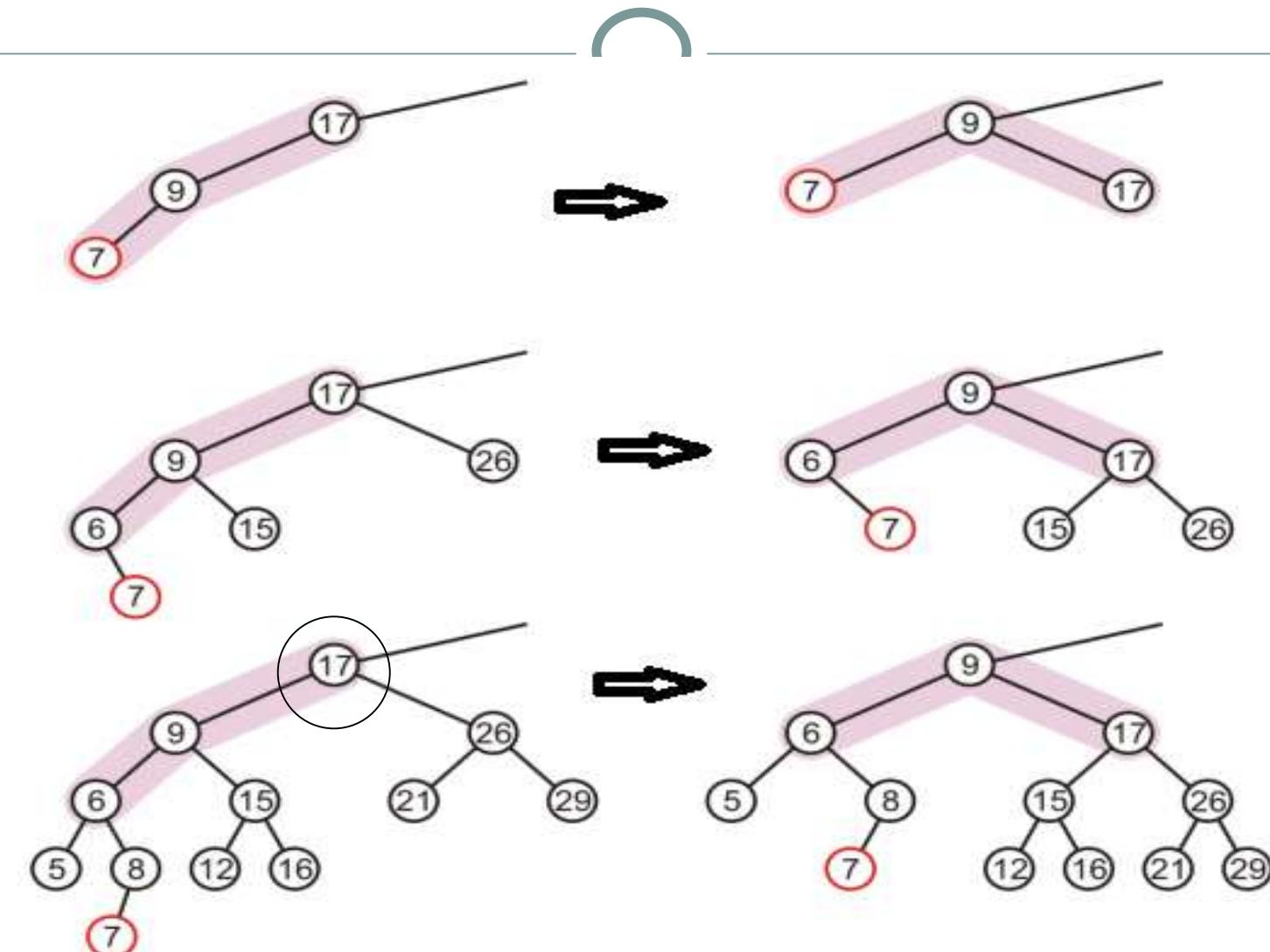


How to correct?

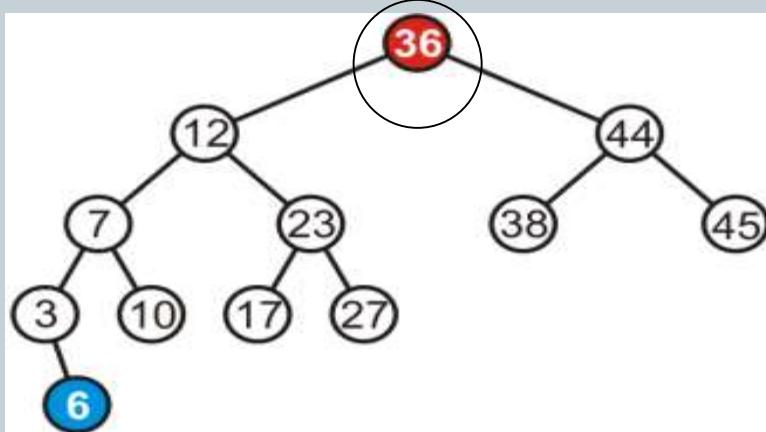
Promote 2 as the root, make 3 to be 2's right child, and 1 remains the left child of 2



LL = LEFT of LEFT ST (Right Rotation)



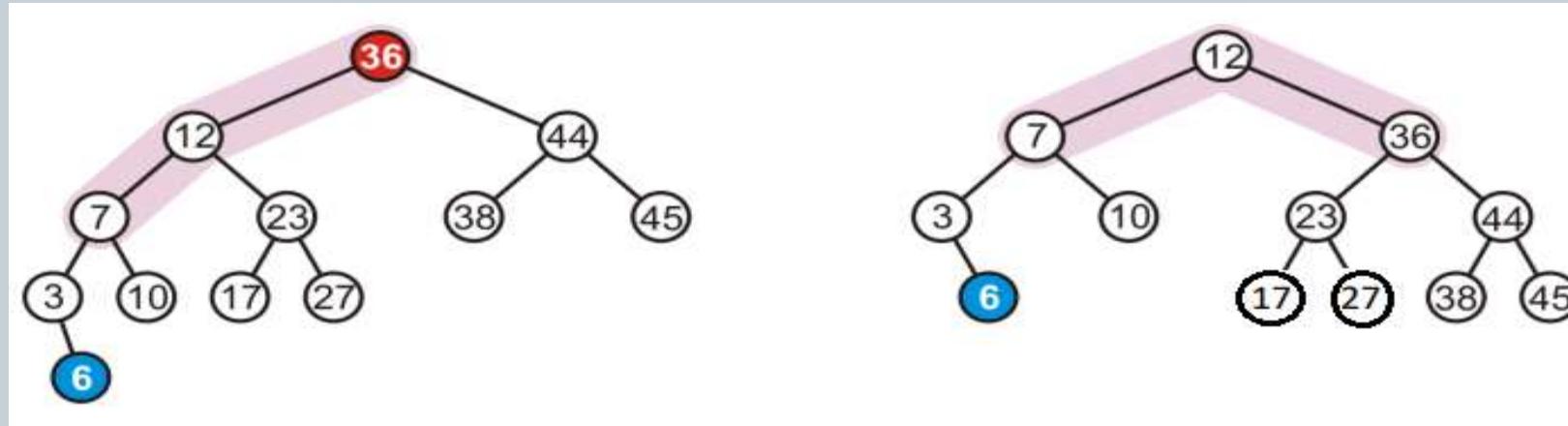
What can be done for BALANCE?



ANS? Recognize Case  
Case is LL

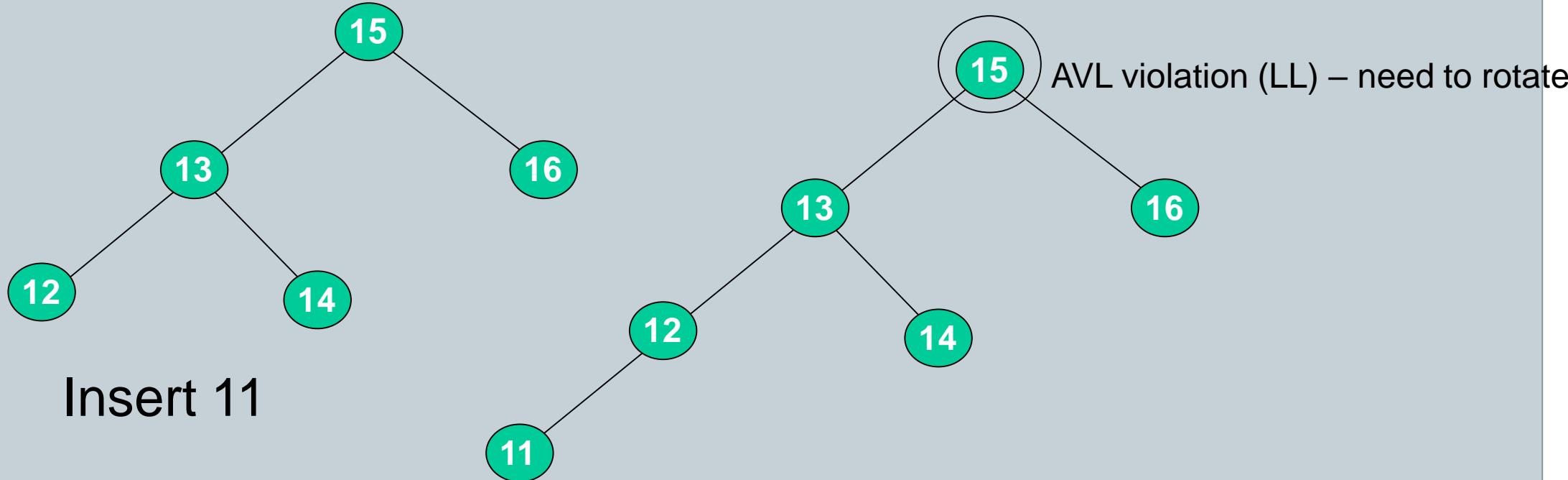
What can be done for BALANCE?

Ans: LL – Right Rotation



# AVL Tree Rotations

Single rotations:

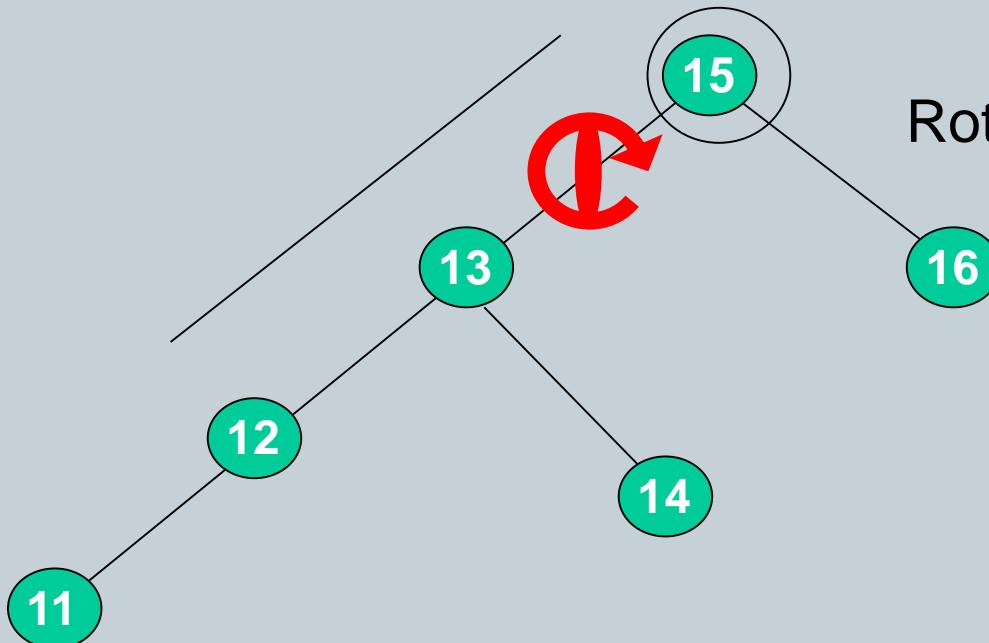


Insert 11

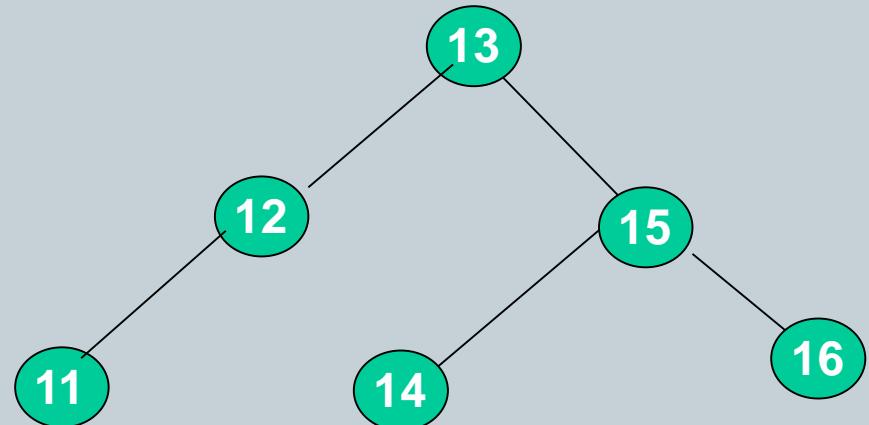
# AVL Tree Rotations

61

LL Case: Single rotations:



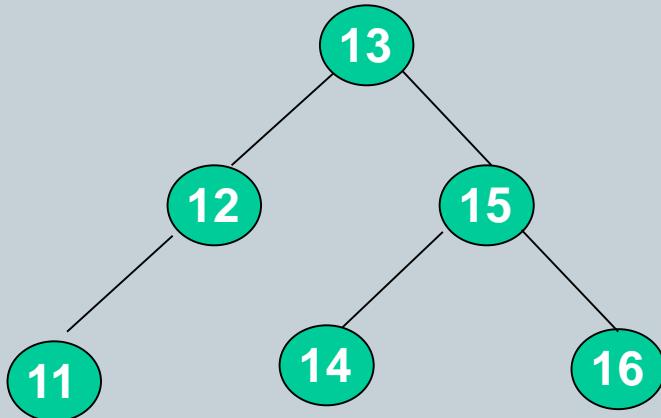
Rotation type: Right Rotation



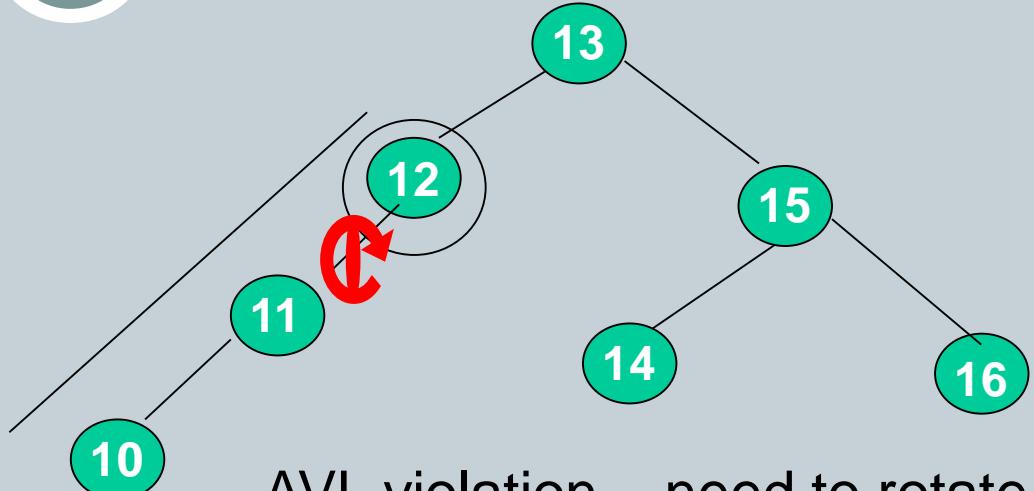
# AVL Tree Rotations

62

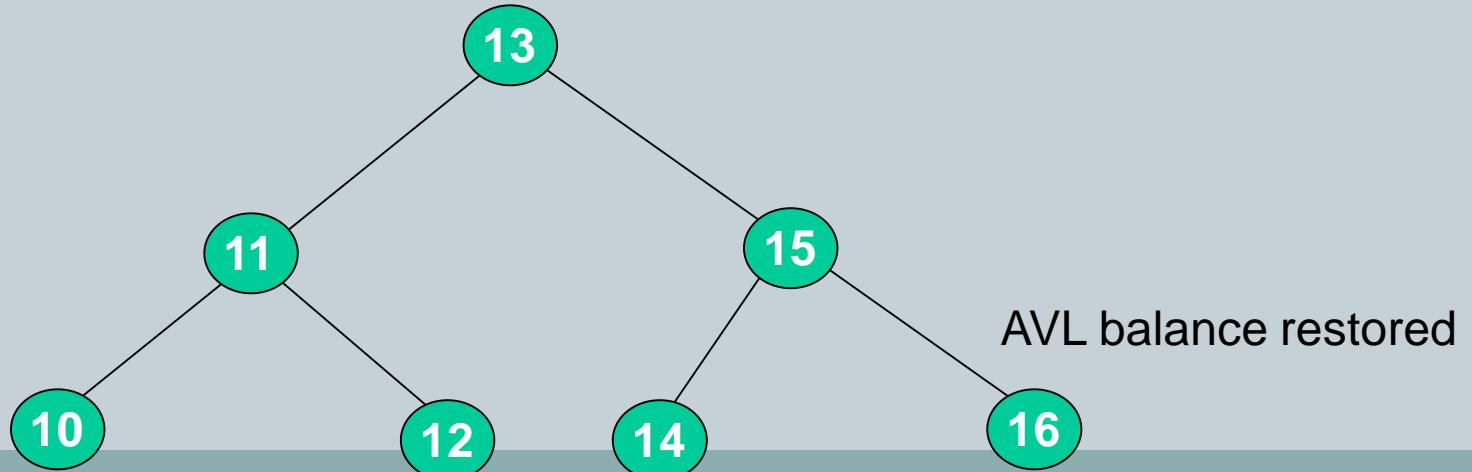
**Single rotations:**



Insert 10



AVL violation – need to rotate



AVL balance restored



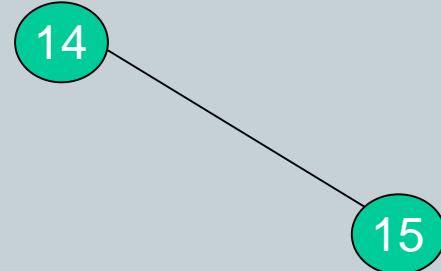
## CASE 2 : RR (MIRROR OF CASE 1 – LL CASE)

# AVL Tree Rotations

64

Single rotations: insert 14, 15, 16

- First insert 14 and 15:



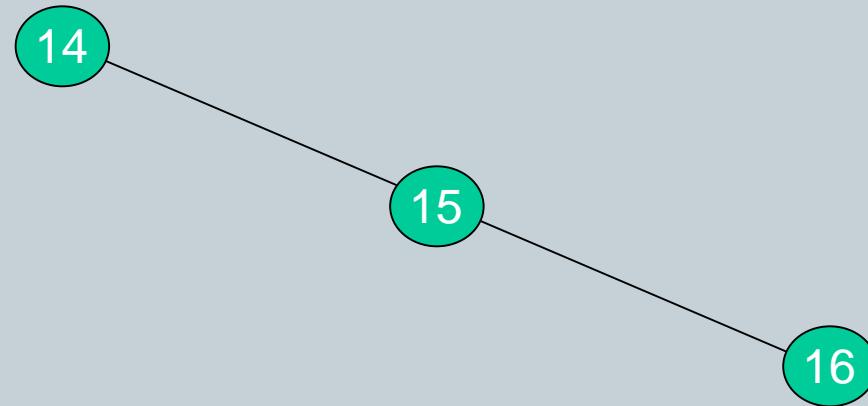
- Now insert 16.

# AVL Tree Rotations

65

Single rotations:

- Inserting 16 causes AVL violation:



- Need to rotate.

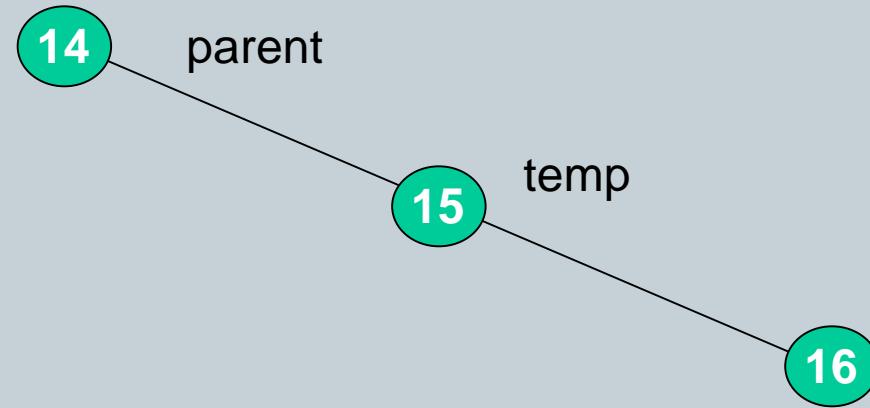
# AVL Tree Rotations

Single rotations:- Right of Right Rotation (Left Rotation)

**(Insert element to right of right subtree )**

Rotate to left

- Inserting 16 causes AVL violation:



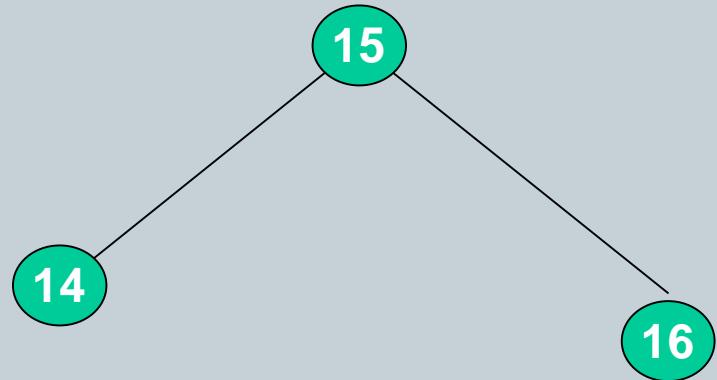
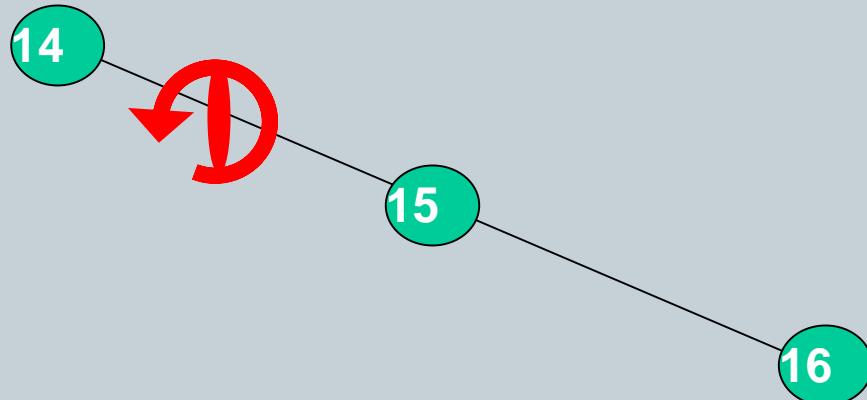
- Need to rotate.

# RR Case : AVL Tree Rotations

67

Single rotations:

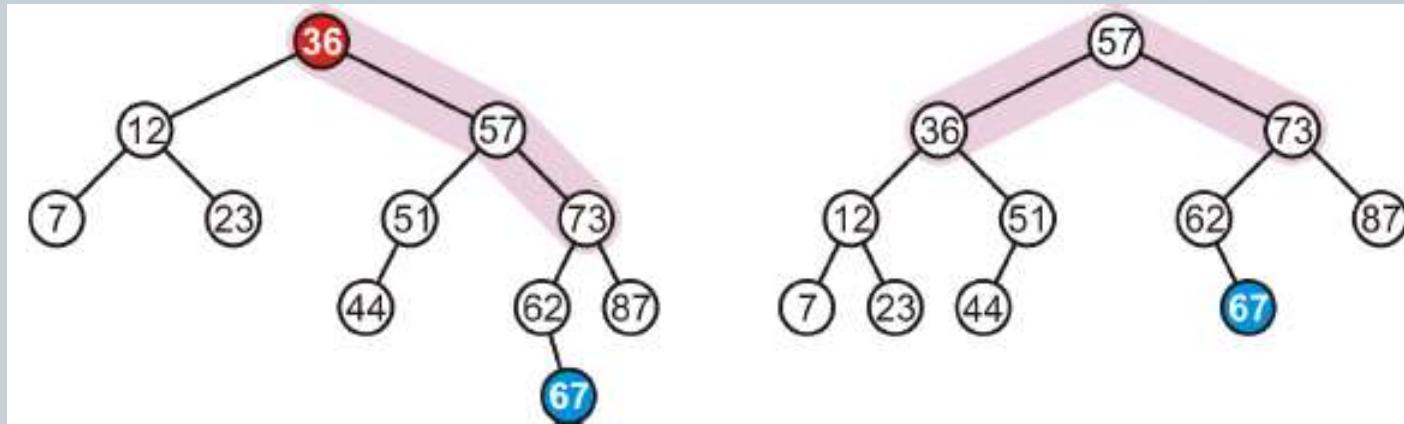
- Rotation type: Left Rotation



# RR (Mirror / Symmetric to LL) (Left Rotation)



Rotation type: **Left Rotation**



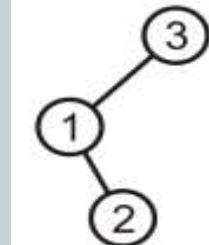
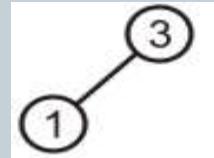


## CASE 3 : RL -> RIGHT OF LEFT ST CASE

# How to correct imbalance? An Example (RL)

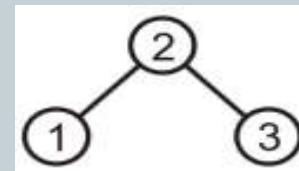
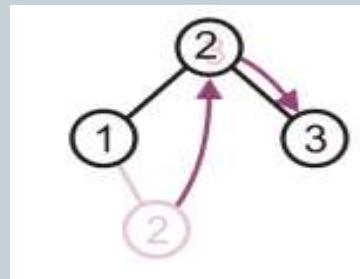
ELBOW

- Add 2 now



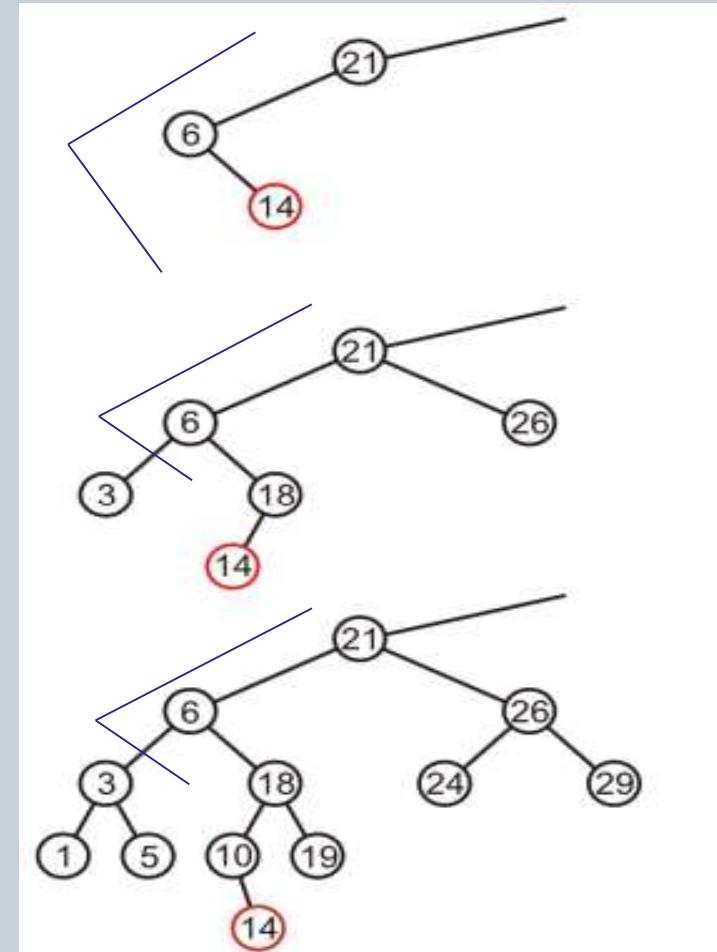
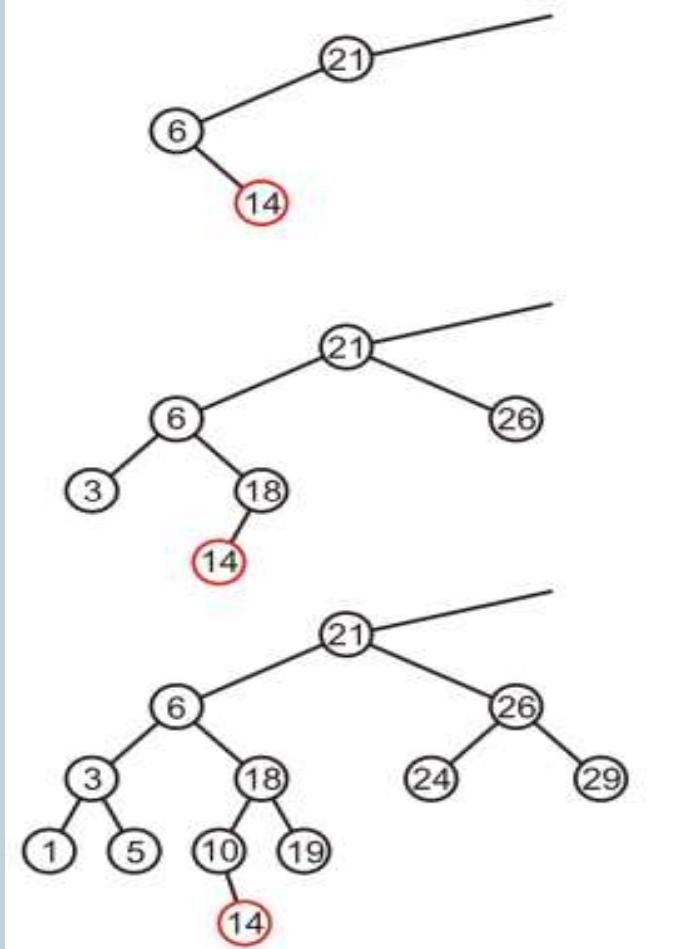
How to correct?

Promote 2 as the root, make 3 to be 2's right child, and 1 remains the left child of 2

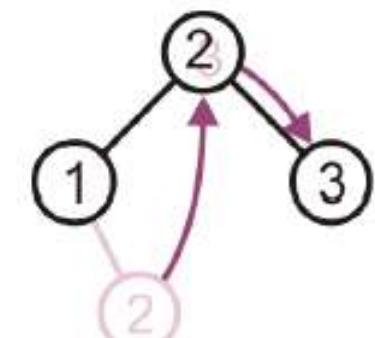
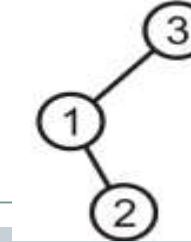
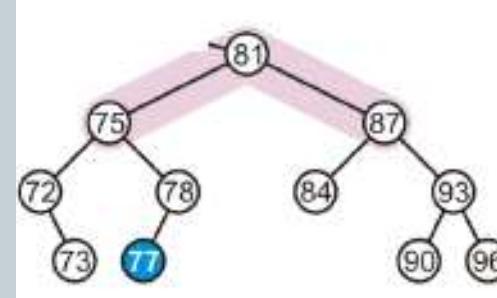
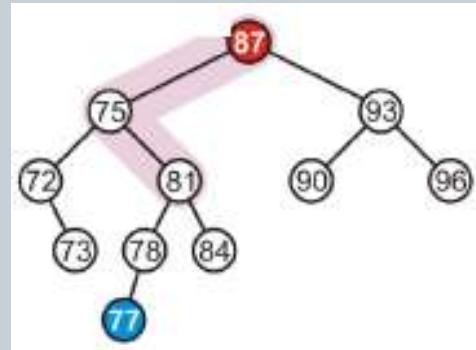
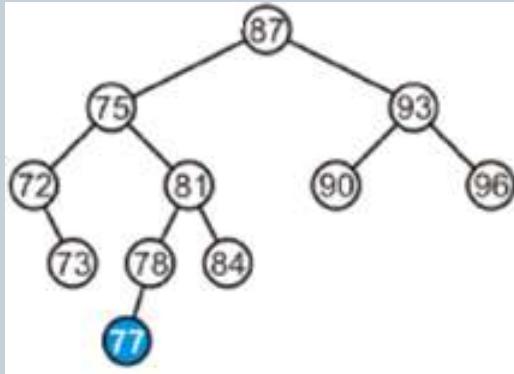


THUMB RULE

# Sample RLs {Elbow}



## Handling RLs



THUMB RULE



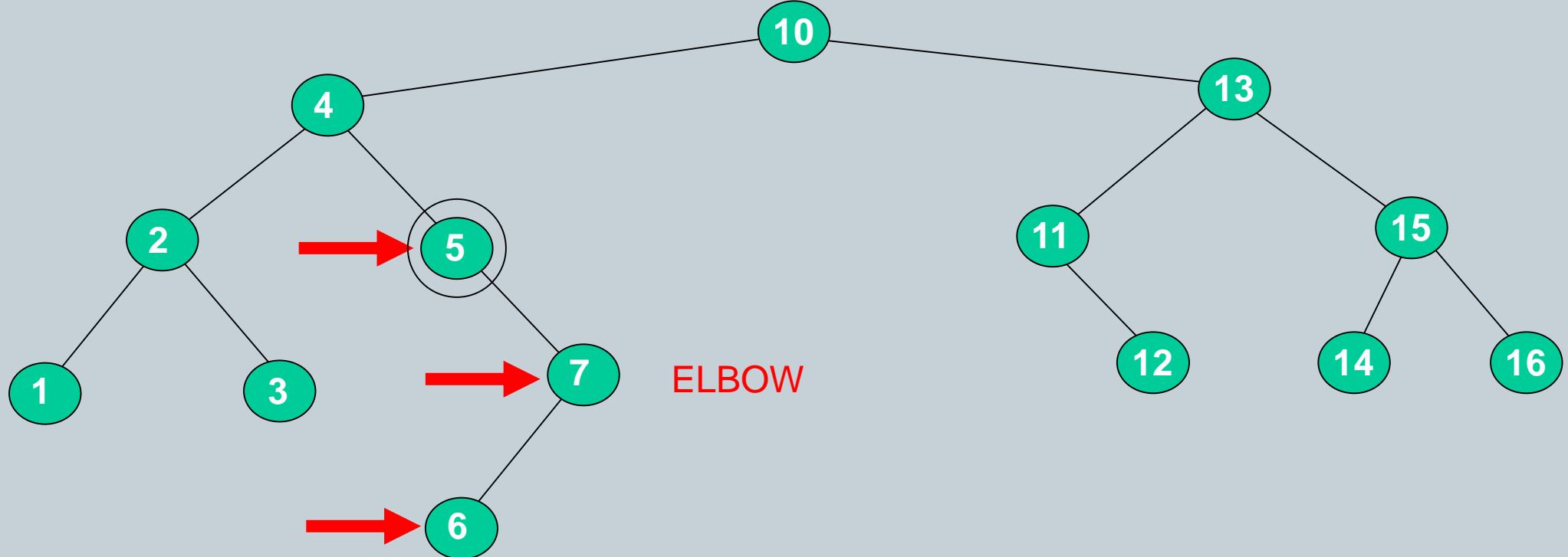
**CASE 4 : LR-> LEFT OF RIGHT ST**  
**(MIRROR CASE OF RL )**

# AVL Tree Rotations for LR

74

## Double rotations:

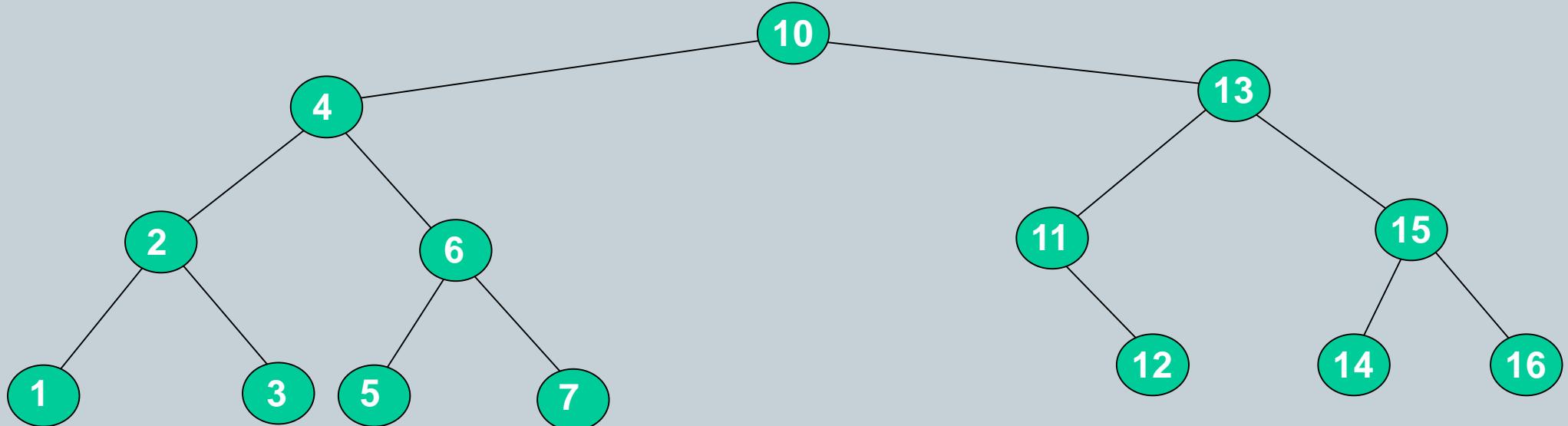
- Rotation type:



# AVL Tree Rotations

## Double rotations:

- AVL balance restored.



- Now insert 9 and 8.



6) How to Implement the Required Result in  
each of the 4 cases

# AVL Balance Results



- To balance itself, an AVL tree may perform the following four kinds of **rotations** –
  - Right rotation (for LL)
  - Left rotation (for RR)
  - Left-Right rotation(for RL)
  - Right-Left rotation(for LR)
- 
- The first two rotations are **single rotations** and the next two rotations are **double rotations**

Note : To have an **unbalanced tree**, we at least need a **tree of height 2**



# Implementation of Balancing in 4 cases

In the 4 CASES:

- (1) LL (Left of Left Subtree) – Right Rotation
- (2) RR (Right of Right Subtree) – Left Rotation
- (3) RL (Right of Left Subtree) – Left Rotation and Then Right Rotation
- (4) LR (Left of Right Subtree) – Right Rotation and then Left Rotations



# CASE 1 : LL-> LEFT OF LEFT ST (RIGHT ROTATION)

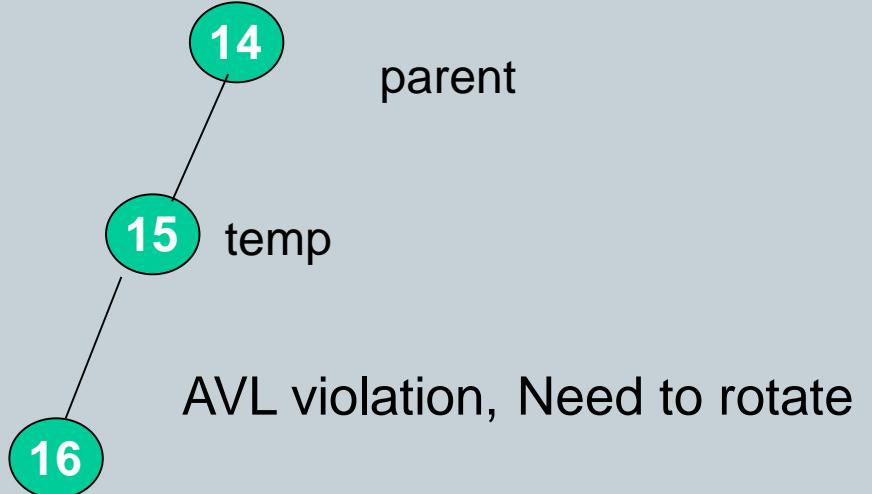
# AVL Tree Rotations

Single rotations:- Left of Left Case (Right Rotation)

(**Insert element to Left of Left subtree**)

Rotate to Right when : Inserting 16 causes AVL violation

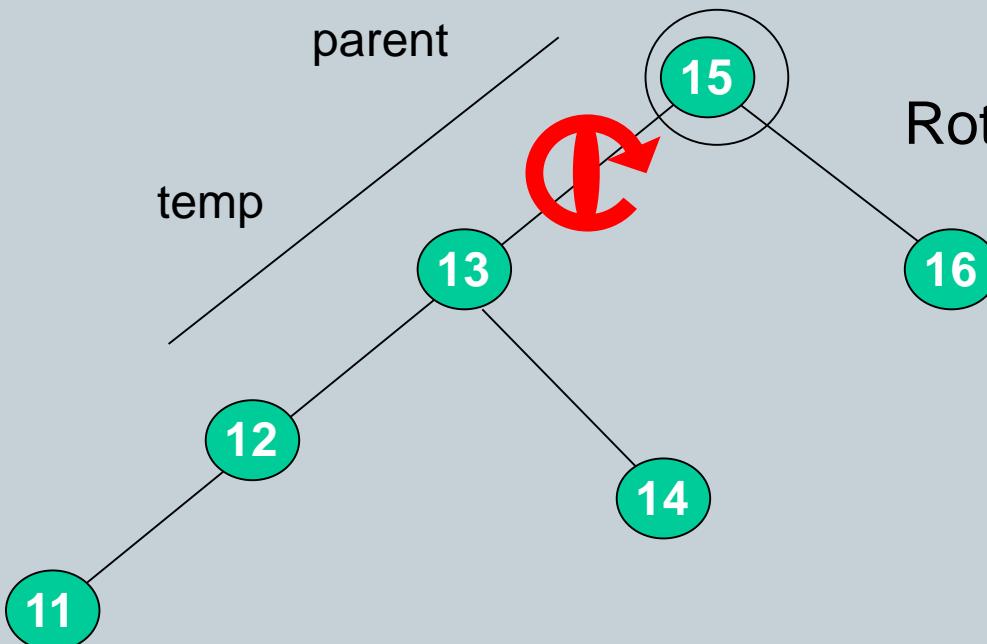
```
ll_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->left;
    parent->left = temp->left; // (NULL here)
    temp->right = parent;
    return temp;
}
```



# AVL Tree Rotations

81

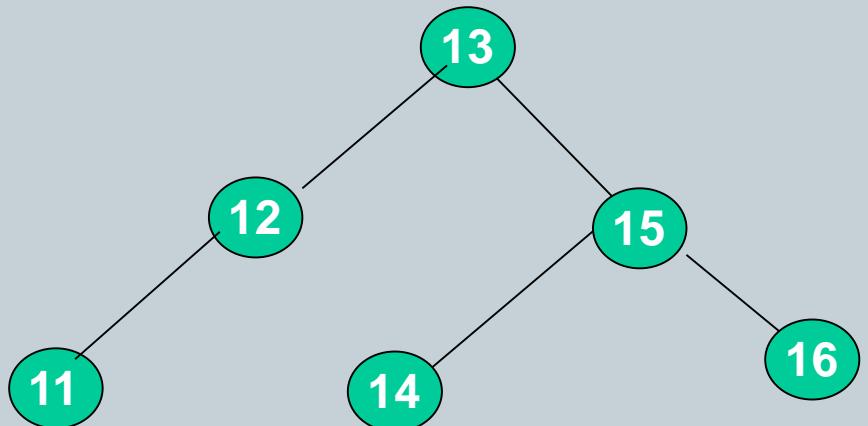
LL Case: Single rotations:



Rotation type: Right Rotation

```
ll_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->left; // (here 13)
    parent->left = temp->left; // (here 14)
    temp->right = parent; // (here 15)

    return temp;
}
```





## CASE 2 : RR (MIRROR OF CASE 1)

# AVL Tree Rotations

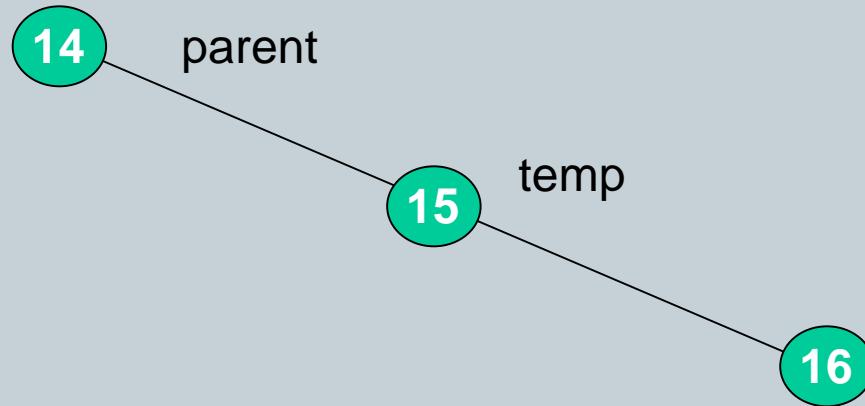
Single rotations:- Right of Right Rotation (Left Rotation)

**(Insert element to right of right subtree )**

Rotate to left

- Inserting 16 causes AVL violation:

```
rr_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->right;
    parent->right = temp->left; //NULL here
    temp->left = parent;
    return temp;
}
```



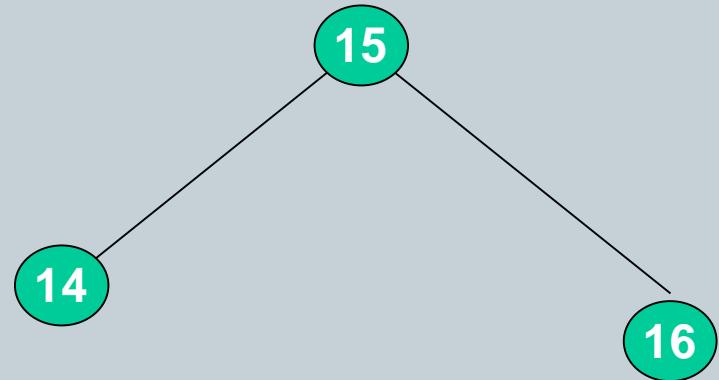
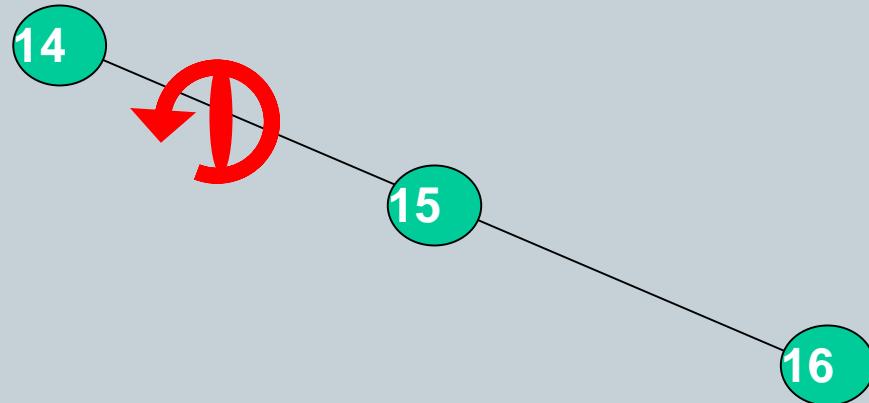
Need to rotate

# RR Case : AVL Tree Rotations

84

Single rotations:

- Rotation type: **Left Rotation**





**CASE 3 : RL -> RIGHT OF LEFT  
ST : LEFT-RIGHT ROTATION**

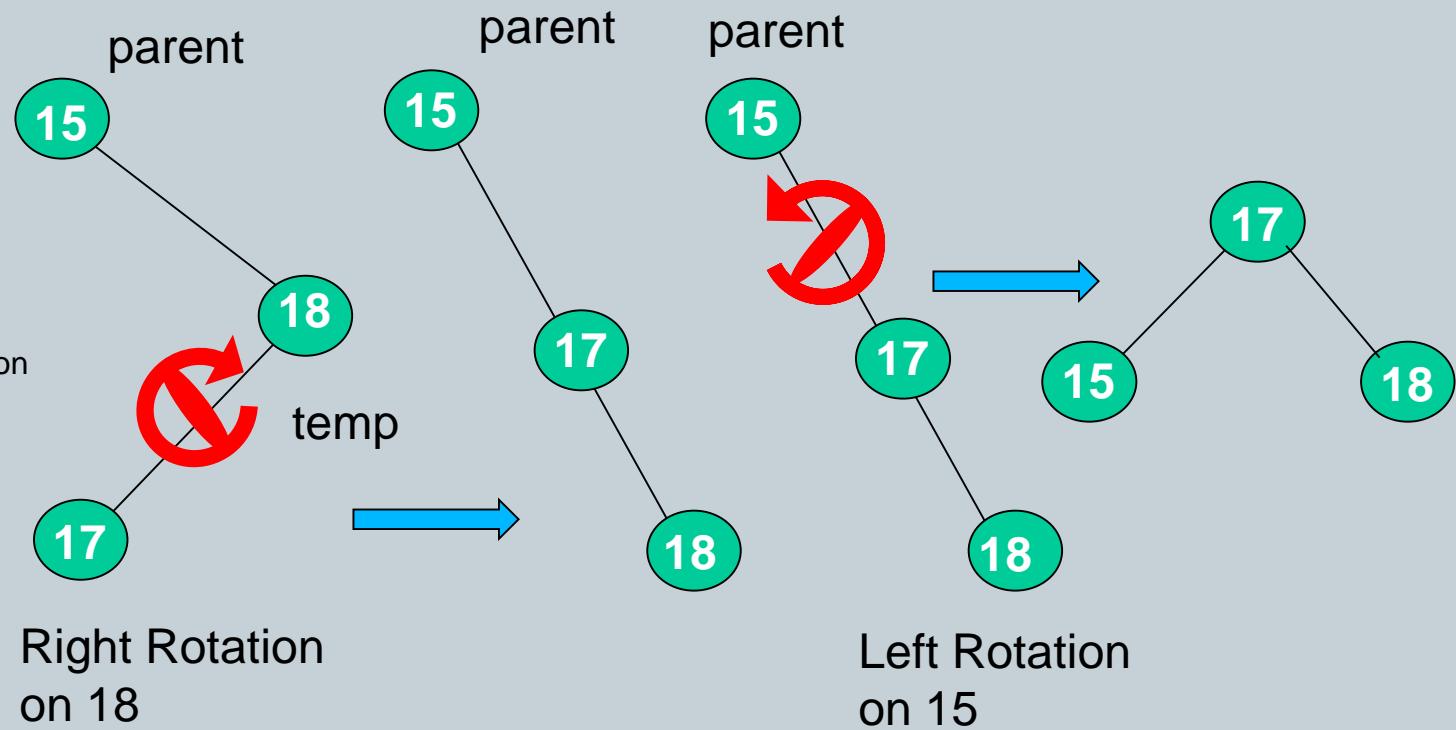
# RL Case: Left-Right rotation:

86

**Insert element as Left child of Right subtree**

Rotate to Right first and then rotate to left

```
rl_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->right;
    parent->right = ll_rotation (temp); //Right Rotation
    Return rr_rotation (parent); //Left Rotation
}
```





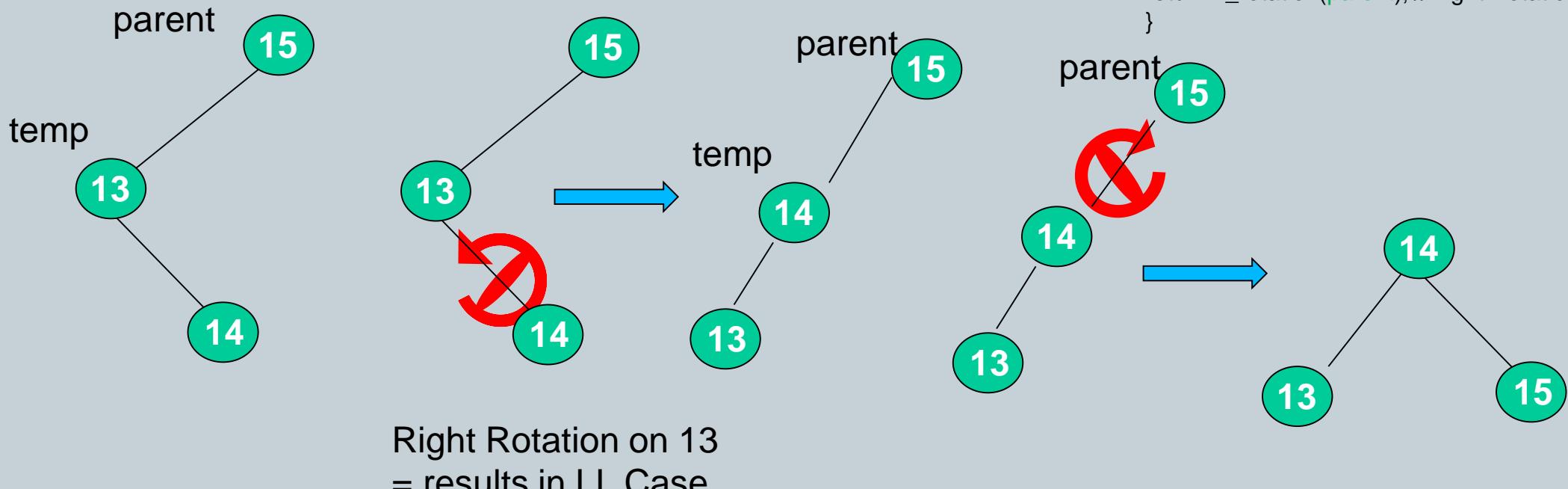
## **CASE 4 : LR-> LEFT OF RIGHT ST : LEFT-RIGHT ROTATION**

# Case LR : Double rotations: Left Right Rotation:

## Insert element as Right child of Left subtree

88

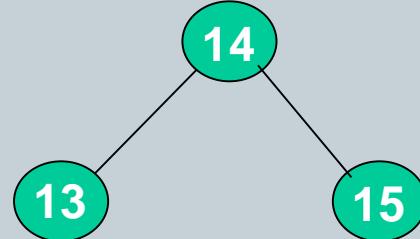
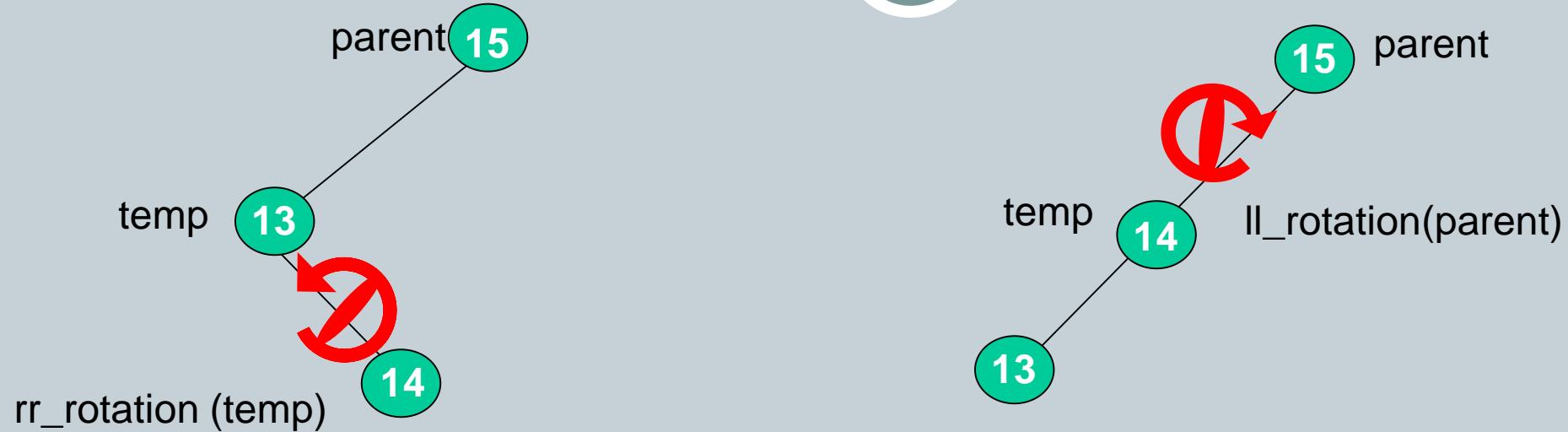
Rotate to left first and then rotate to right



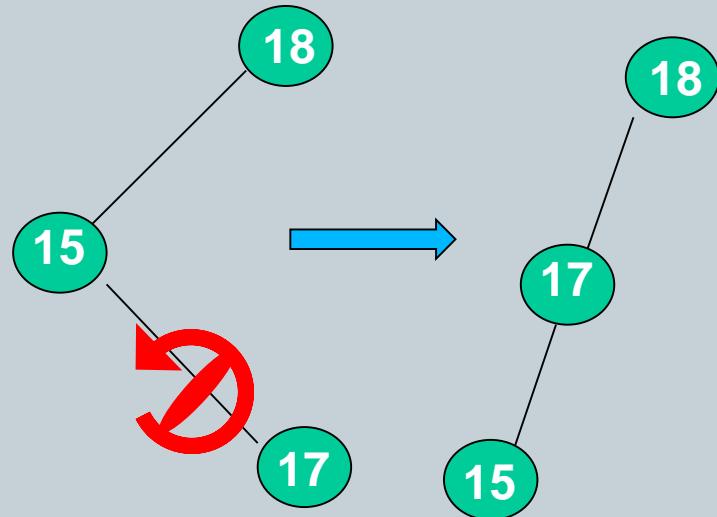
```
lr_rotation(avl_node *parent)
{
    avl_node *temp;
    temp = parent->left;
    parent->left = rr_rotation(temp); // Left Rotation
    return ll_rotation(parent); //Right Rotation
}
```

# LR CASE: AVL Tree Rotations : First Left Then Right Rotation

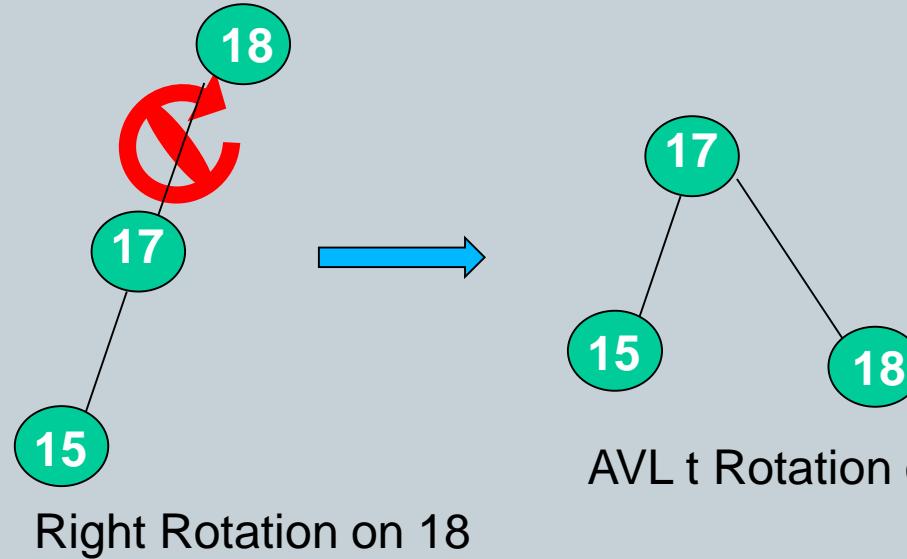
89



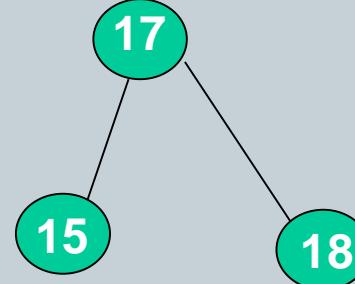
## LR CASE: AVL Tree Rotations : First Left Then Right Rotation



Left Rotation on 15 = results in LL Case



Right Rotation on 18



AVL t Rotation on 18



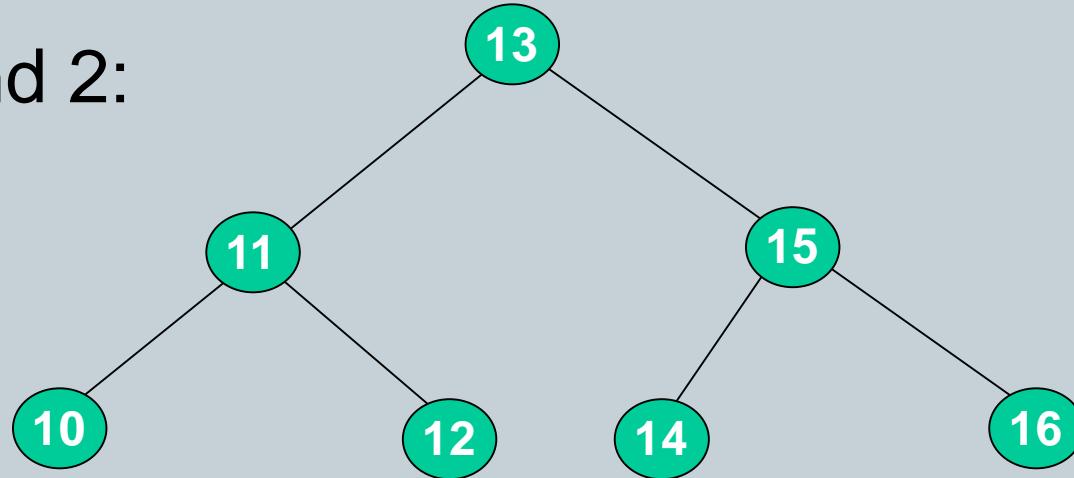
# AVL WALK THROUGH for a SET of INSERTIONS

# AVL Tree Rotations

92

**Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8 in existing tree.**

- First insert 1 and 2:

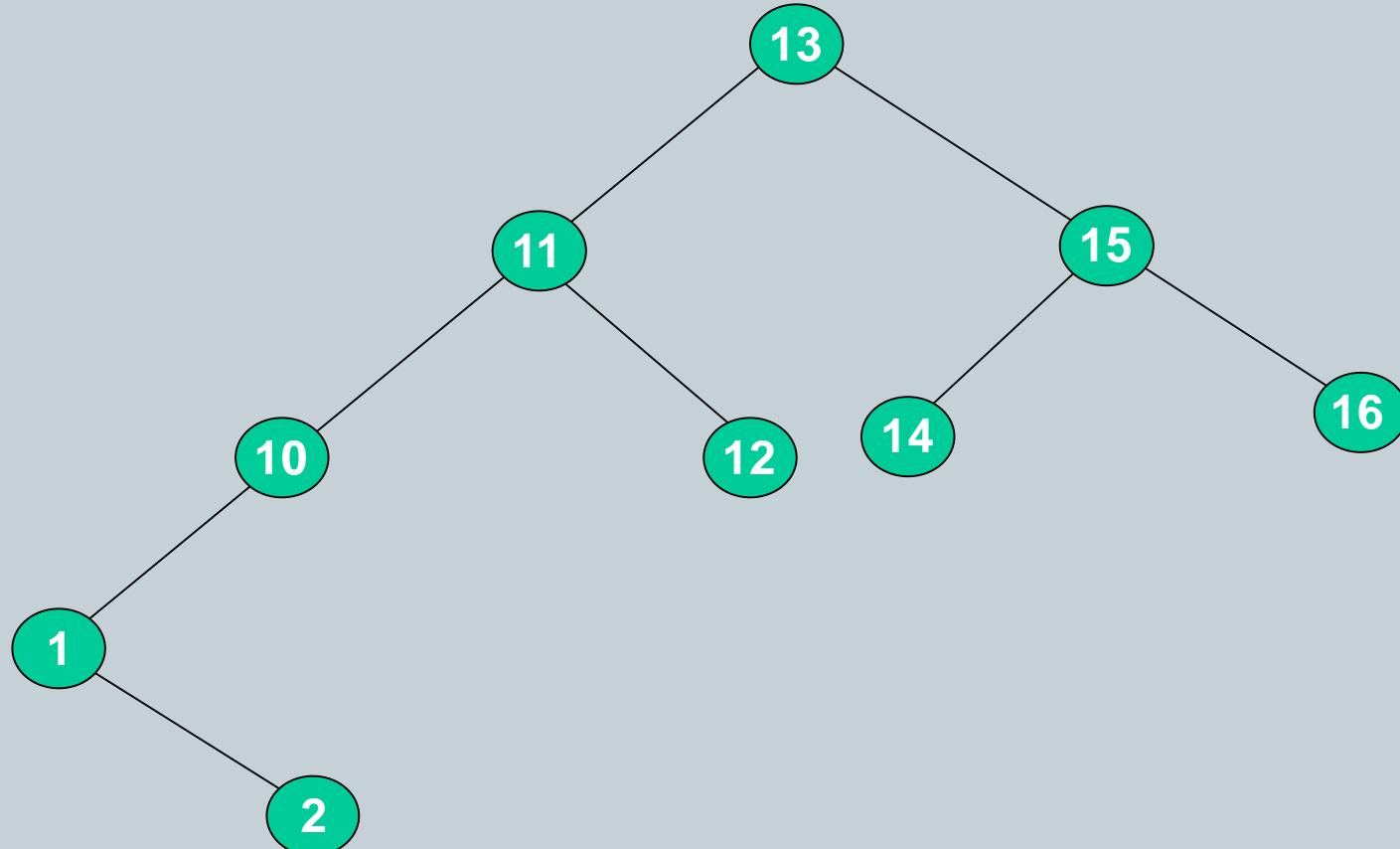


# AVL Tree Rotations

93

## Double rotations:

- AVL violation - rotate

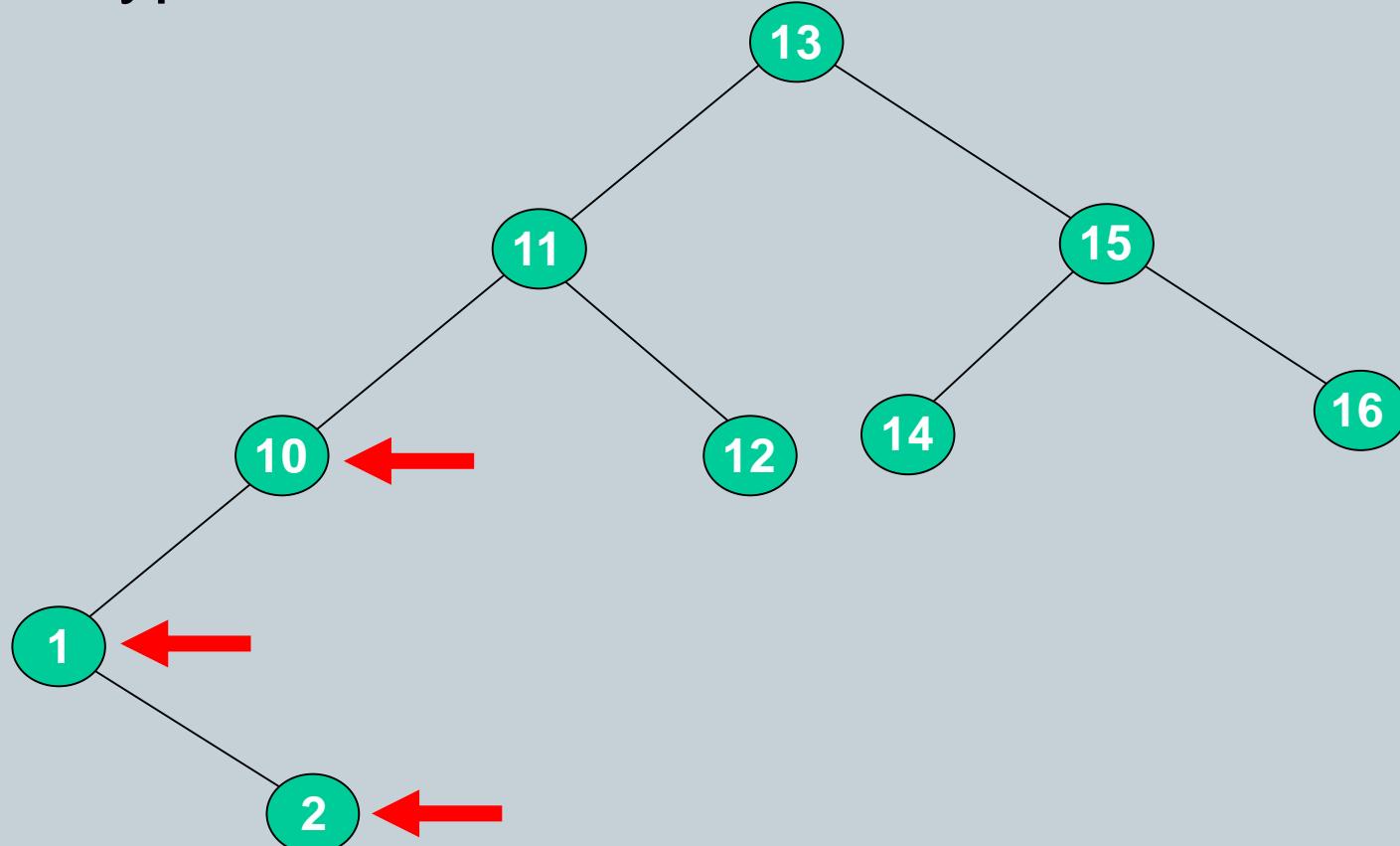


# AVL Tree Rotations

94

Double rotations:

- Rotation type:

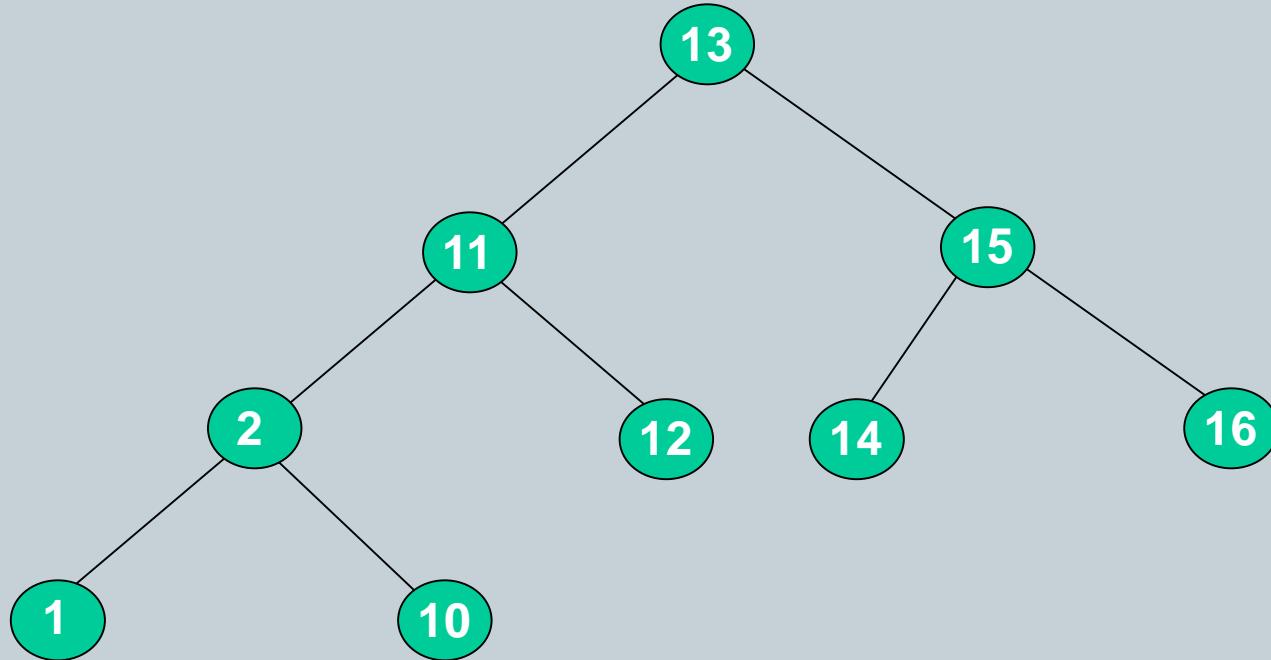


# AVL Tree Rotations

95

## Double rotations:

- AVL balance restored:



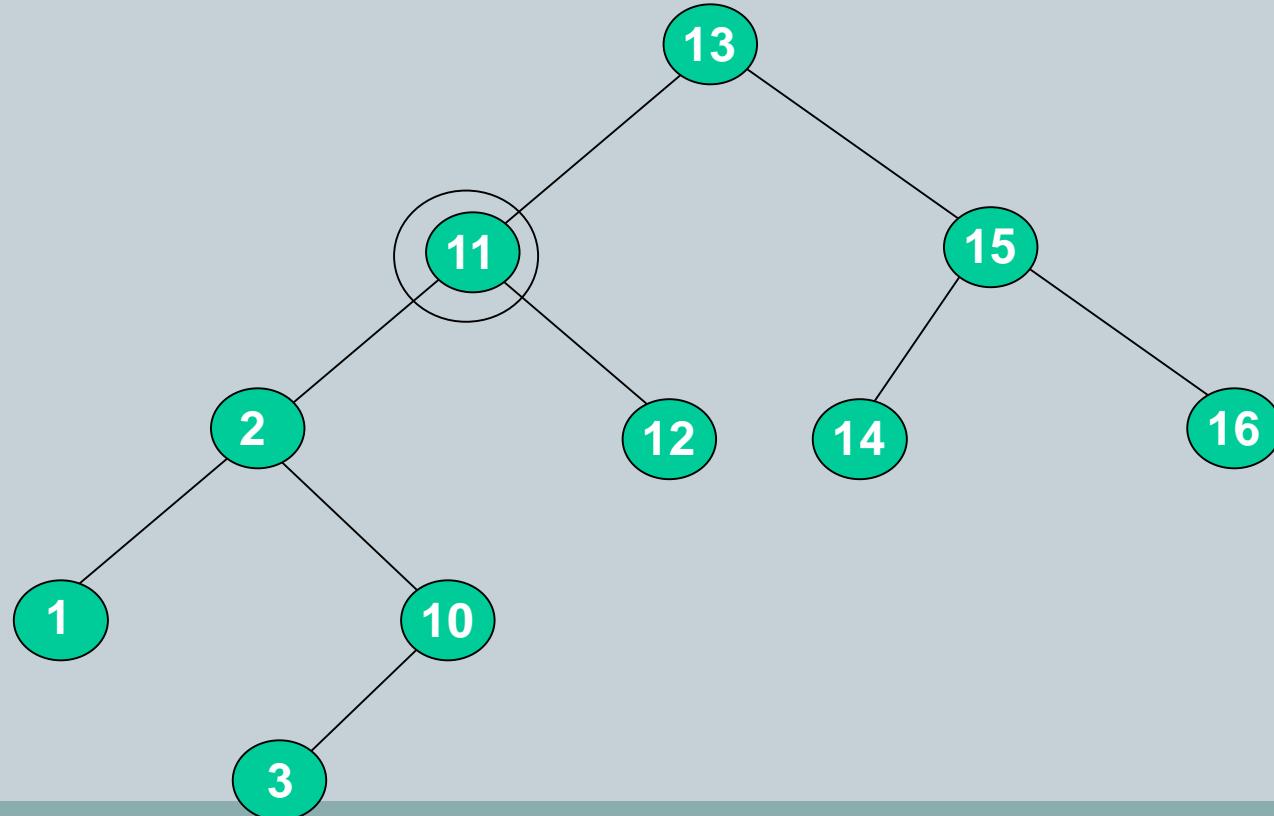
- Now insert 3.

# AVL Tree Rotations

96

## Double rotations:

- AVL violation – rotate:

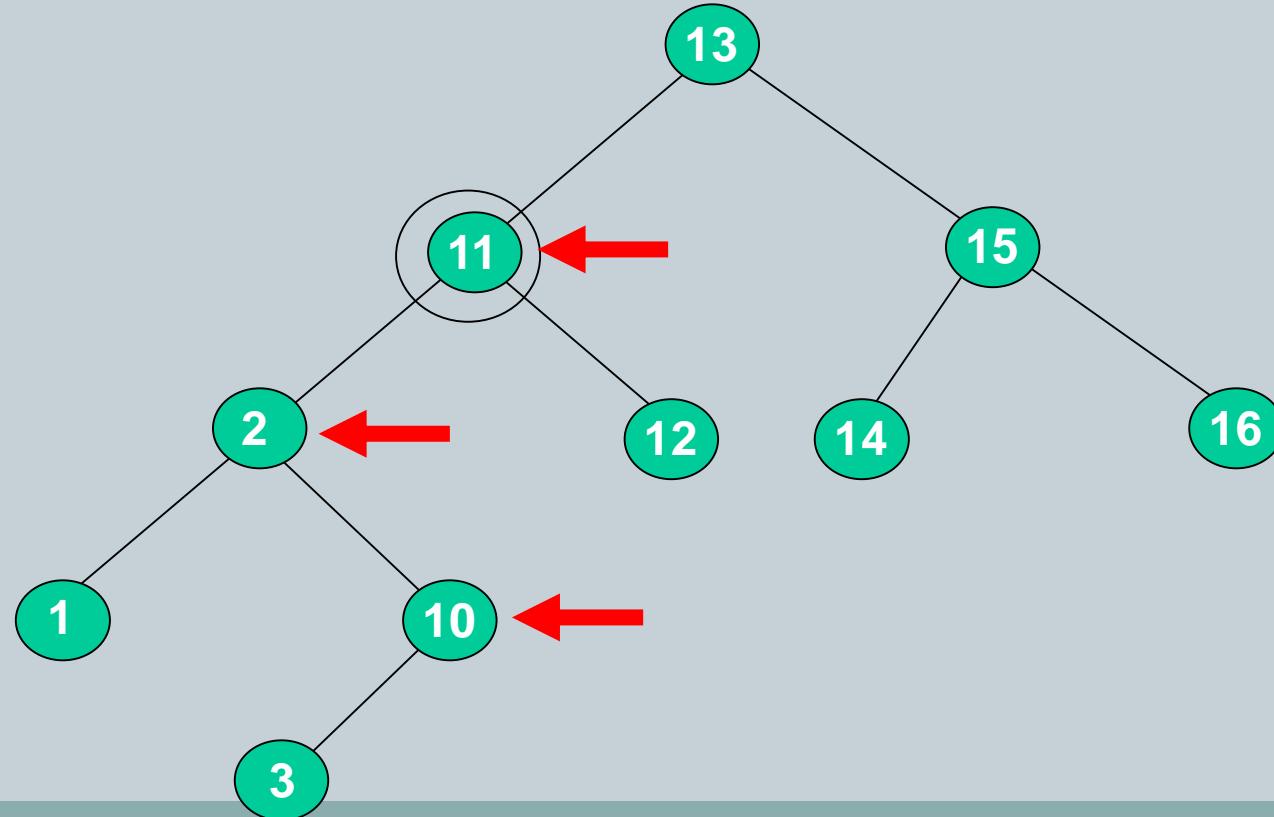


# AVL Tree Rotations

97

## Double rotations:

- AVL violation (RL case elbow) – rotate:

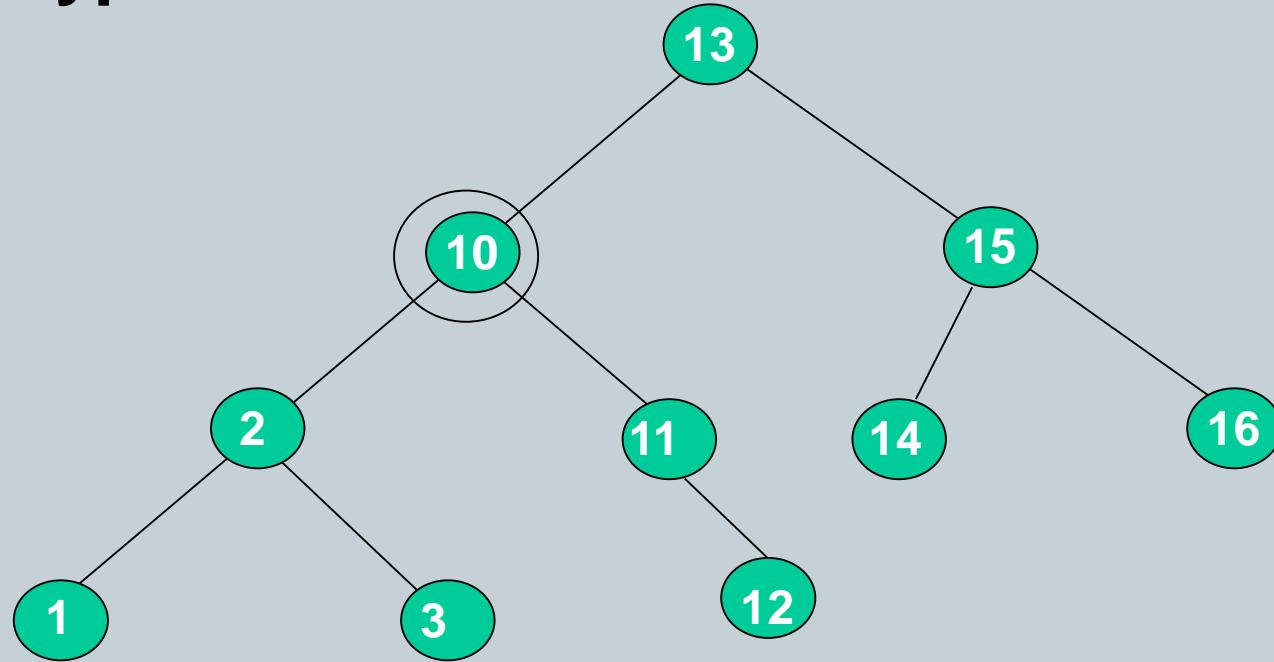


# AVL Tree Rotations

98

## Double rotations:

- Rotation type:



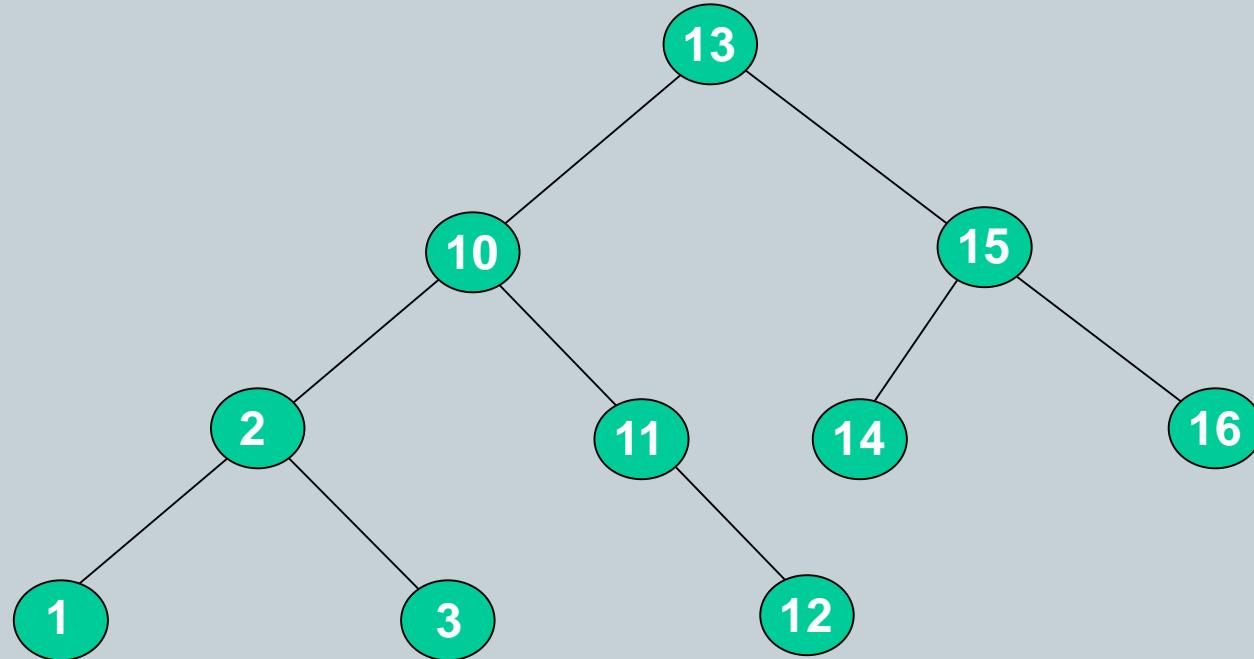
# AVL Tree Rotations

99

## Double rotations:

- **AVL balance restored:**

After Right Rotation on 11



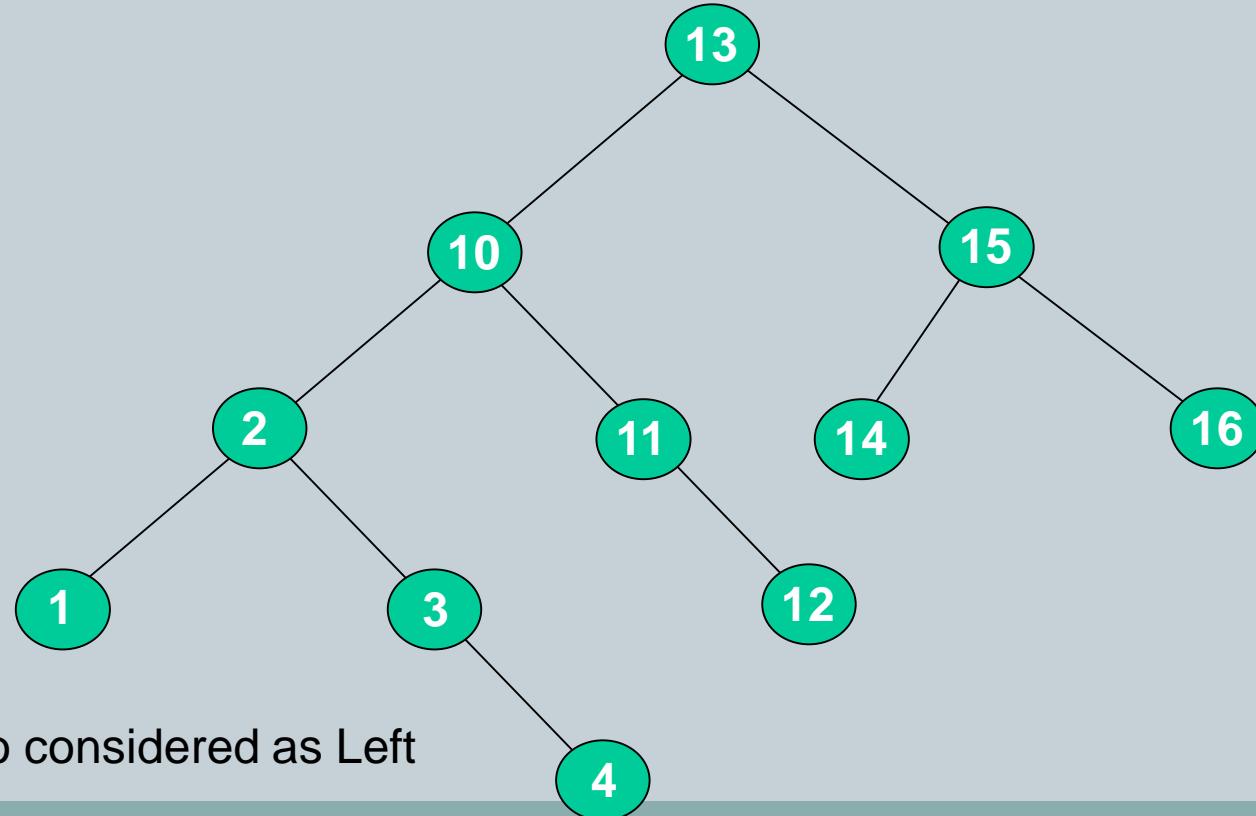
- Now insert 4.

# AVL Tree Rotations

100

## Rotations:

- AVL violation - rotate



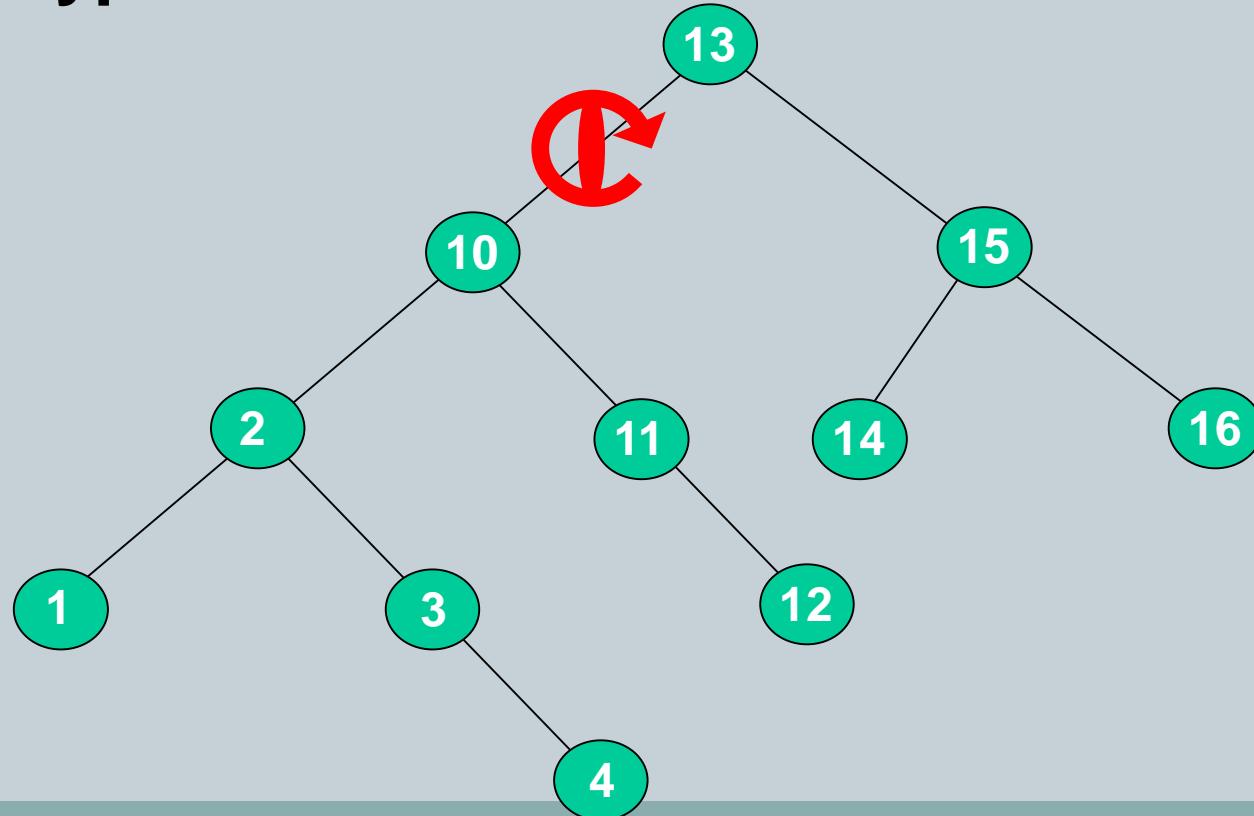
Not direct Left Right so considered as Left  
Left  
so performed right Rotation

# AVL Tree Rotations

101

**rotations:**

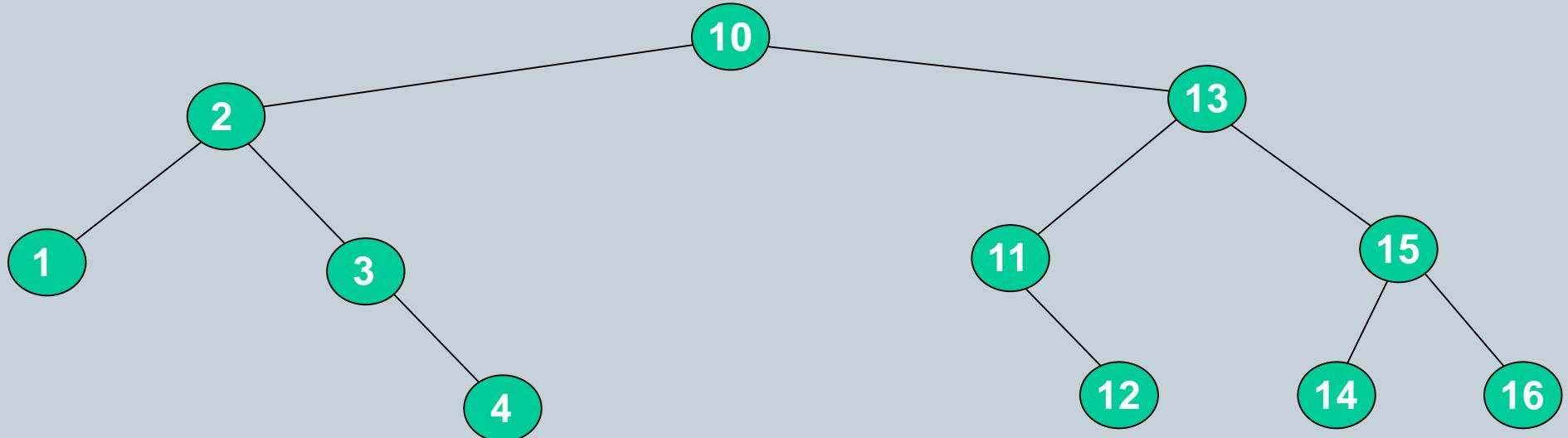
- **Rotation type:**



# AVL Tree Rotations

102

## Double rotations:

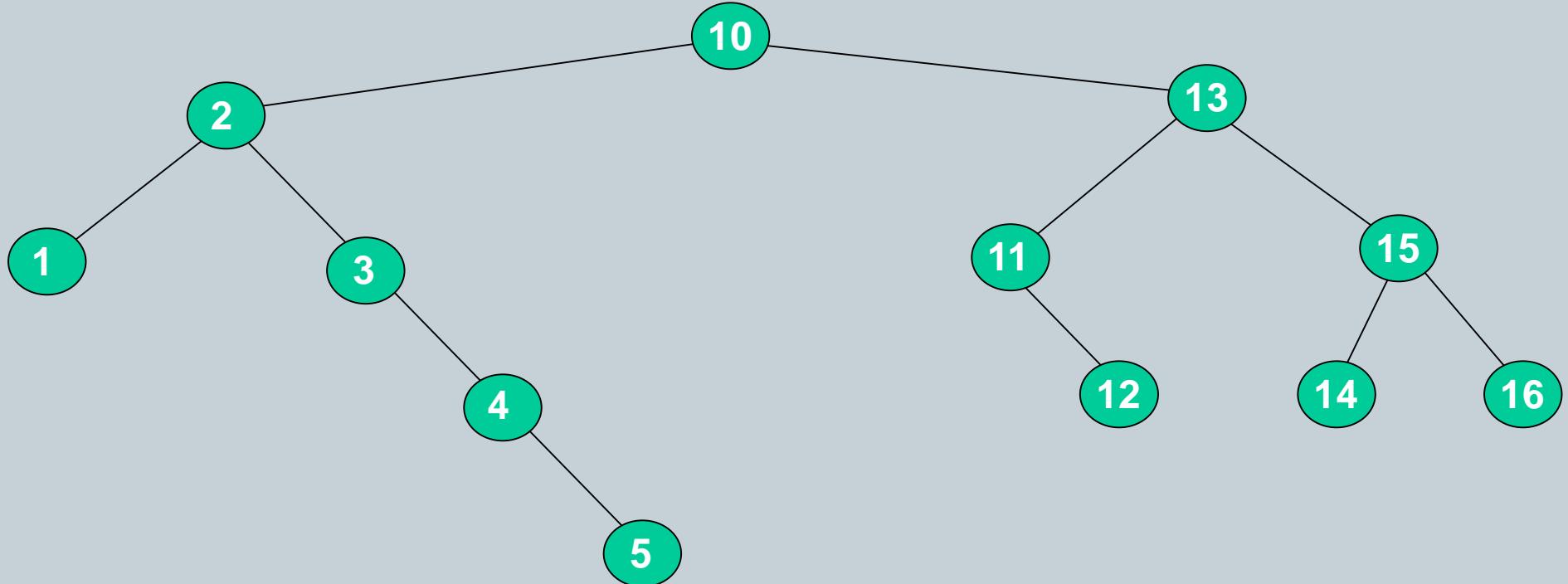


- Now insert 5.

# AVL Tree Rotations

103

**rotations:**



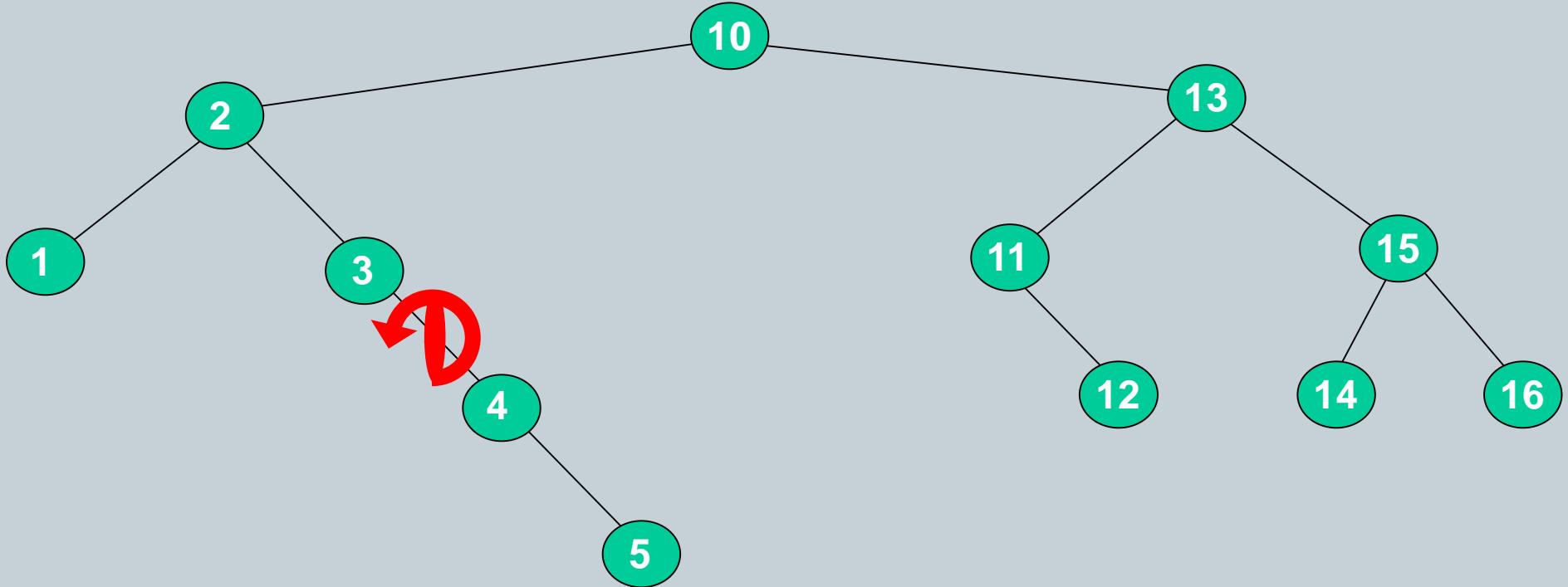
- **AVL violation – rotate.**

# AVL Tree Rotations

104

## Single rotations:

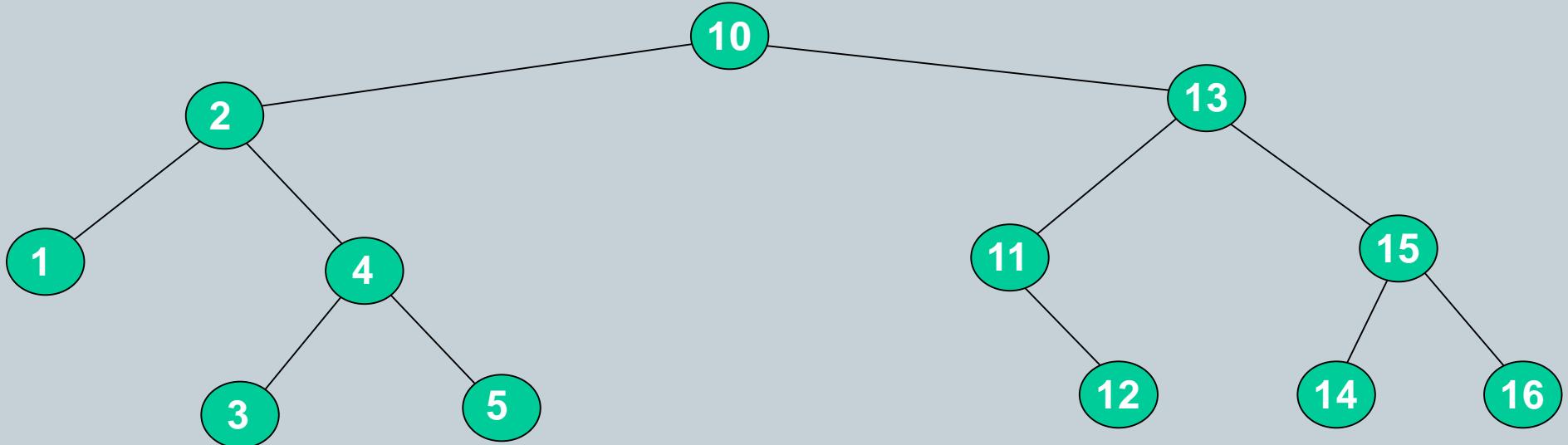
- Rotation type:



# AVL Tree Rotations

## Single rotations:

- AVL balance restored:



105

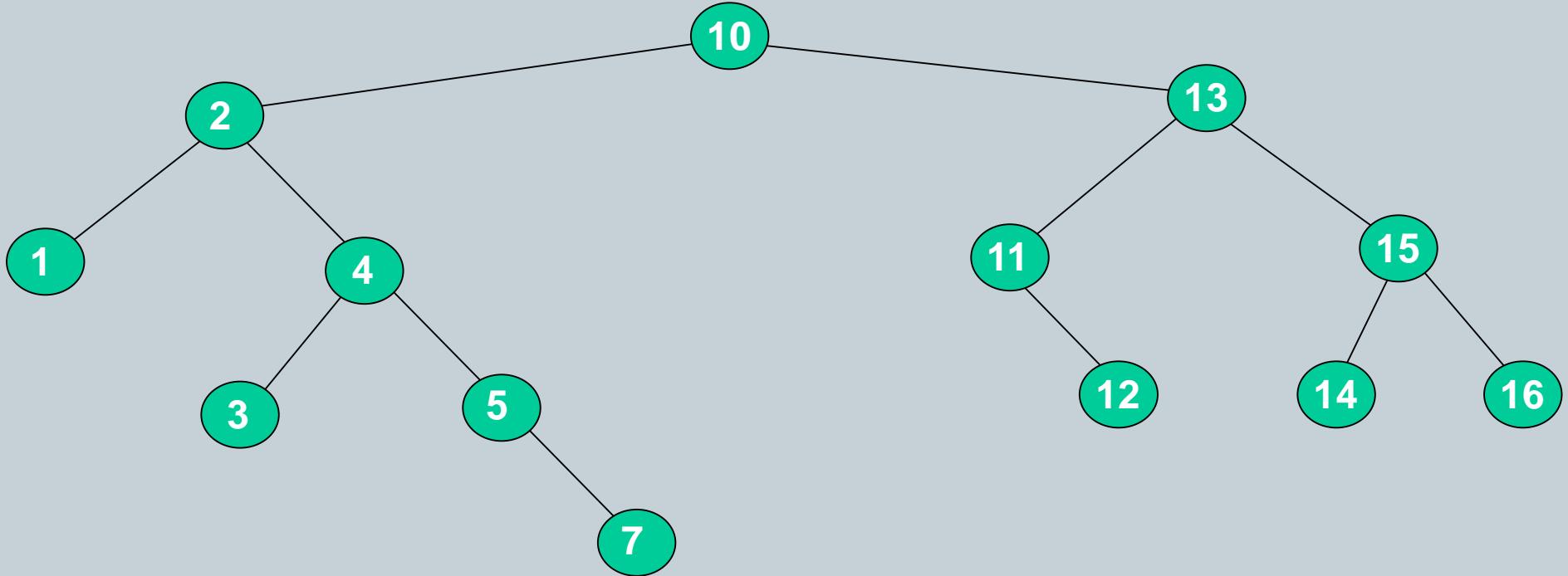
- Now insert 7.

# AVL Tree Rotations

106

## Single rotations:

- AVL violation – rotate.

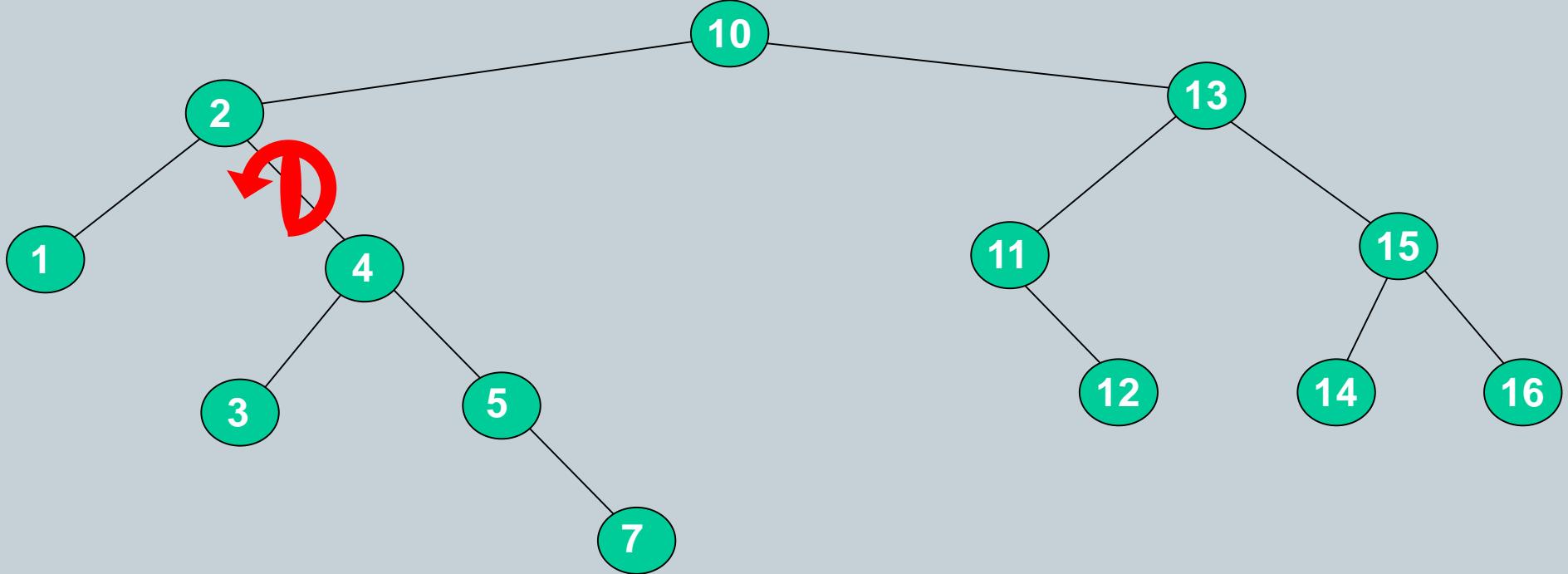


# AVL Tree Rotations

107

## Single rotations:

- Rotation type:

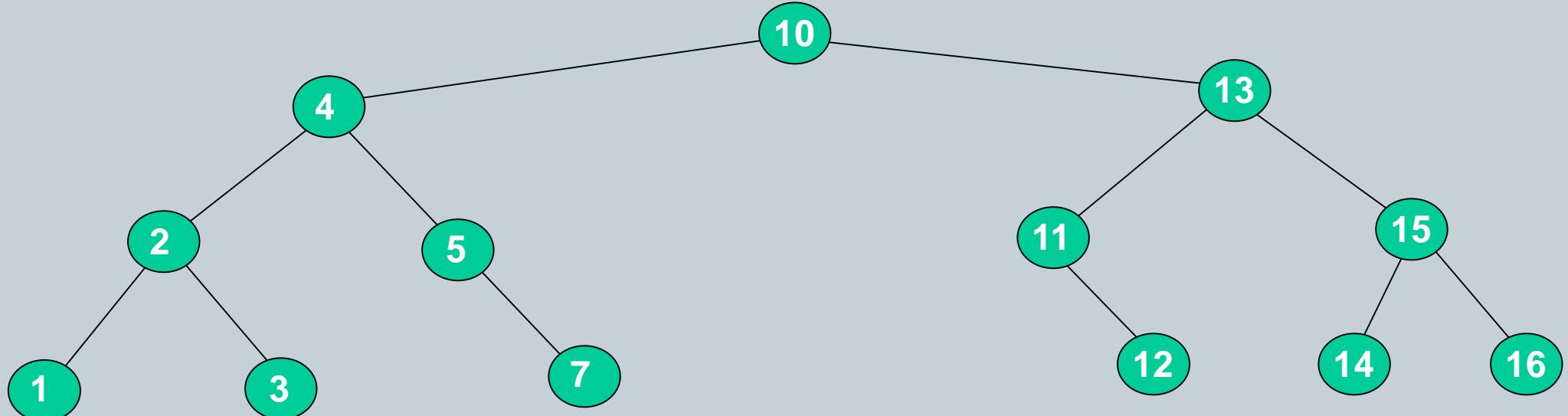


# AVL Tree Rotations

108

## Double rotations:

- AVL balance restored.



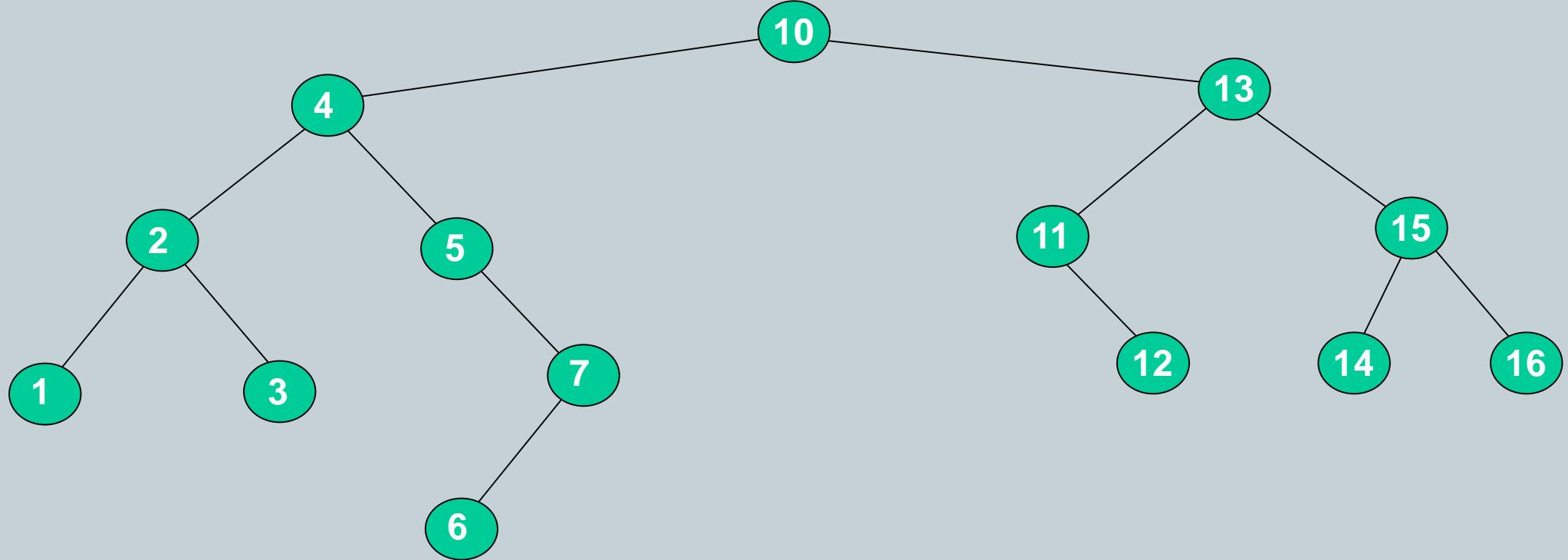
- Now insert 6.

# AVL Tree Rotations

109

## Double rotations:

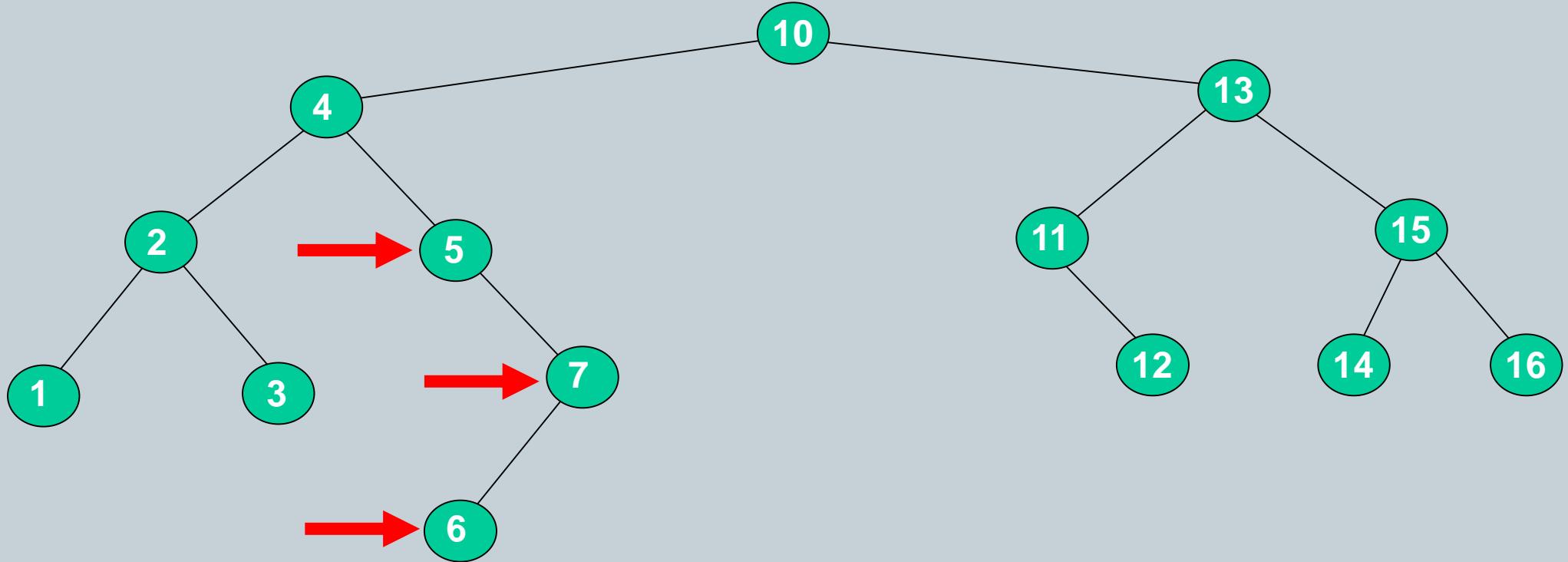
- AVL violation - rotate.



# AVL Tree Rotations

## Double rotations:

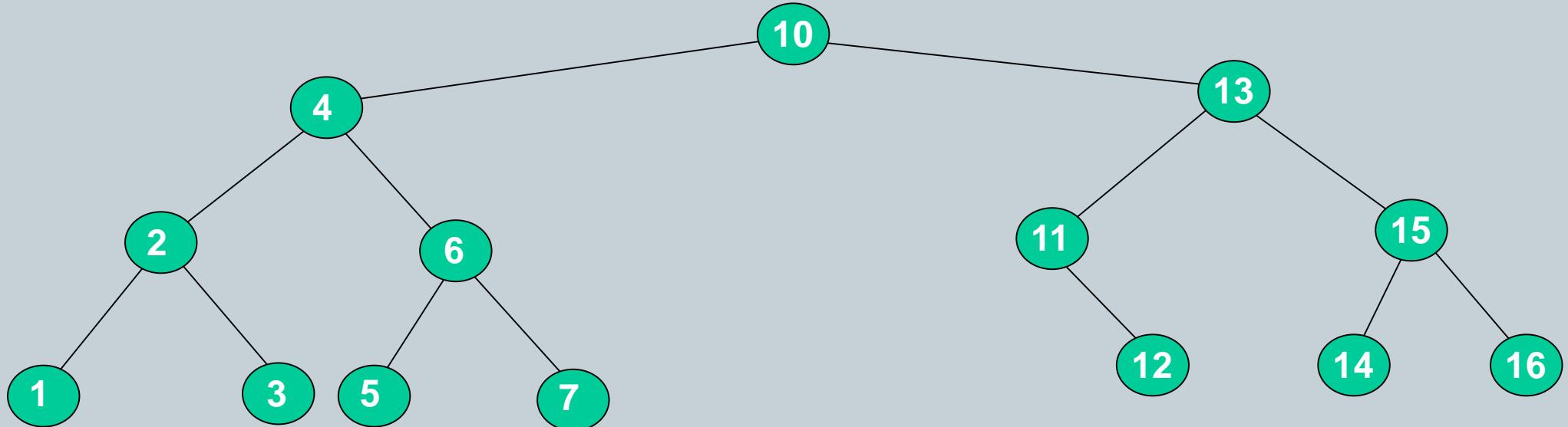
- Rotation type:



# AVL Tree Rotations

## Double rotations:

- AVL balance restored.

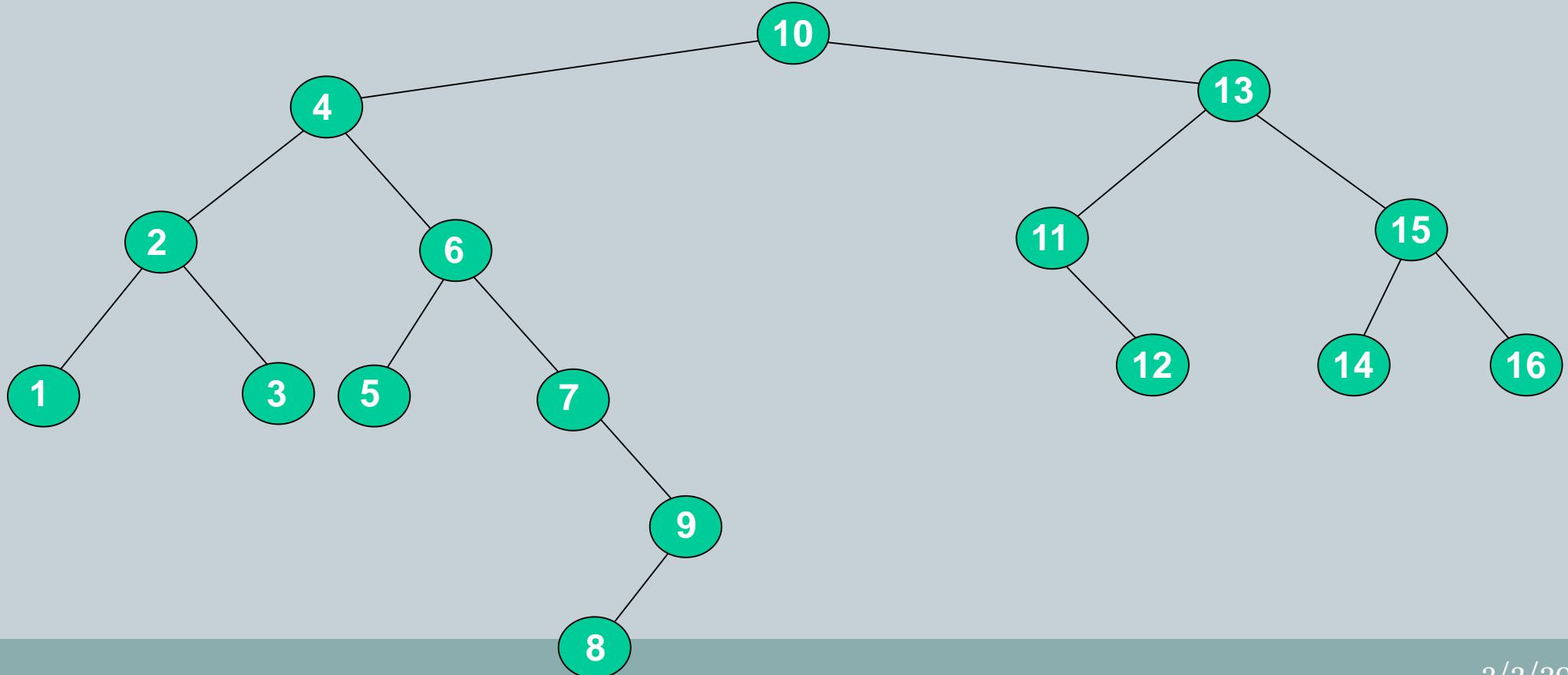


- Now insert 9 and 8.

# AVL Tree Rotations

## Double rotations:

- AVL violation - rotate.

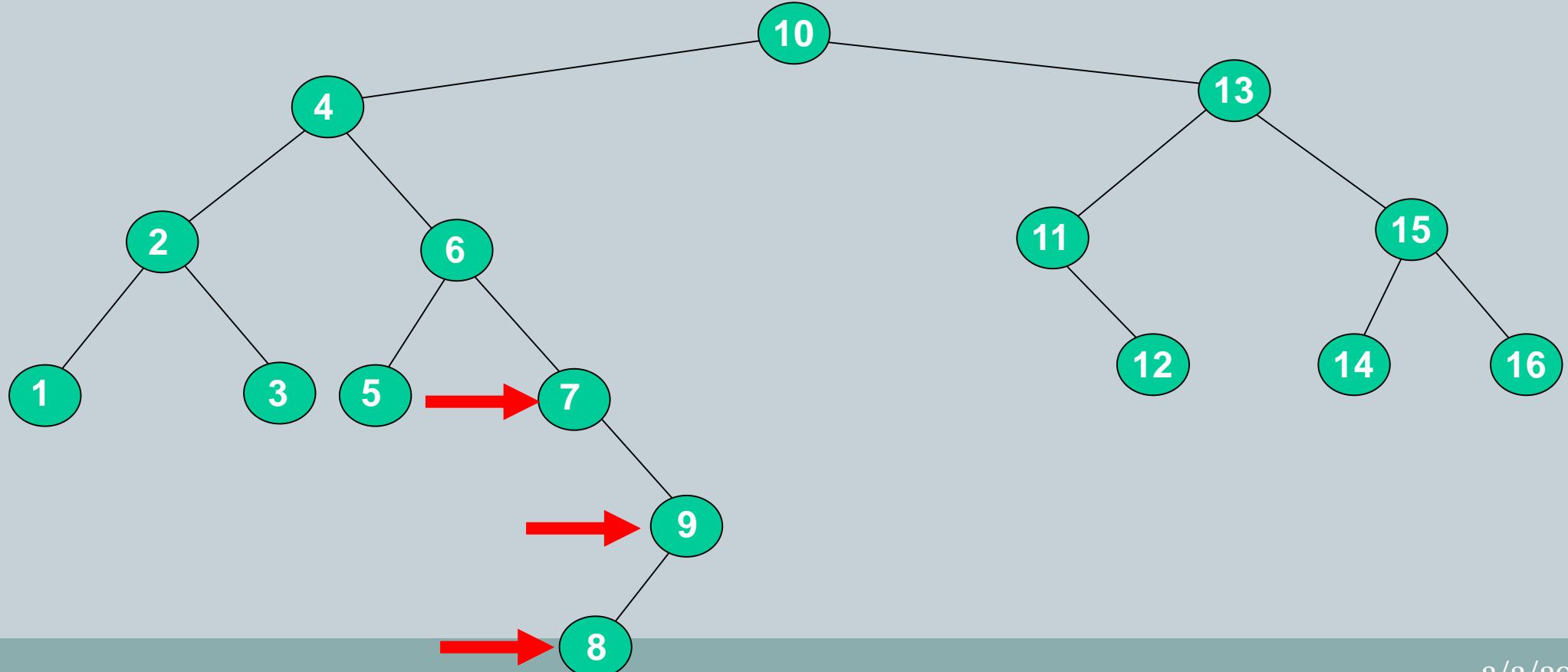


# AVL Tree Rotations

113

## Double rotations:

- Rotation type:

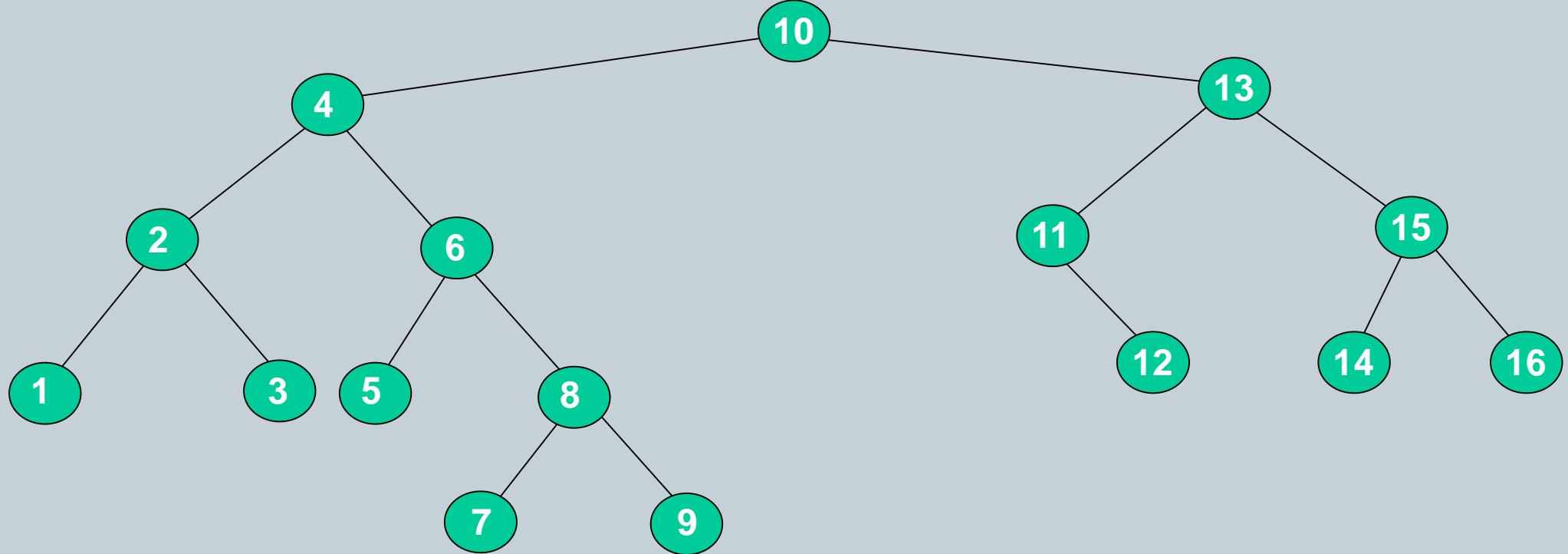


# AVL Tree Rotations

114

Final tree:

- Tree is almost perfectly balanced



# AVL PROs and CONs



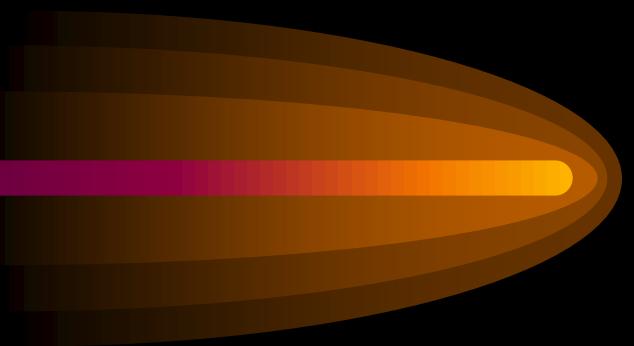
Arguments for AVL trees:

1. Search is  $O(\log N)$  since AVL trees are always balanced.
2. Insertion and deletions are also  $O(\log n)$
3. The height balancing adds no more than a constant factor to the speed of insertion.

Arguments against using AVL trees:

1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).

# *Graphs*



# *Objectives of This Session*

- Graph

- Define the term graph
- Graph terminology
- Adjacency
- Graph representations
- Graph traversals
- Algorithm for depth first traversal

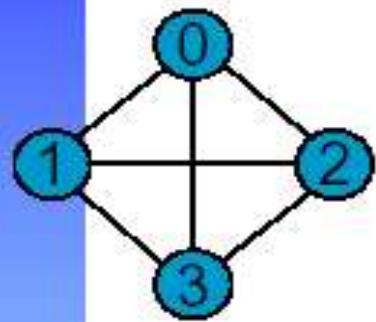
# *Objectives of This Session*

- Algorithm for Breadth first traversal
- State applications of graph
- Prim's algorithm to find minimum spanning tree
- Kruskal's algorithm
- Dijkstra's algorithm to find shortest path between two nodes

# *Graphs : Definitions*

- A graph consists of a set of nodes(or vertices) and a set of arcs(or edges). Each arc in a graph is specified by a pair of nodes.  $\{(A,B),(C,D)\}$
- A Graph  $G$  consists of two sets  $V$  and  $E$ .  $V$  is a finite non-empty set of vertices and  $E$  is a set of pairs of vertices called edges.
- $G=(V,E)$

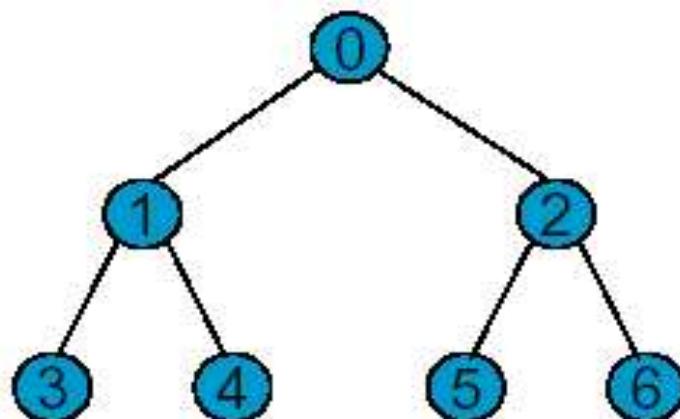
## Three Sample Graphs



$$V(G_1) = \{0, 1, 2, 3\}$$

$$E(G_1) = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}$$

(a) G<sub>1</sub>



$$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$$

$$E(G_2) = \{(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)\}$$

(b) G<sub>2</sub>



$$V(G_3) = \{0, 1, 2\}$$

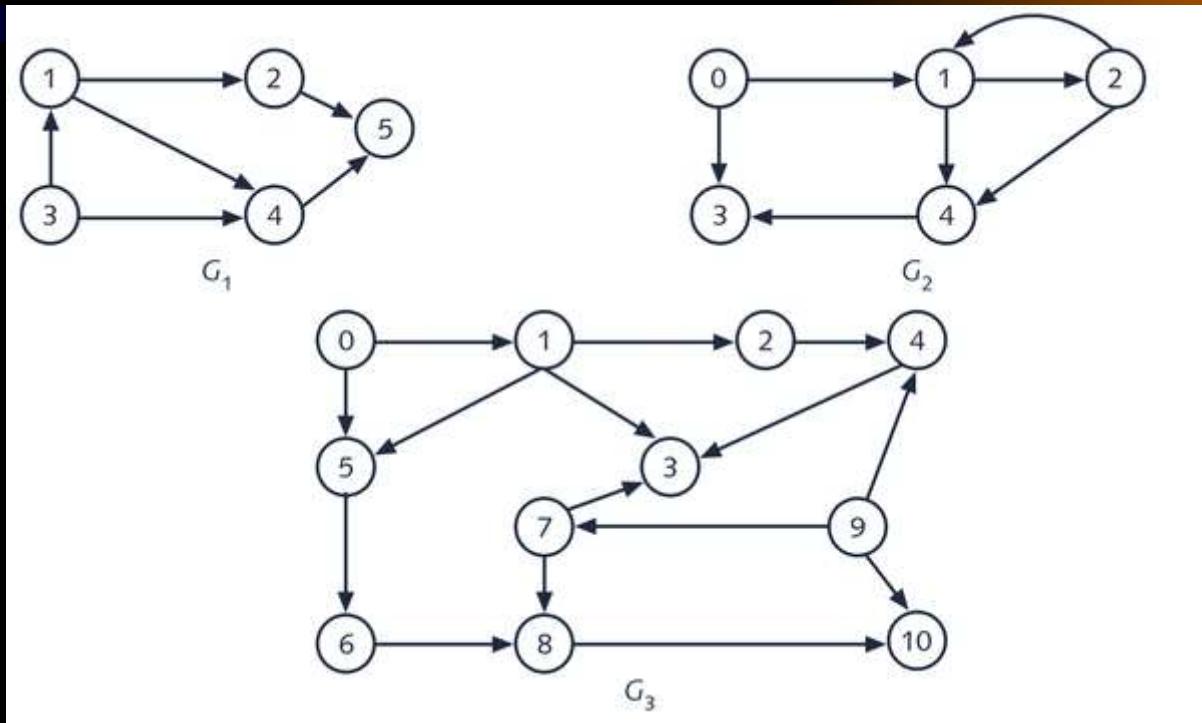
$$E(G_3) = \{<0, 1>, <1, 0>, <1, 2>\}$$

(c) G<sub>3</sub>

# Definitions:

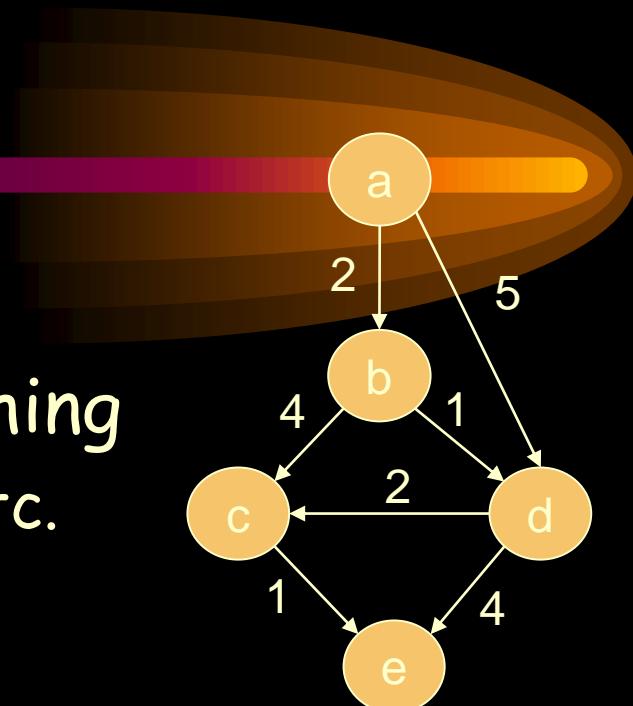
- In an undirected graph, an edge is unordered ie  $(v_1, v_2) = (v_2, v_1)$
- In a directed graph, an edge is ordered  $\langle v_1, v_2 \rangle \neq \langle v_2, v_1 \rangle$ . Here first element of the pair,  $v_1$  is called start vertex and second element of the pair  $v_2$  is called the end vertex.
- In a directed graph, the max number of edges is  $n(n-1)$ .

# Various Directed Graphs



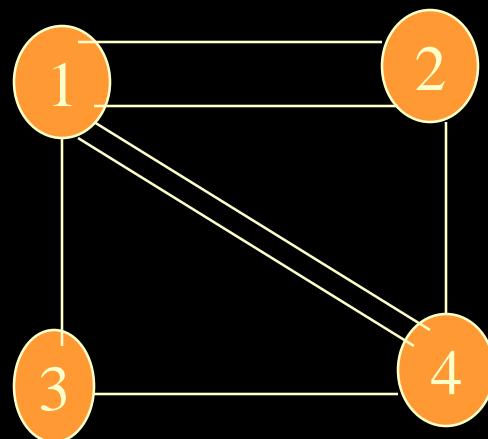
# *Weighted graphs*

- Edges or vertices can have associated "weights"
- Weights can represent anything
  - Distances, costs, capacities, etc.



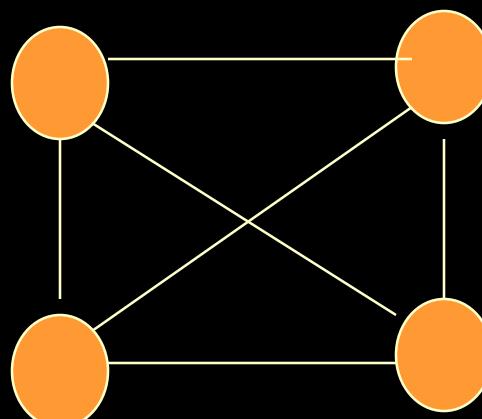
# *Definitions :*

- If there is more than one edge between two vertices, it is called a multigraph.



# *complete graph.*

- An undirected graph, in which every vertex has an edge to all other vertices is called complete graph. Maximum number of edges in complete graph is  $n(n-1)/2$ . In a directed graph, the max number of edges is  $n(n-1)$ .



# Adjacency

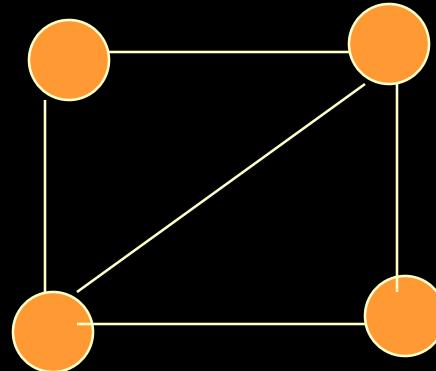
- Adjacent : If  $(v_1, v_2)$  is an edge in  $E(G)$ , then  $v_1$  and  $v_2$  are adjacent vertices.

# Path

- A path from vertex  $v_0$  to vertex  $v_n$  in a graph  $G$  is a sequence of vertices  $v_0, v_1, v_2 \dots v_{n-1}, v_n$  such that  $(v_0, v_1), (v_1, v_2) \dots (v_{n-1}, v_n)$  are edges in  $E(G)$ , the length of the path is the number of edges on it. (if it is a weighted graph, the length of the path is the sum of the individual weights on the edges).

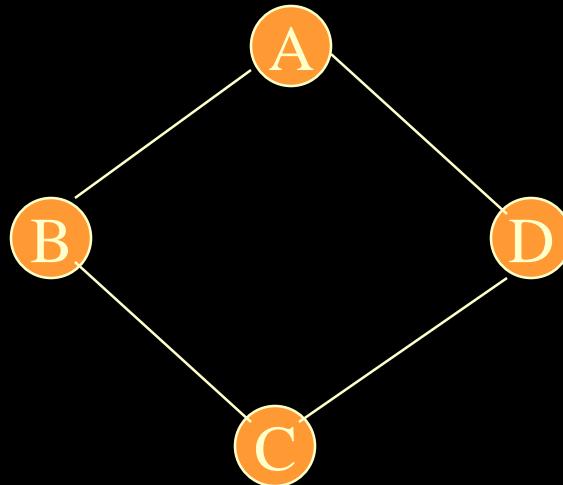
# Paths

- A simple path is a path in which all vertices are distinct.
- A cycle is a simple path in which the first and last vertices are the same.



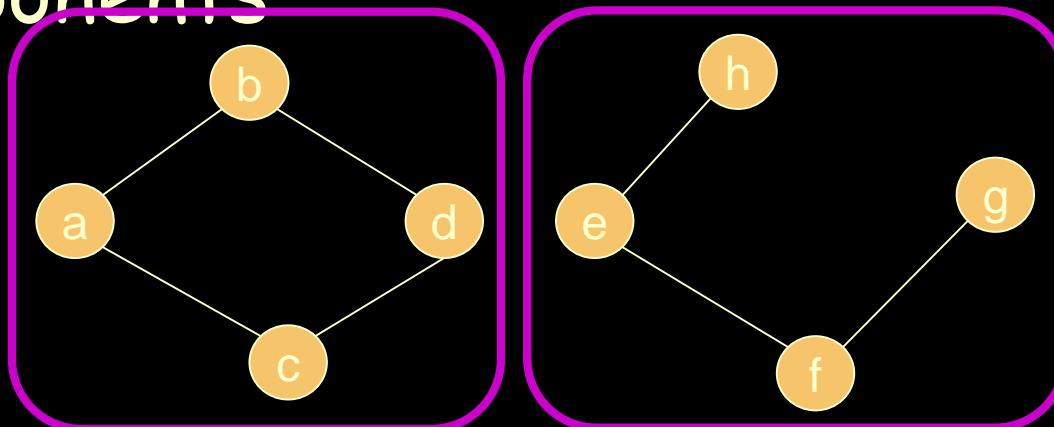
# *Connected Graph:*

- An undirected graph is said to be connected if there exist a path between every pair of vertices  $v_i$  and  $v_j$ .



# *Example Of Not Connected*

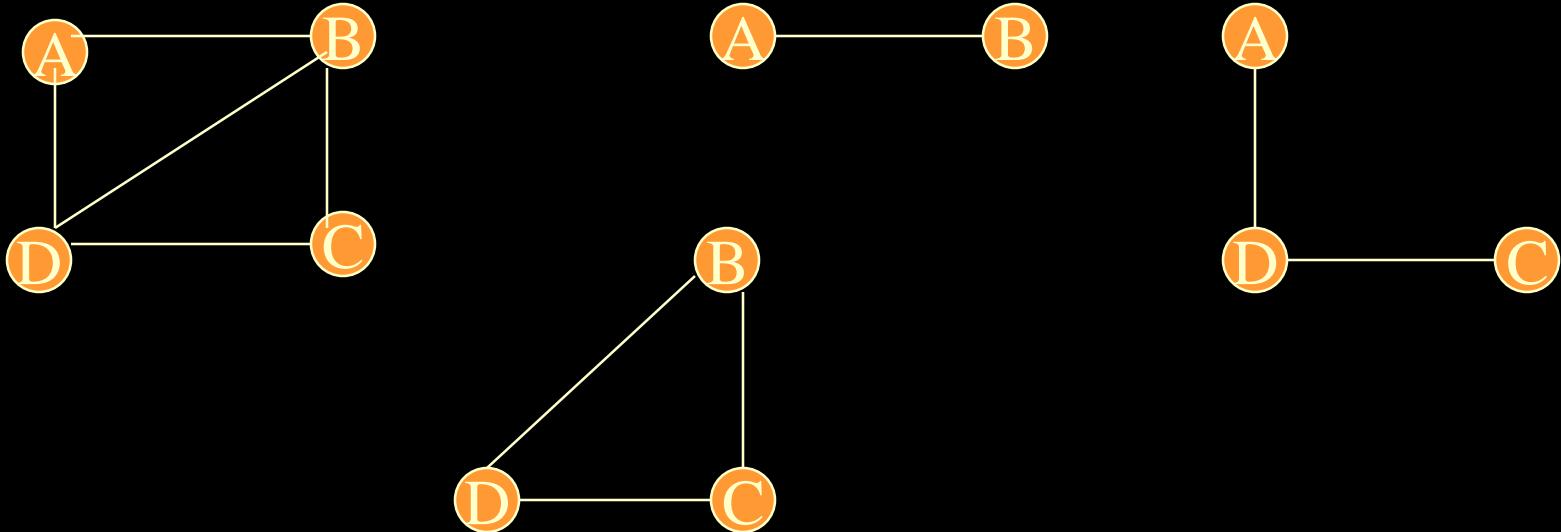
- There is no path from a to h (e, f, or g)
- The subgraphs  $SG_1 = \{a, b, c, d\}$ , and  $SG_2 = \{e, f, g, h\}$  are “connected components”



- An directed graph  $G$  is said to be **strongly connected** if for every pair of distinct vertices  $v_i, v_j$  in  $V(G)$  there is a directed path from  $v_i$  to  $v_j$  and also from  $v_j$  to  $v_i$ .

# *Subgraph:*

- A subgraph of  $G$  is a graph  $G_1$  such that  $V(G_1)$  is a subset of  $v(G)$  and  $E(G_1)$  is a subset of  $E(G)$ .



SUBGRAPHS

# *Degree of Vertex:*

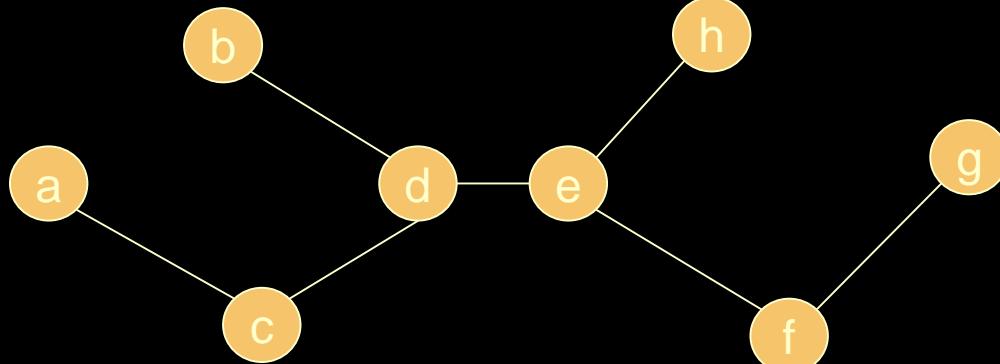
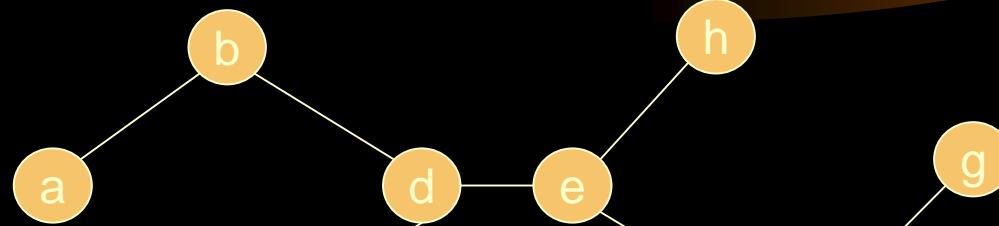
- The total number of edges linked to a vertex is called it's degree.
- In-degree : is the total number of edges coming to that node.
- Out-degree : is the total number of edges going out from that node.
- A vertex that has only outgoing edges and no incoming edges is called a source.
- A vertex that has only incoming edges and no outgoing edges is called a sink.



# Tree



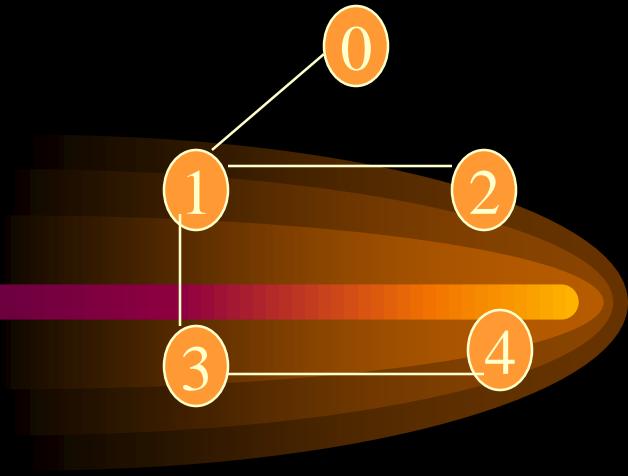
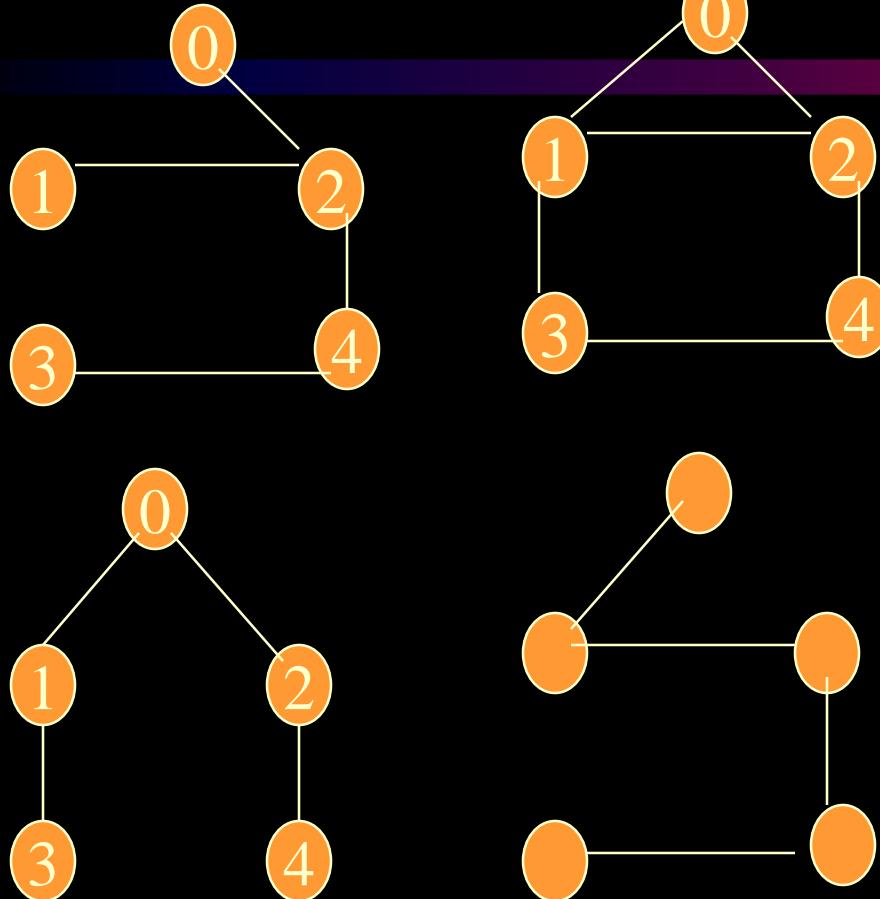
- Connected graph that has no cycles.
- $n$  vertex connected graph with  $n-1$  edges.



# *Spanning Tree*

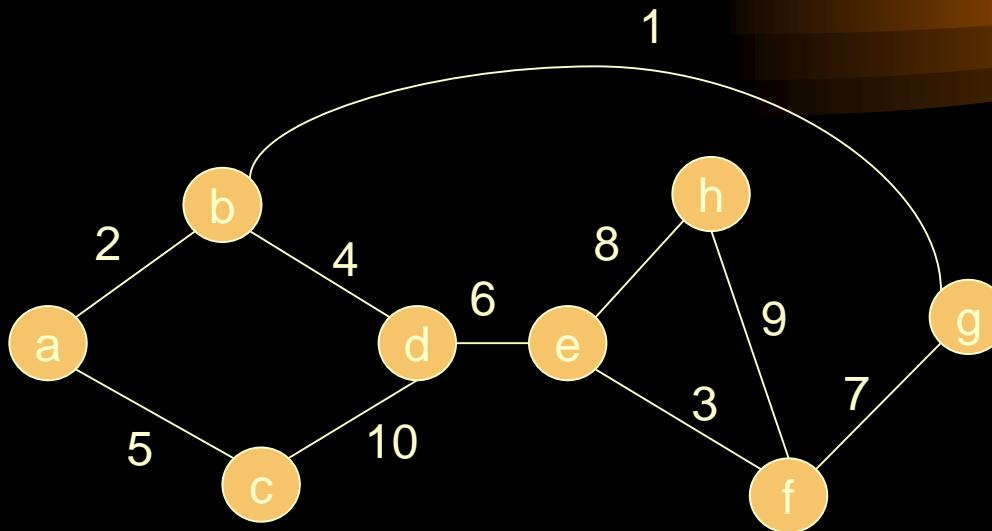
- A spanning tree of a graph  $G$  is a Subgraph that includes all vertices of the original graph and no cycle.
- If graph  $G$  is not connected there is no spanning tree.
  - If original graph has  $n$  vertices, the spanning tree has  $n$  vertices.
  - A graph may have multiple spanning trees.

# *Spanning Tree*



# Minimum Cost Spanning Tree

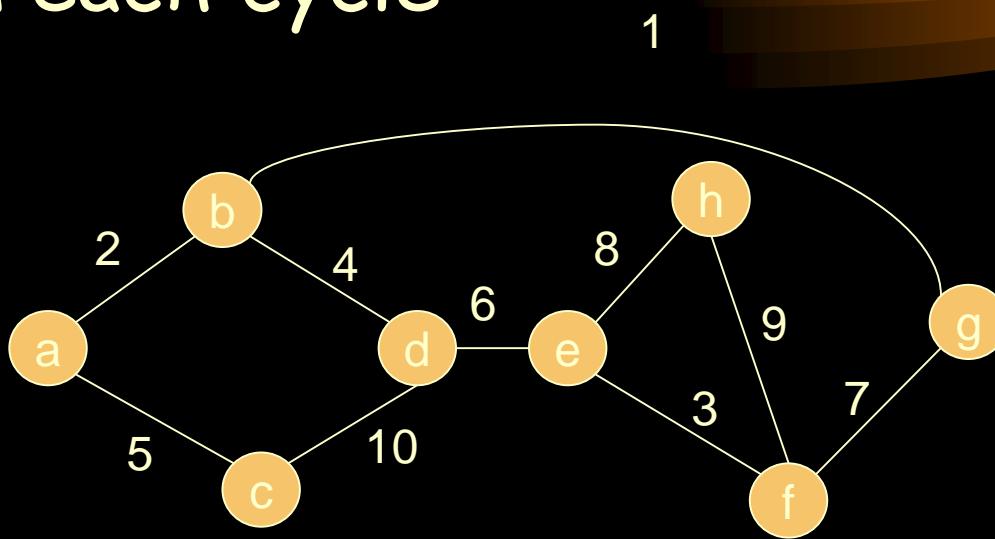
- Reduce graph to tree of smallest cost



- Tree cost is sum of edge weights/costs.

# Minimum Cost Spanning Tree

- Approach - Remove the most expensive edge on each cycle



Spanning tree cost = 29 ()

# *Graph Representations*

- How to store graphs in a computer?
- Adjacency Matrix
- Adjacency Lists

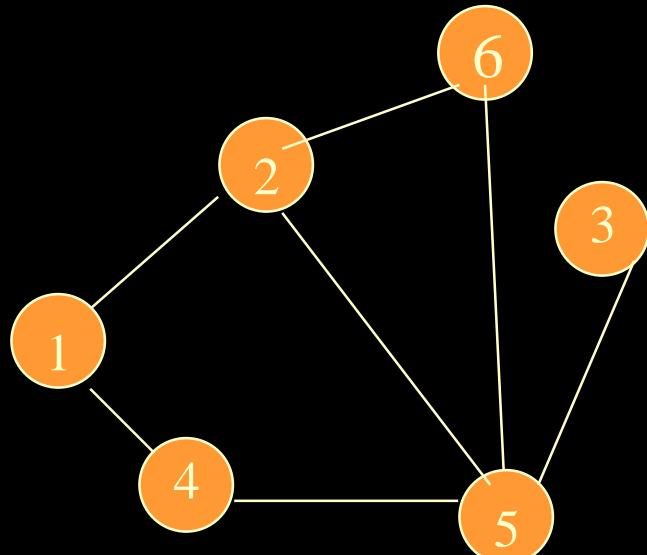
# *Graph Representations*

## Adjacency Matrix :

- If  $G=(V,E)$  has  $n$  vertices  $n \geq 1$ , the adjacency matrix of  $G$  is a 2D  $n \times n$  array  $A$  with the property that  $A(I,J)=1$  if edge  $\langle V_i, V_j \rangle$  exists in  $E(G)$  and  $A(I,J)=0$  if no such edge exists in  $G$ .

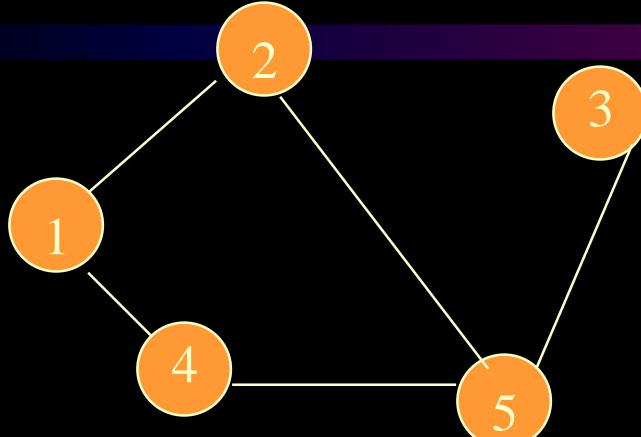
# Adjacency Matrix

- $A(i,j) = 1$  if  $(i,j)$  is an edge



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

# Adjacency Matrix Properties

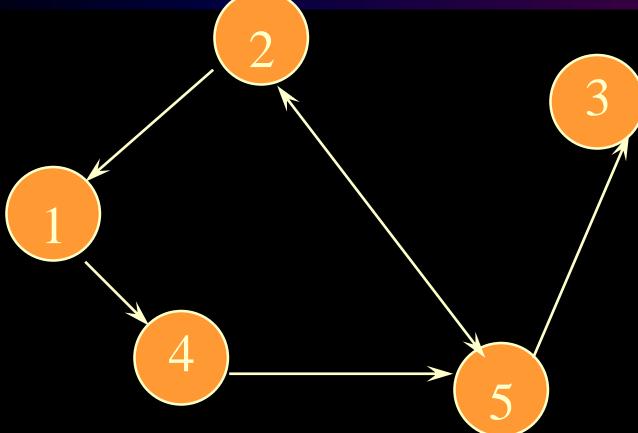


	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

- Diagonal entries are zero.
- Adjacency matrix of an undirected graph is symmetric.

$$A(i,j) = A(j,i) \text{ for all } i \text{ and } j.$$

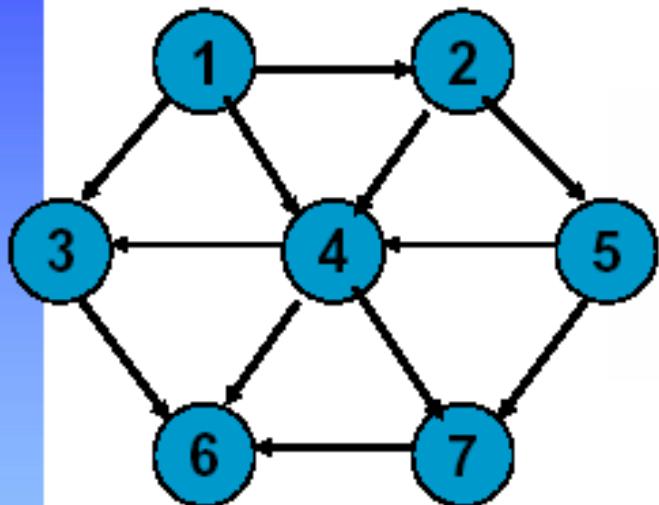
# Adjacency Matrix (Digraph)



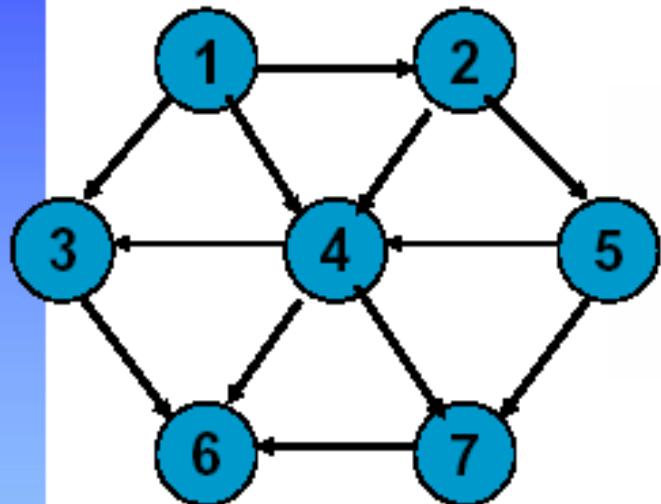
	1	2	3	4	5
1	0	0	0	1	0
2	1	0	0	0	1
3	0	0	0	0	0
4	0	0	0	0	1
5	0	1	1	0	0

- Diagonal entries are zero. (If self-edges are disallowed)
- Adjacency matrix of a digraph need not be symmetric.

# *Adjacency Matrix Representation*



# *Adjacency Matrix Representation*



	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	0
4	0	0	1	0	0	1	1
5	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

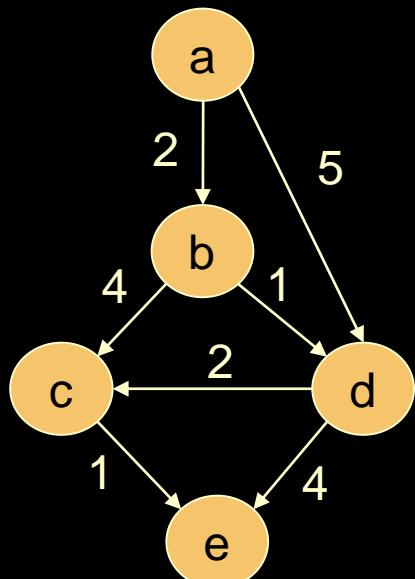
# Adjacency Matrix

- $n^2$  bits of space
- For an undirected graph, may store only lower or upper triangle (exclude diagonal).
- $O(n)$  time to find vertex degree Number of 1's in row.
- Constant time to test if an edge exists
- For directed graph In degree is no. of 1's in column
- Out degree no of 1's in row.

# Weighted Graphs

- Cost adjacency matrix.  
Stores the edge weight in the adjacency matrix
  - $C(i,j) = \text{cost of edge } (i,j)$

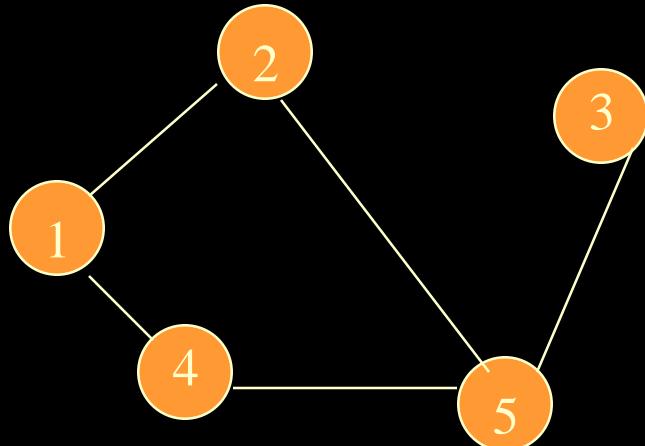
Do you now know how to represent an undirected and weighted graph using a matrix?



$G$	a	b	c	d	e
a	0	2	0	5	0
b	0	0	4	1	0
c	0	0	0	0	1
d	0	0	2	0	4
e	0	0	0	0	0

# Adjacency Lists

- Adjacency list for vertex  $i$  is a linear list of vertices adjacent to vertex  $i$ .



$aList[1] = [2,4]$

$aList[2] = [1,5]$

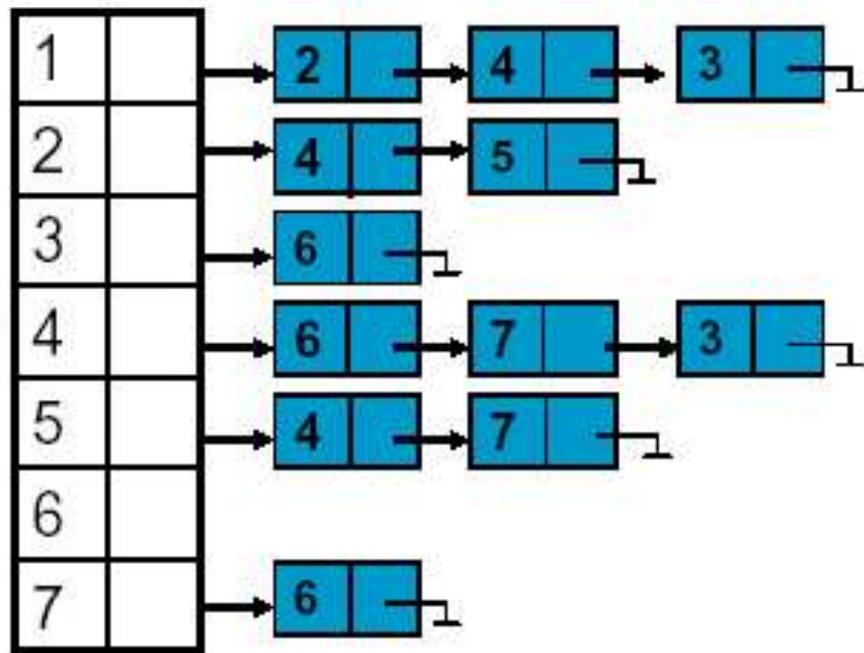
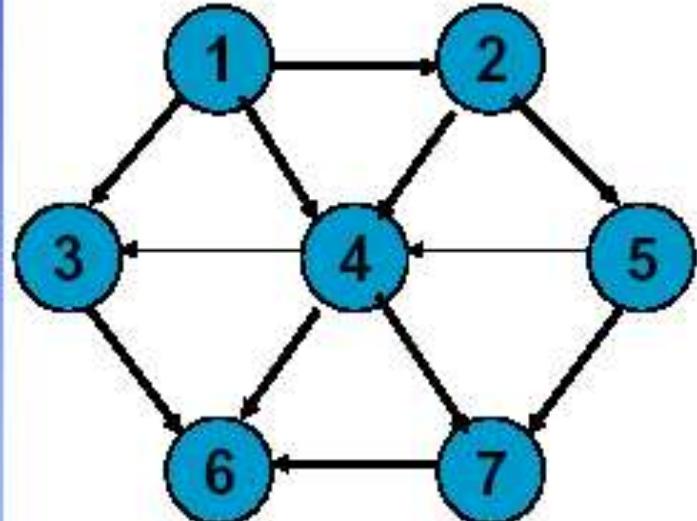
$aList[3] = [5]$

$aList[4] = [5,1]$

$aList[5] = [2,4,3]$

- Adjacency list of a graph with  $n$  nodes can be represented by an array of pointers (nodes). Each pointer points to a linked list of the corresponding vertex.

# Adjacency List Representation



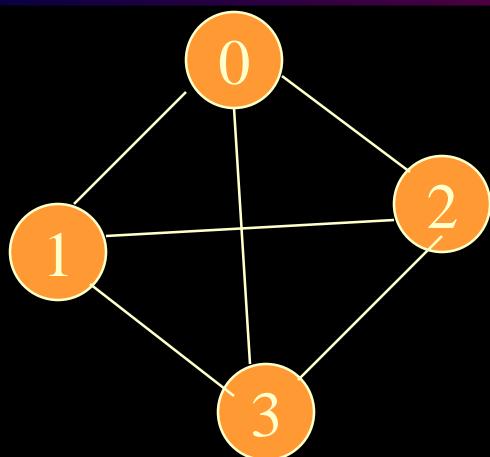
Space:  $O(|V| + |E|)$

# *Adjacency matrix vs. adjacency list representation*

- **Adjacency matrix**
  - Good for dense graphs
  - Memory requirements( Consider all edges)
  - Connectivity between two vertices can be tested quickly
- **Adjacency list**
  - Good for sparse graphs
  - Memory requirements(Consider only connected edges)
  - Vertices adjacent to another vertex can be found quickly

# *Solve by using adjacency list:*

- 1.

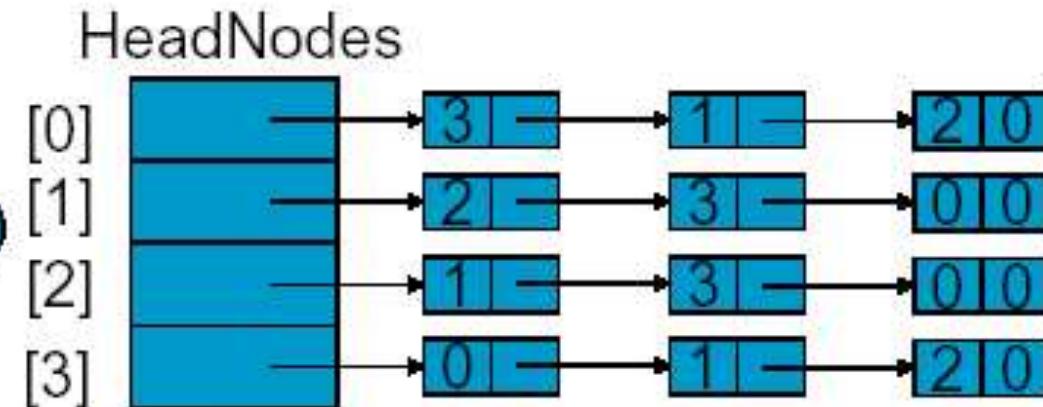
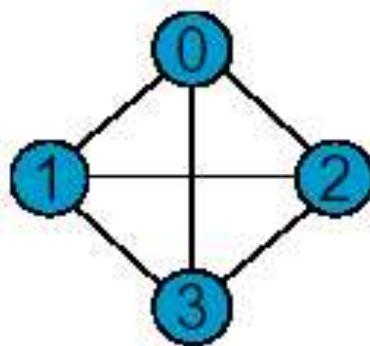


- 2.

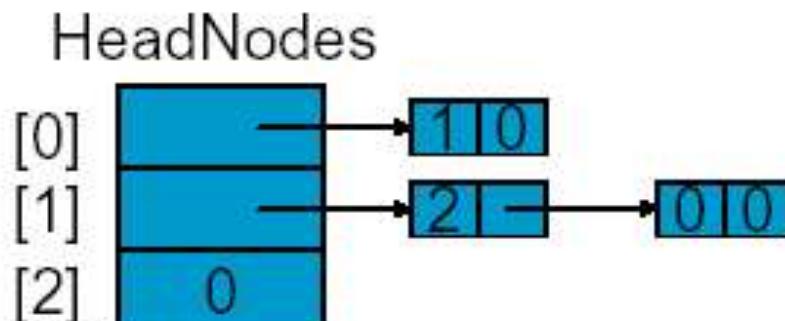


0
1
2
3

# *Adjacent Lists*



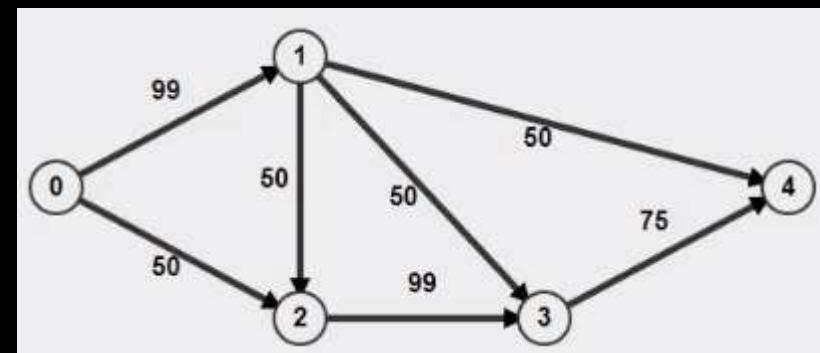
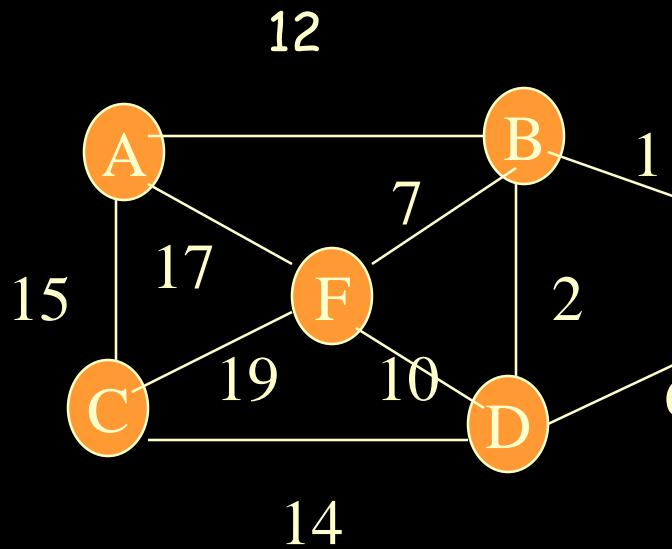
(a)  $G_1$



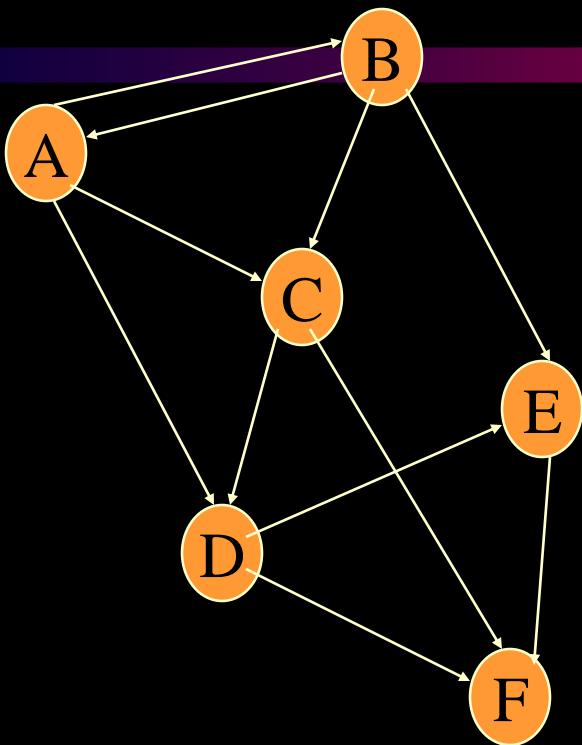
(b)  $G_3$

- Adjacency list representation of graph is very efficient when graph has large number of vertices but very few edges.
- Degree of node in undirected graph is given by length of corresponding link list
- Out degree is calculated from length of link list
- In degree requires lot of comparisons. List pointed by all the vertices must be examined (how many times node is appearing in representation that is a in degree of that node )
- For weighted graph link list consist of weight also.

*Construct adjacency matrix and adjacency list for the following graphs:*



*Construct adjacency matrix and adjacency list for the following graph:*



# *Graph Traversals :*

- Traversal of graph means visit each node & visit it exactly once.
- Depth First Search
- Breadth First Search

# *Depth First Search with recursion*

$n <$ -Number of nodes

Initialize Visited[ ](all vertices) to zero.

For ( $i=0; i < n; i++$ )

    Visited[i]=0;

    Void DFS(vertex i)

{

    Visited[i]=1;

    For each vertex w adjacent to i do

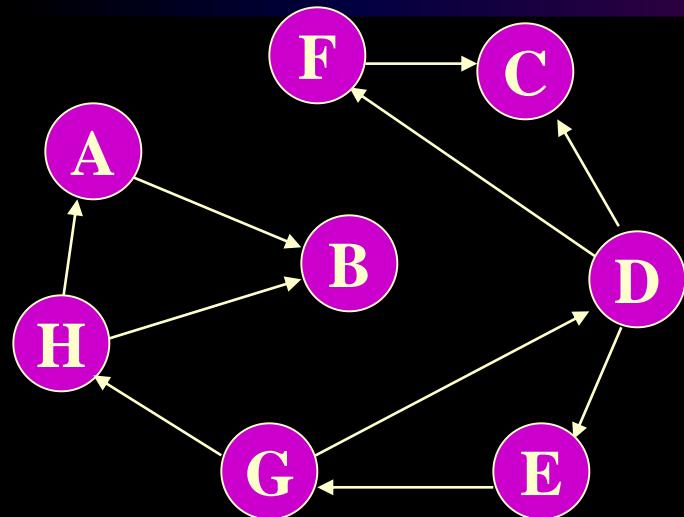
        If visited(w)=0 then call DFS(w)

    End

}

# Walk-Through

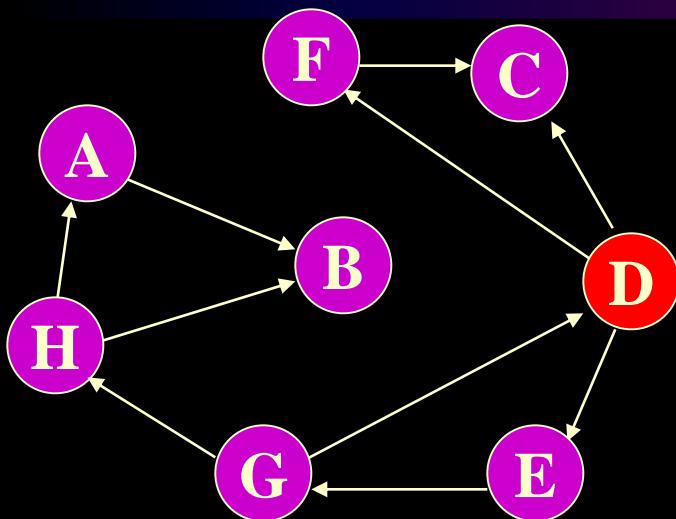
Visited  
Array



A
B
C
D
E
F
G
H

Task: Conduct a depth-first search of the graph starting with node D

# Walk-Through



Visited

Array
B
C
D ✓
E
F
G
H

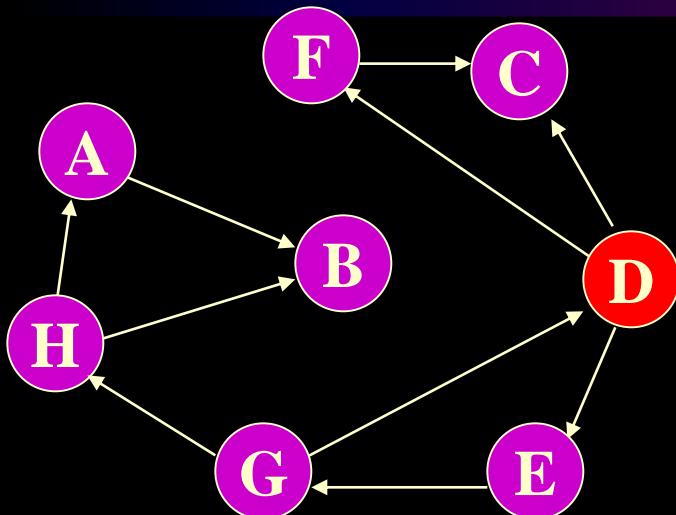
D

The order nodes are visited:

D

Visit D

# Walk-Through



Visited

Array
B
C
D ✓
E
F
G
H

D

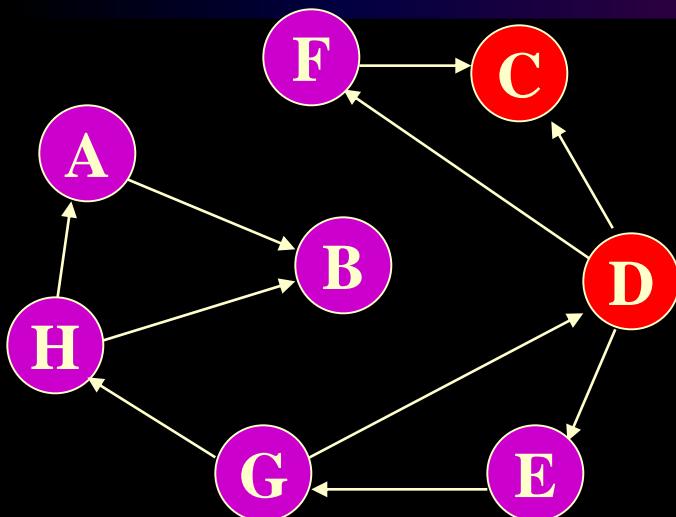
The order nodes are visited:

D

Consider nodes adjacent to D, decide to visit C first

Prefer **Alphabetical** order

# Walk-Through



The order nodes are visited:  
D, C

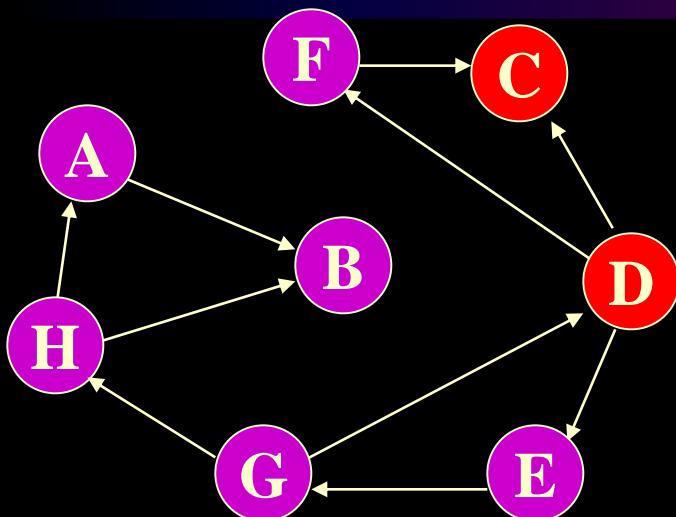
Visited

Array
B
C ✓
D ✓
E
F
G
H

C  
D

Visit C

# Walk-Through



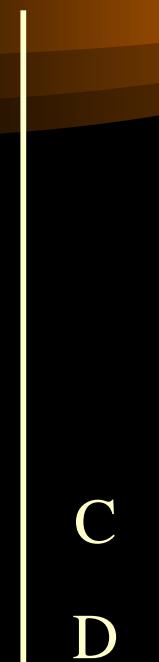
The order nodes are visited:

D, C

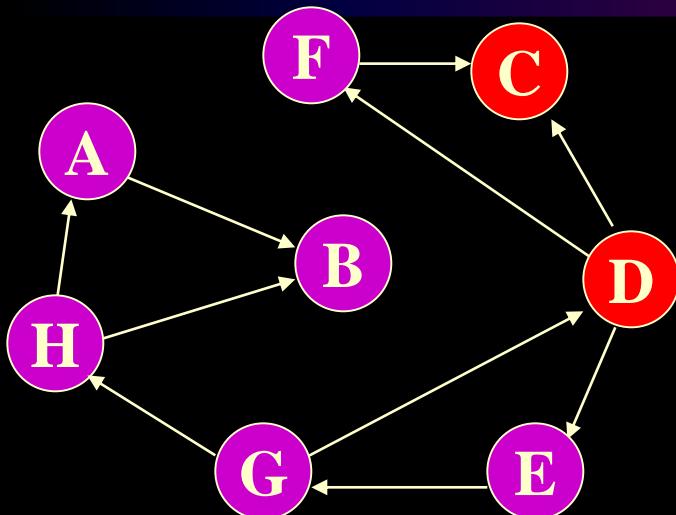
No nodes adjacent to C; cannot continue → *backtrack*, i.e., pop stack and restore previous state

Visited

Array
B
C ✓
D ✓
E
F
G
H



# Walk-Through

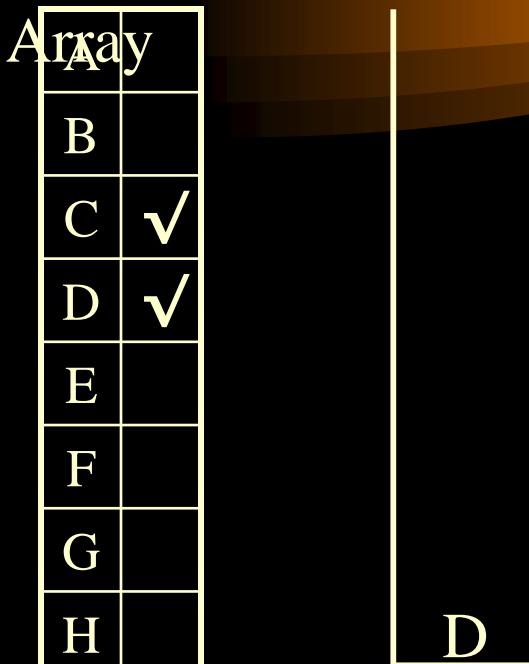


The order nodes are visited:

D, C

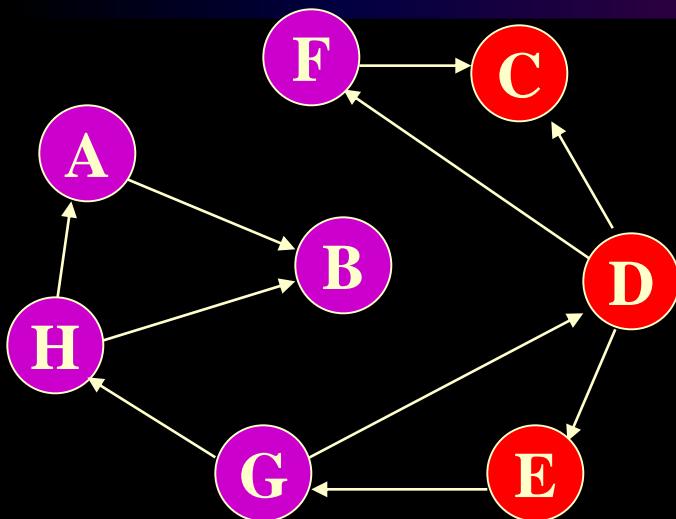
Visited

Array
B
C ✓
D ✓
E
F
G
H



**Back to D – C has been visited,  
decide to visit E next**

# Walk-Through



Visited

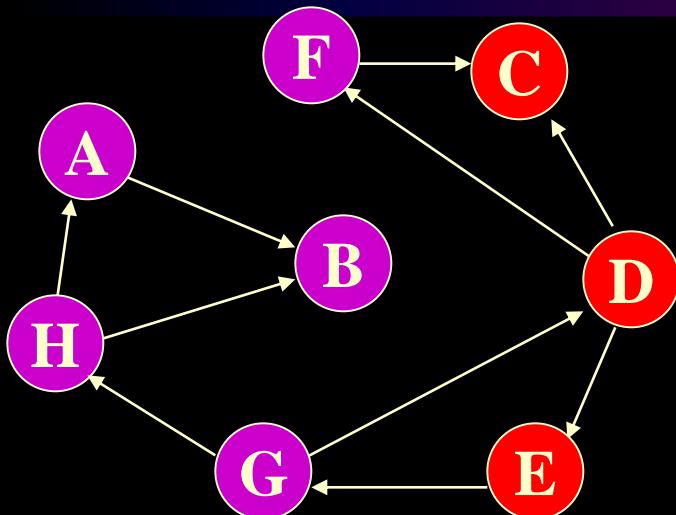
Array
B
C ✓
D ✓
E ✓
F
G
H



The order nodes are visited:

D, C, E

# Walk-Through



The order nodes are visited:  
D, C, E

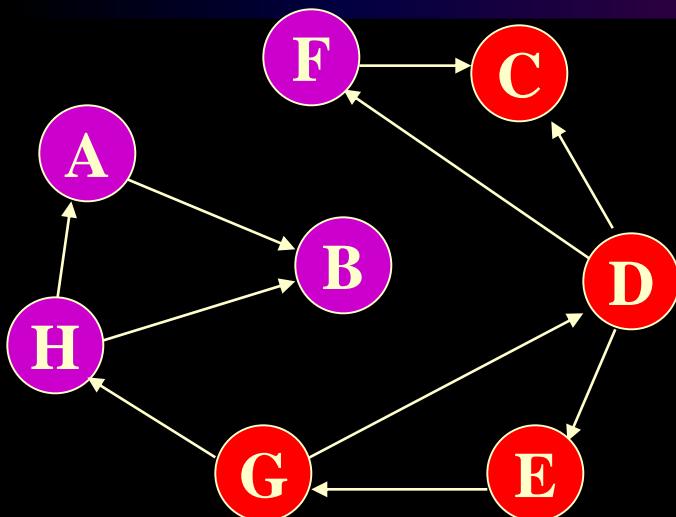
Visited

Array
B
C ✓
D ✓
E ✓
F
G
H

E  
D

**Only G is adjacent to E**

# Walk-Through



The order nodes are visited:  
D, C, E, G

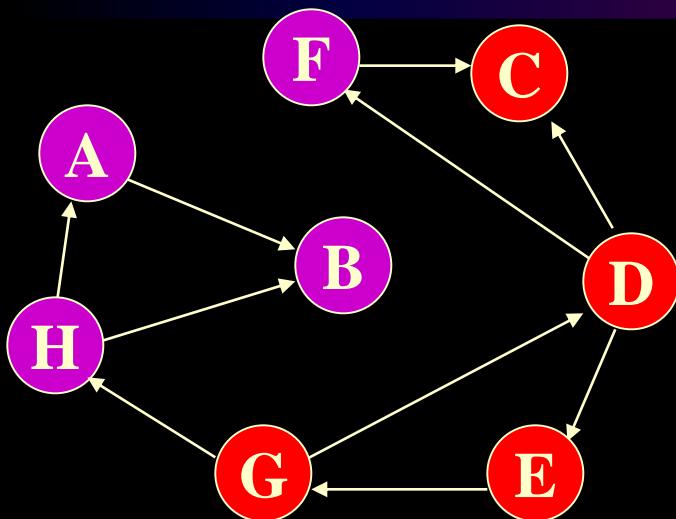
Visited

Array
B
C ✓
D ✓
E ✓
F
G ✓
H

G  
E  
D

Visit G

# Walk-Through



Visited

Array
B
C ✓
D ✓
E ✓
F
G ✓
H

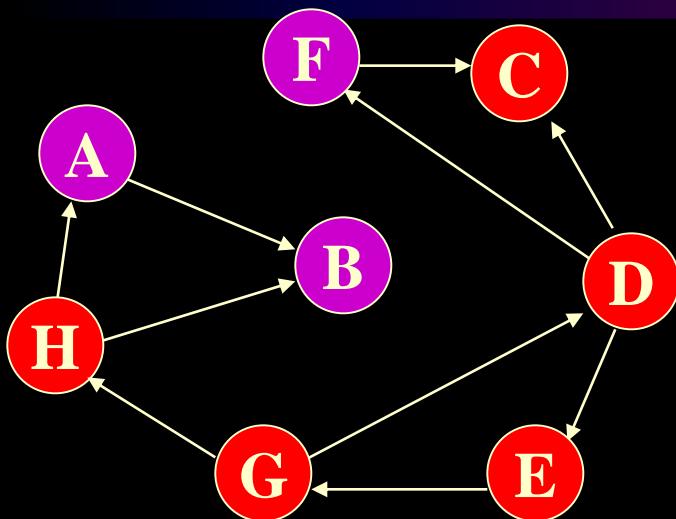
G
E
D

The order nodes are visited:

D, C, E, G

**Nodes D and H are adjacent to G. D has already been visited. Decide to visit H.**

# Walk-Through



The order nodes are visited:  
D, C, E, G, H

Visited

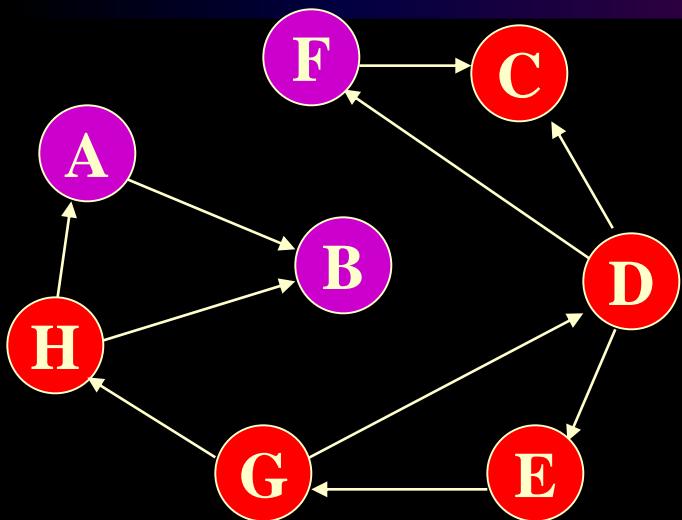
Array
B
C ✓
D ✓
E ✓
F
G ✓
H ✓

H  
G  
E  
D

Visit H

# Walk-Through

Visited  
Array



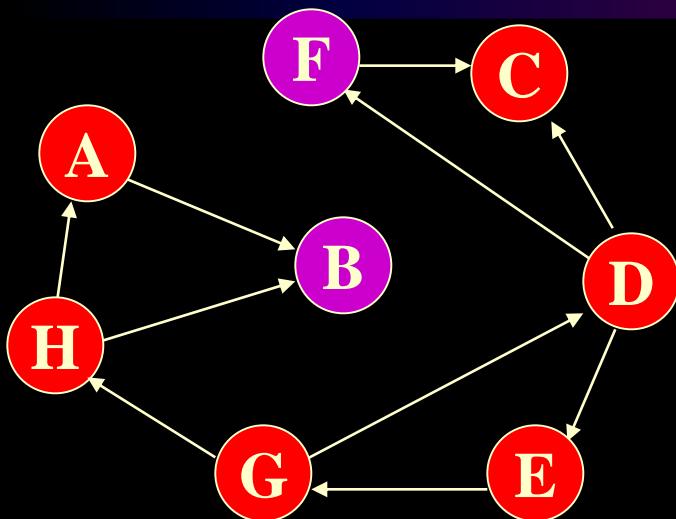
A	
B	
C	✓
D	✓
E	✓
F	
G	✓
H	✓

H
G
E
D

The order nodes are visited:  
D, C, E, G, H

**Nodes A and B are adjacent to H. Decide to visit A next.**

# Walk-Through



The order nodes are visited:  
D, C, E, G, H, A

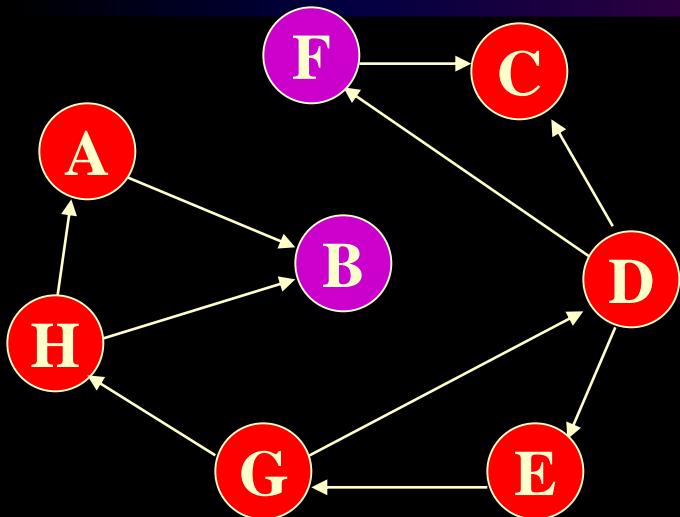
Visited

Array	✓
B	
C	✓
D	✓
E	✓
F	
G	✓
H	✓

A  
H  
G  
E  
D

Visit A

# Walk-Through



The order nodes are visited:  
D, C, E, G, H, A

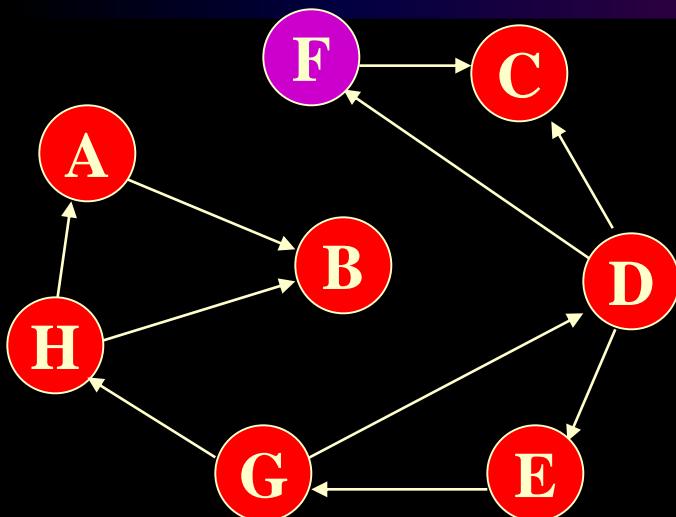
Visited

Array	✓
B	
C	✓
D	✓
E	✓
F	
G	✓
H	✓

A
H
G
E
D

Only Node B is adjacent to A.  
Decide to visit B next.

# Walk-Through



Visited

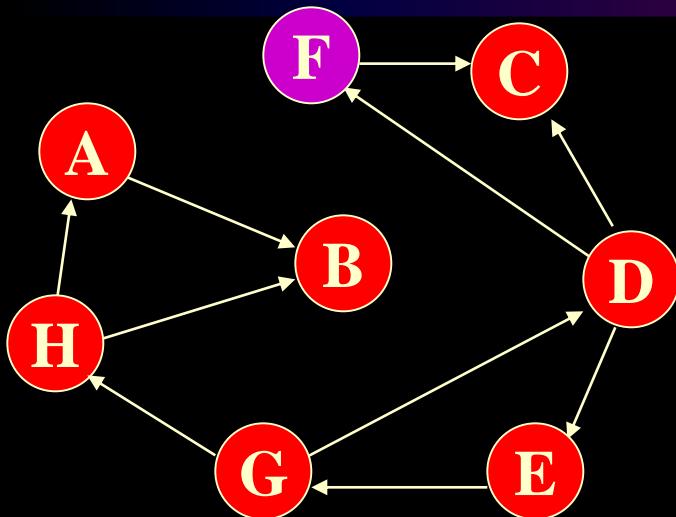
Array	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓

B  
A  
H  
G  
E  
D

The order nodes are visited:  
D, C, E, G, H, A, B

Visit B

# Walk-Through



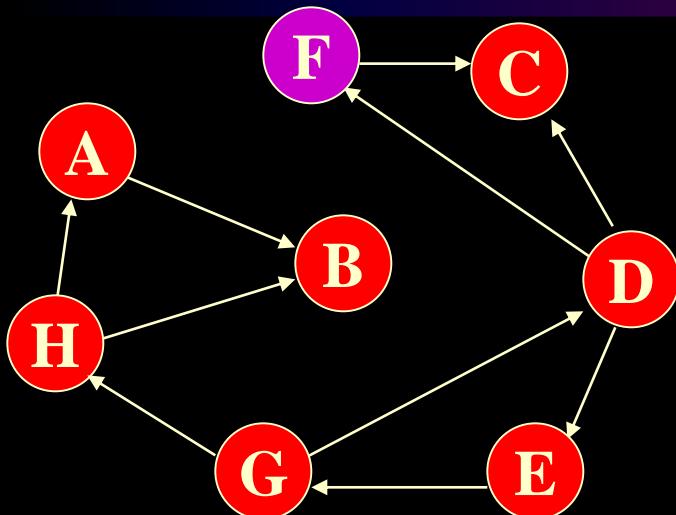
The order nodes are visited:  
D, C, E, G, H, A, B

Visited

Array	Visited
A	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓

No unvisited nodes adjacent to  
**B.** Backtrack (pop the stack).

# Walk-Through



The order nodes are visited:

D, C, E, G, H, A, B

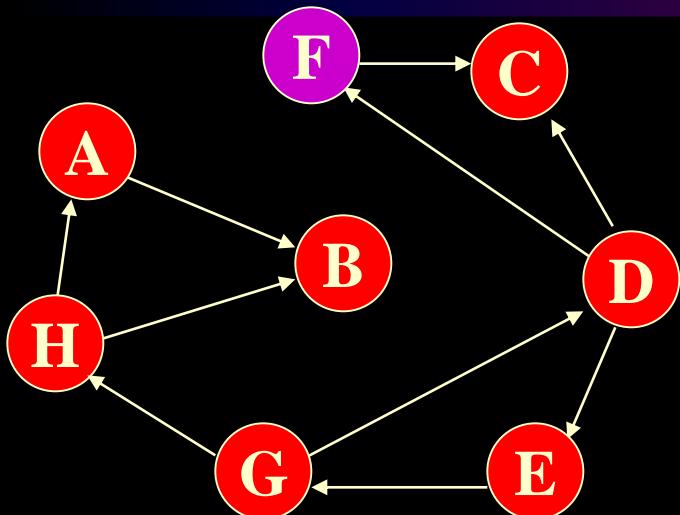
Visited

Array	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓



**No unvisited nodes adjacent to A. Backtrack (pop the stack).**

# Walk-Through



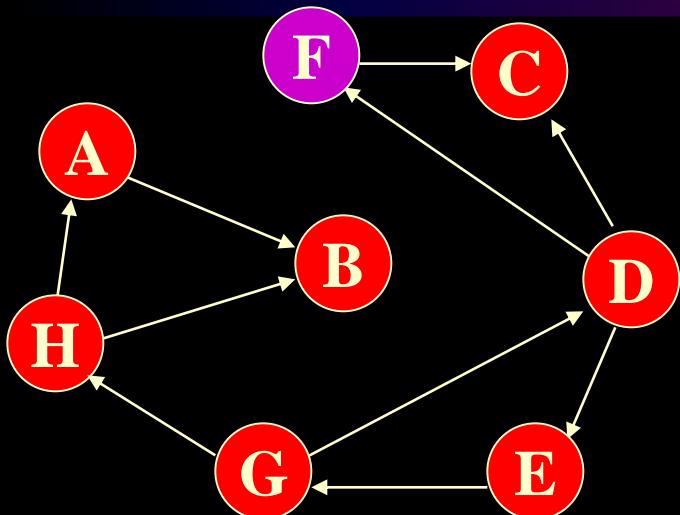
The order nodes are visited:  
D, C, E, G, H, A, B

Visited

Array	Visited
A	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓

No unvisited nodes adjacent to  
H. Backtrack (pop the  
stack).

# Walk-Through



Visited

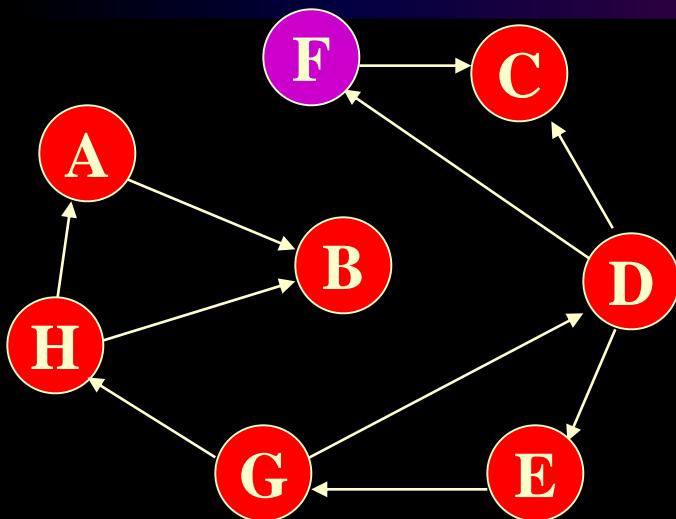
Array	Visited
A	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓

The order nodes are visited:

D, C, E, G, H, A, B

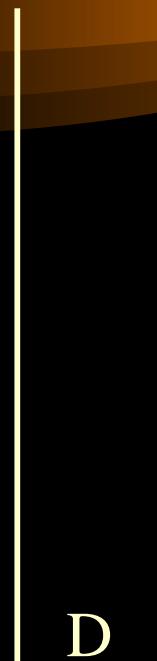
No unvisited nodes adjacent to G. Backtrack (pop the stack).

# Walk-Through



Visited

Array	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓

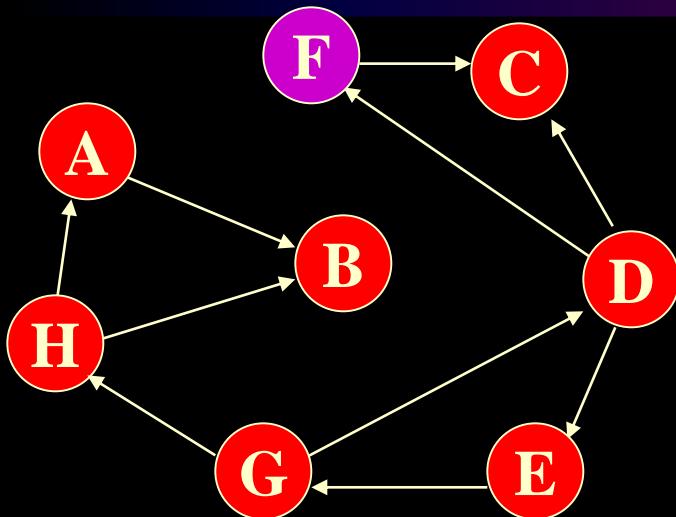


The order nodes are visited:

D, C, E, G, H, A, B

**No unvisited nodes adjacent to E. Backtrack (pop the stack).**

# Walk-Through



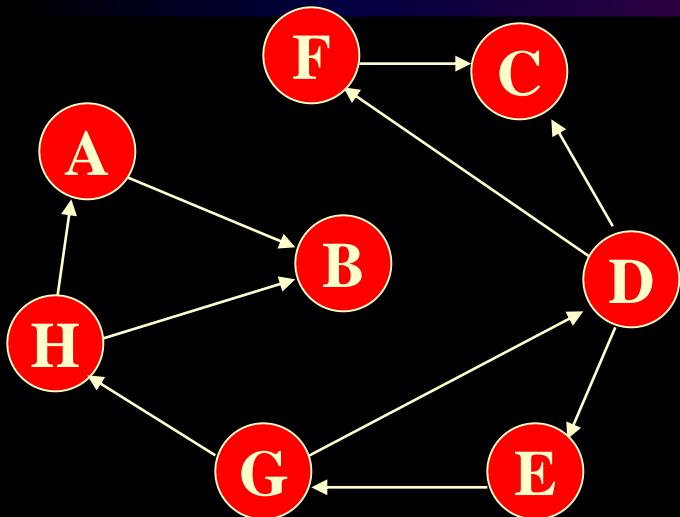
The order nodes are visited:  
D, C, E, G, H, A, B

Visited

Array	Visited
A	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓

F is unvisited and is adjacent to  
D. Decide to visit F next.

# Walk-Through



The order nodes are visited:  
D, C, E, G, H, A, B, F

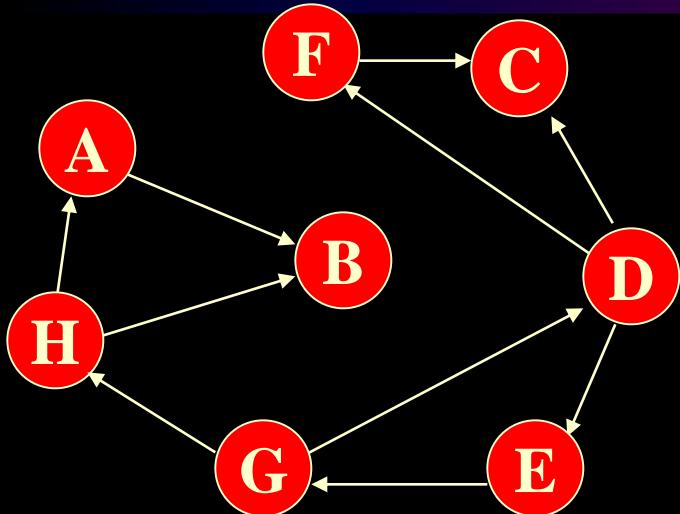
Visited  
Array

A	✓
B	✓
C	✓
D	✓
E	✓
F	✓
G	✓
H	✓

F  
D

Visit F

# Walk-Through



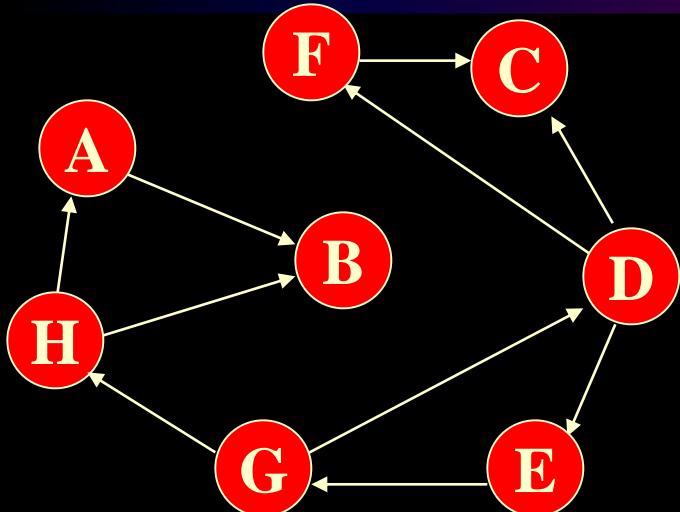
The order nodes are visited:  
D, C, E, G, H, A, B, F

Visited

Array	Visited
A	✓
B	✓
C	✓
D	✓
E	✓
F	✓
G	✓
H	✓

No unvisited nodes adjacent to  
F. Backtrack.

# Walk-Through



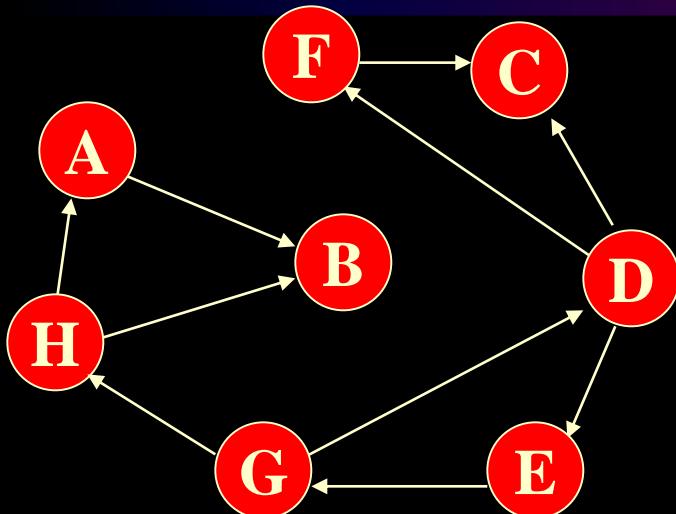
The order nodes are visited:  
D, C, E, G, H, A, B, F

Visited

Array	✓
B	✓
C	✓
D	✓
E	✓
F	✓
G	✓
H	✓

No unvisited nodes adjacent to  
D. Backtrack.

# Walk-Through

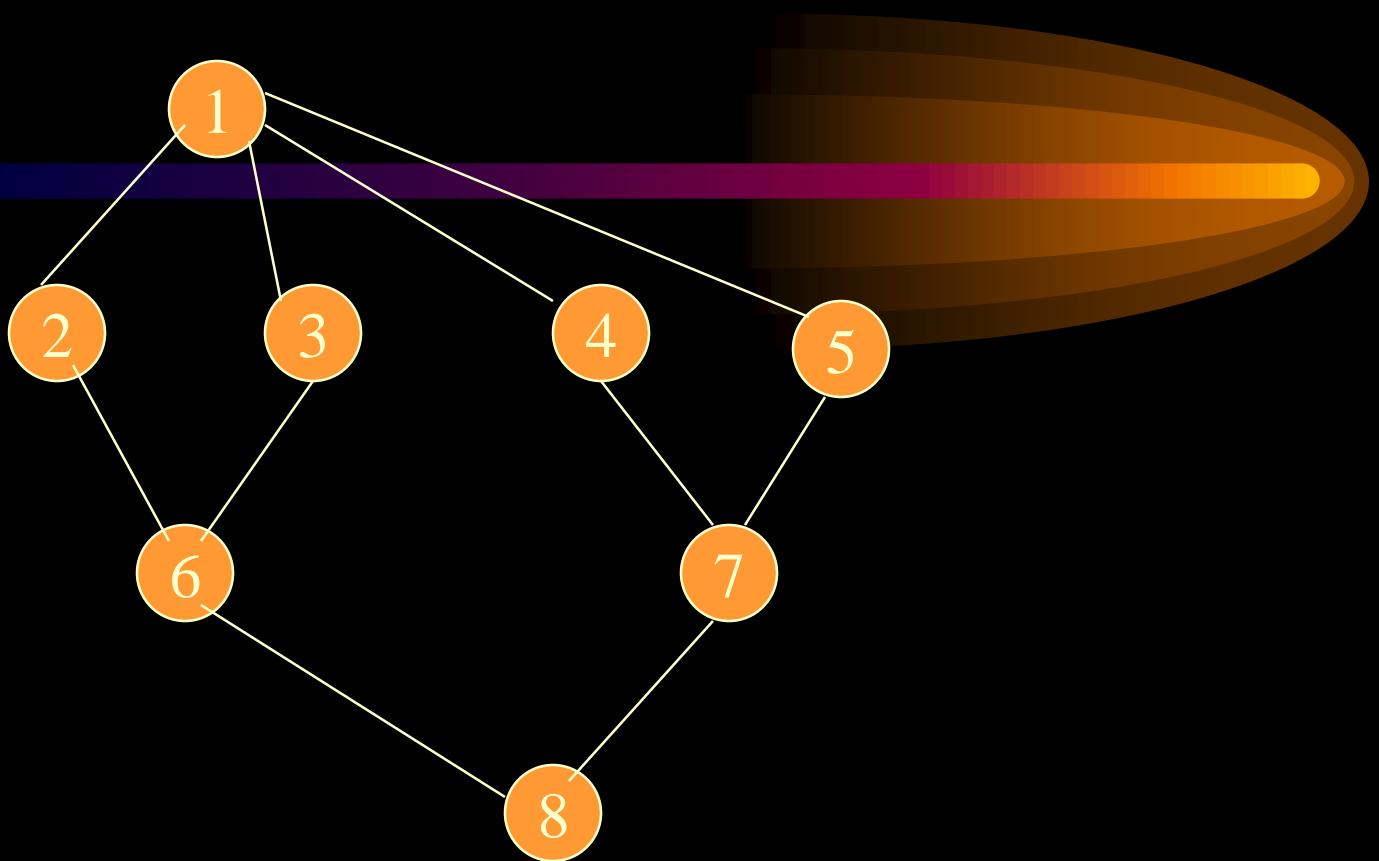


The order nodes are visited:  
D, C, E, G, H, A, B, F

Visited

Array	✓
B	✓
C	✓
D	✓
E	✓
F	✓
G	✓
H	✓

Stack is empty. Depth-first traversal is done.



DFS traversal sequence is 1,2,6,3,8,7,4,5

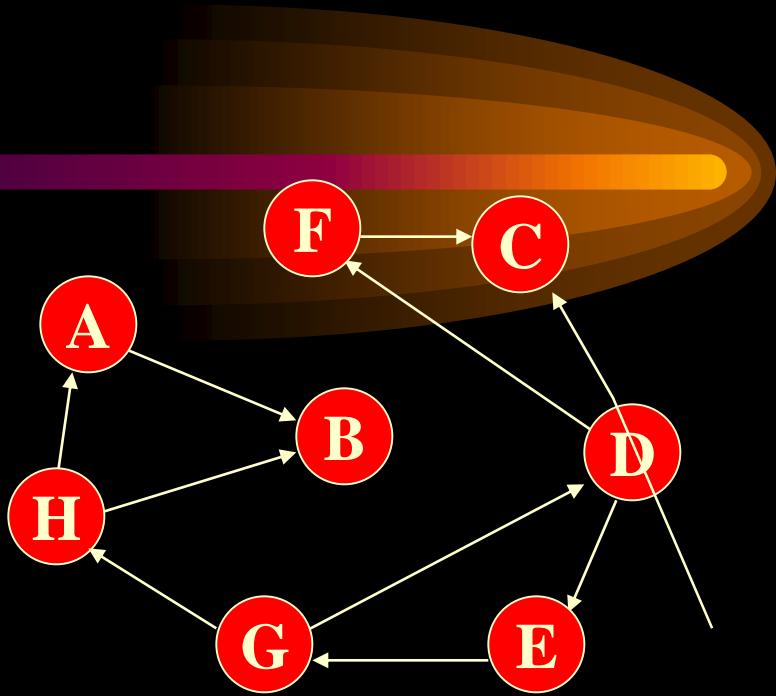
# *Algorithm for Depth first search*

- Step 1 : Select any node in the graph. Push it in the stack. pop same node. Mark this node as visited.
- Step 2: Find the adjacent nodes of that node. push them in the stack if it is not visited. pop the node from top of stack. Visit this new node. Mark this node as visited.
- Step 3 : Repeat step 2 till stack becomes empty.
- Step 4: Repeat above steps if there are any more nodes which are still unvisited.

```

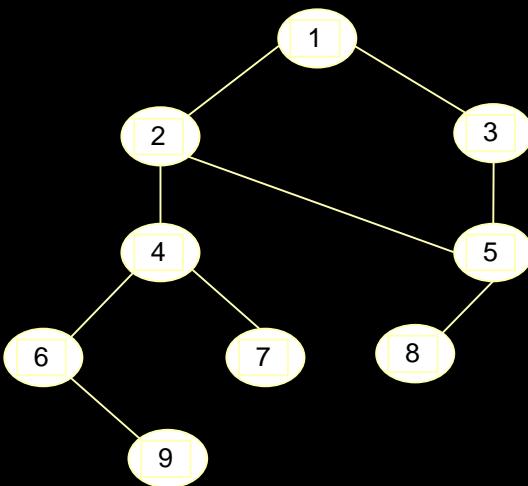
• Dfs_nonrecursive(vertex i)
• {
•     vertex w;
•     Stack s;
•     initialize the stack s;
•     push i in the stack s;
•     while(stack is not empty)
•     {
•         i=pop(s);
•         if(!visited[i])
•         {
•             visited[i]=1;
•             for each w adjacent to i
•                 if(! Visited[w])
•                     push w in the stack s;
•         }
•     }

```



# *Depth first search*

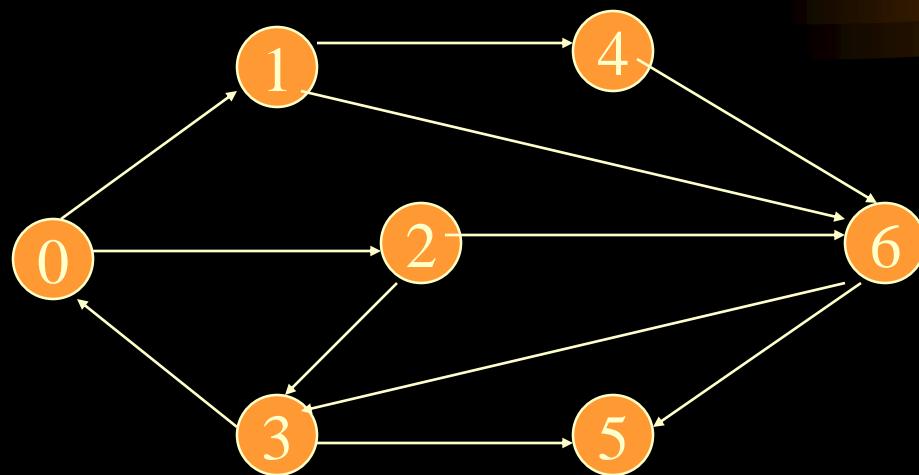
- Find out DFS for following graph.



1,2,4,6,9,7,5,3,8

1 2 4 6 9 7 5 3 8

# Show non recursive dfs on following graph.



BFS  
0, 1, 2, 4, 6, 3, 5

OUTPUT: 0 1 4 6 3 5 2  
(DFS)

# *Algorithm for Breadth first search*

- Step 1 : Start with any node mark it as visited. place it into the queue.
- Step 2: Delete element from the queue. Find the adjacent nodes to the node marked in step 1 and place it in the queue.
- Step 3: Visit the node at the front of the queue. Delete from the queue, place it's adjacent nodes in the queue.
- Step 4 : Repeat step 3 till the queue is not empty.
- Step 5: Stop.

# Breadth First Search

Visited[ ](all vertices) is initialized to zero

Void BFS(int v)

{

Q: A Q type variable;

Initialize Q;

Visited[v]=1

Add the vertex V to queue Q

While(Q is not empty)

{

v<-delete an element from the Q.

for all vertices w adjacent to v

{

if (!visited[w]) // add w to queue

{

visited(w)=1

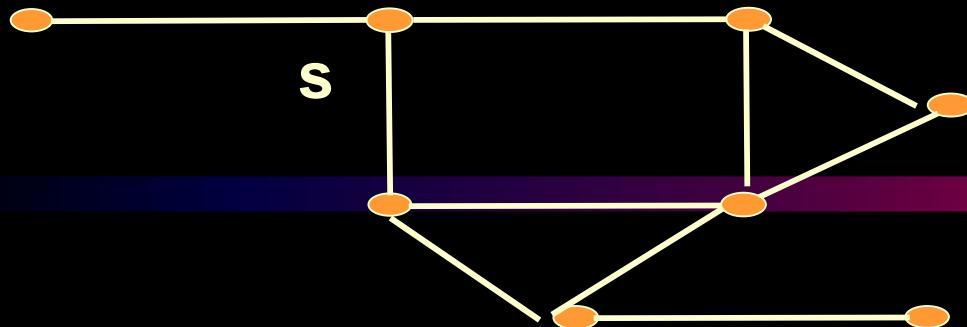
add the vertex w to Q.

}

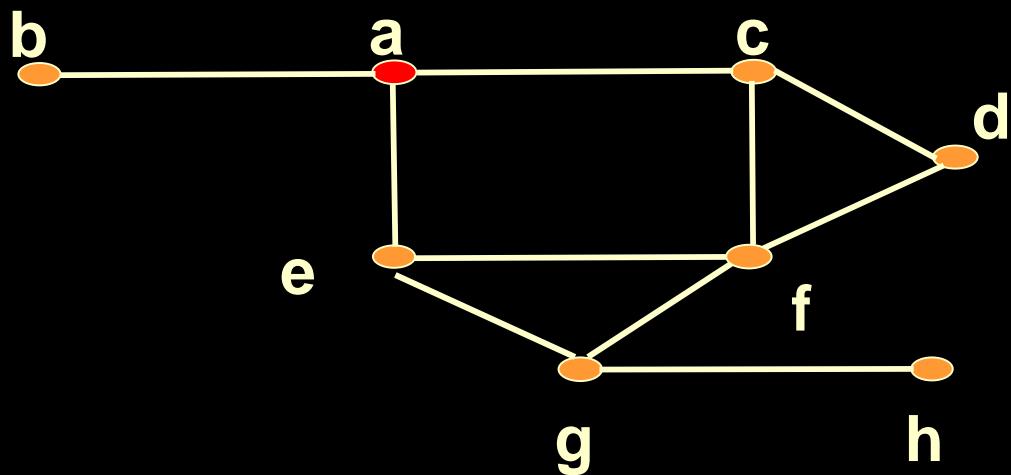
}

}

## Example execution of BFS

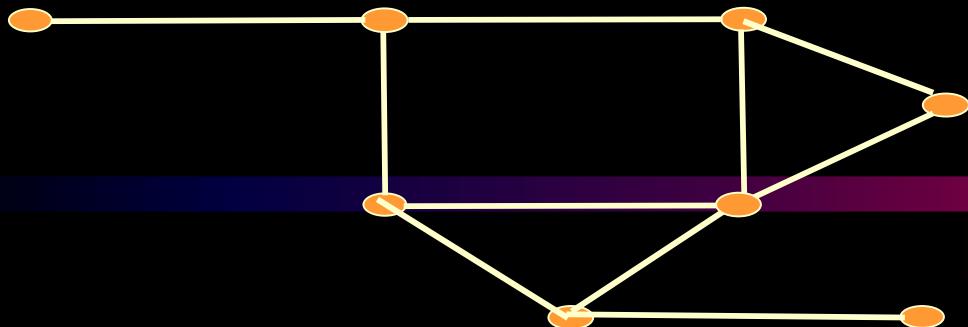


undirected  
graph  $G$

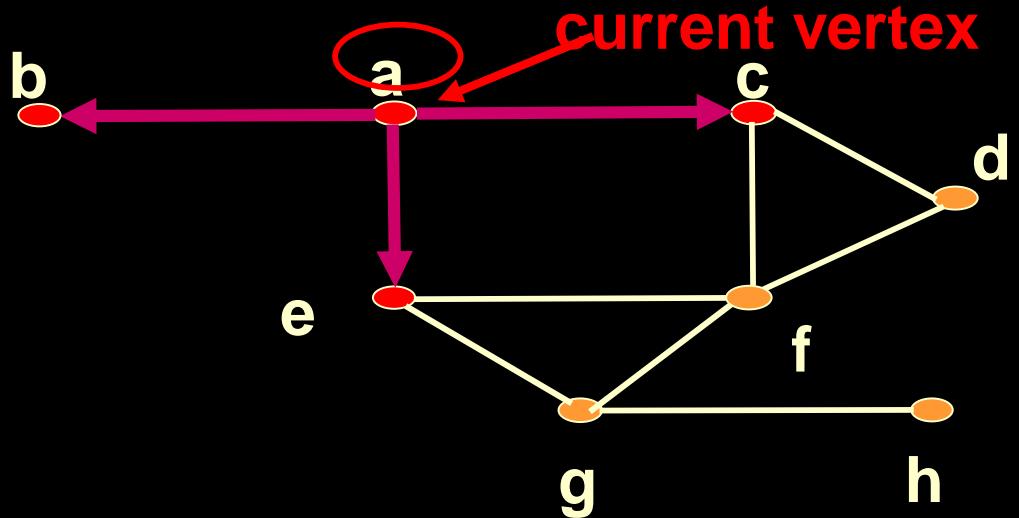


Queue  
 $= \langle a \rangle$

## Example execution of BFS



undirected  
graph  $G$



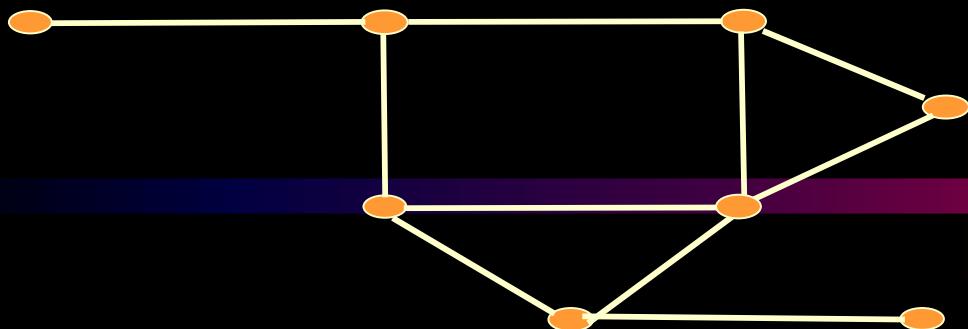
Queue  
 $= \langle b, c, e \rangle$

v  
3/3/2022

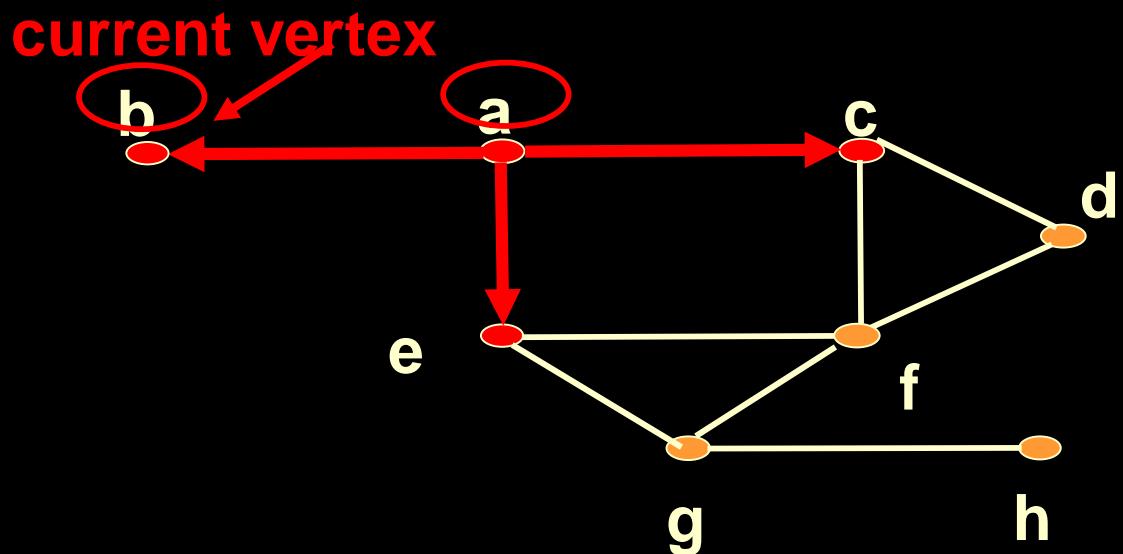
means vertex v has been processed

Graphs

## Example execution of BFS



undirected  
graph  $G$



Queue  
 $= \langle c, e \rangle$

v

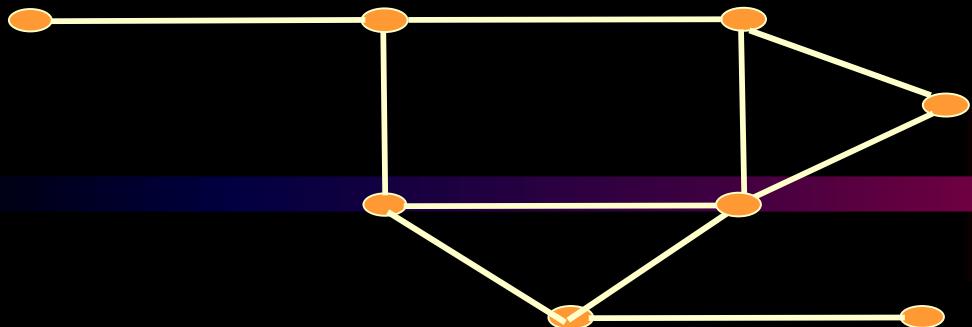
means vertex v has been processed

3/3/2022

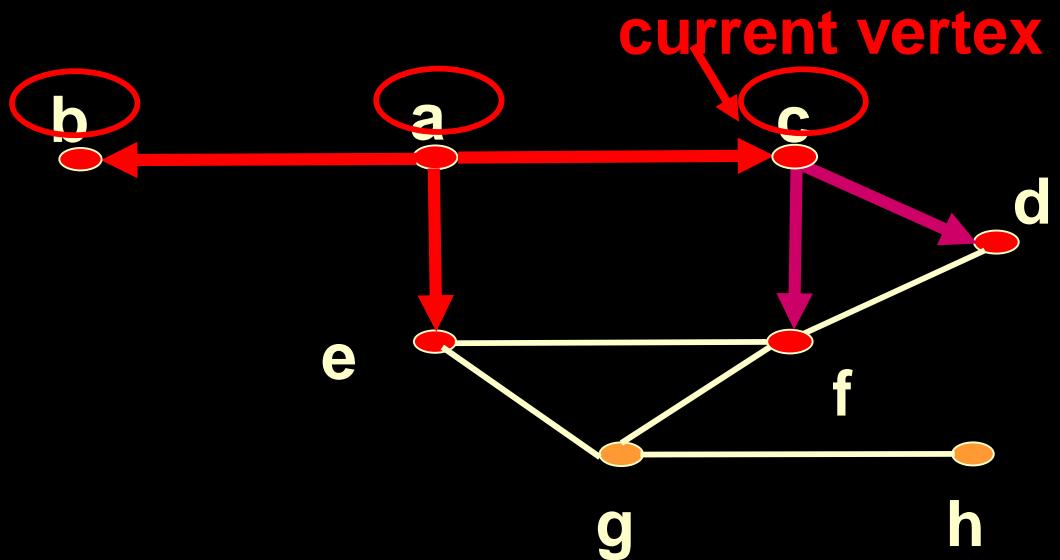
Graphs

78

## Example execution of BFS



undirected  
graph  $G$



Queue  
 $= \langle e, d, f \rangle$

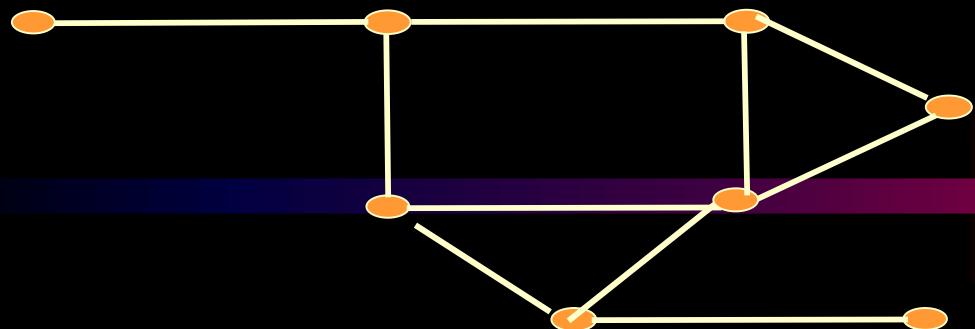


means vertex  $v$  has been processed

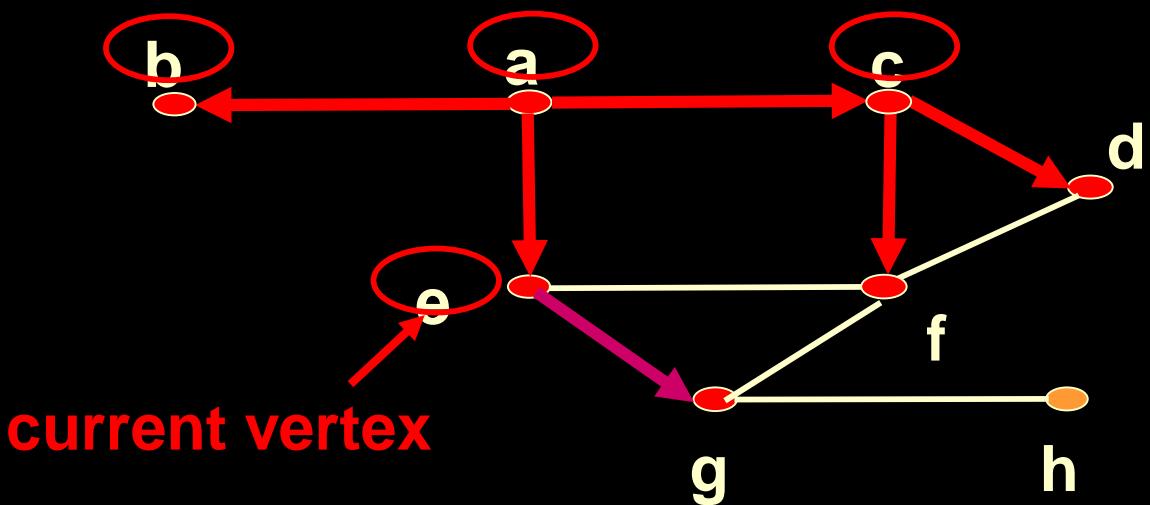
3/3/2022

Graphs

## Example execution of BFS



undirected  
graph  $G$



Queue  
 $= \langle d, f, g \rangle$

current vertex

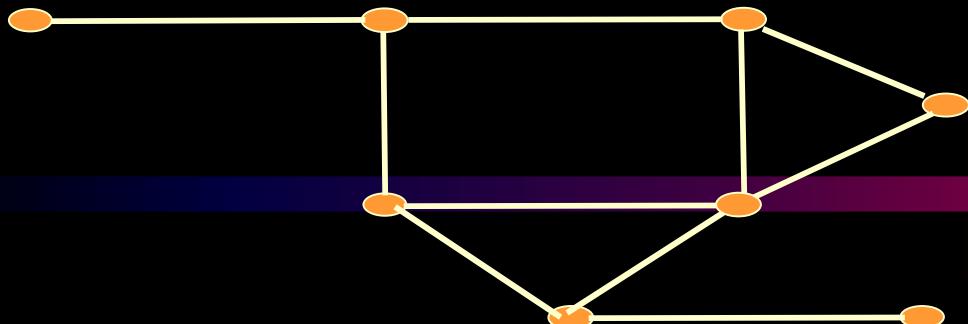


means vertex  $v$  has been processed

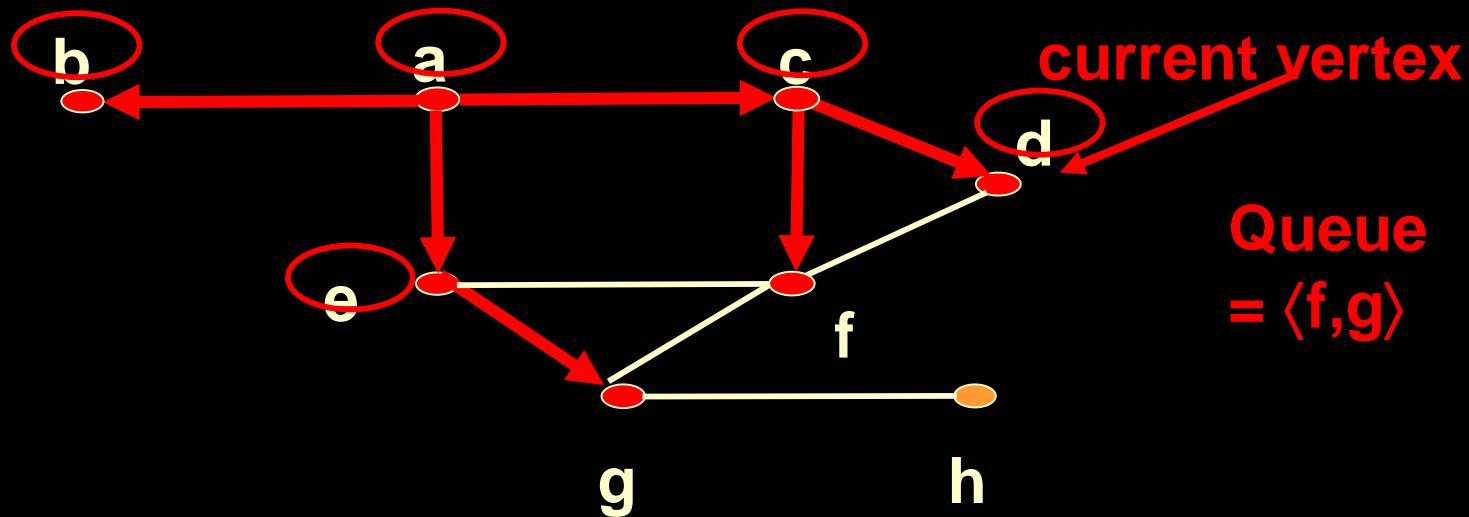
3/3/2022

Graphs

## Example execution of BFS



undirected  
graph  $G$



v

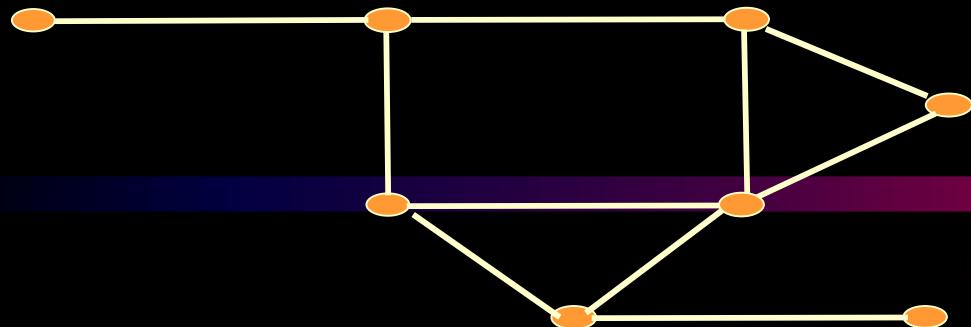
means vertex v has been processed

3/3/2022

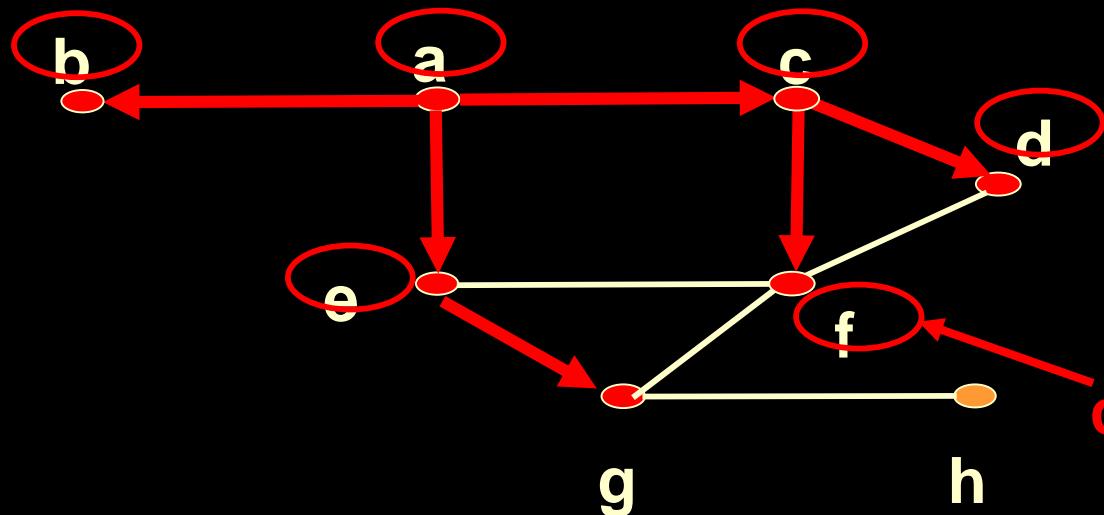
Graphs

81

## Example execution of BFS



undirected  
graph  $G$



Queue  
 $= \langle g \rangle$

current vertex

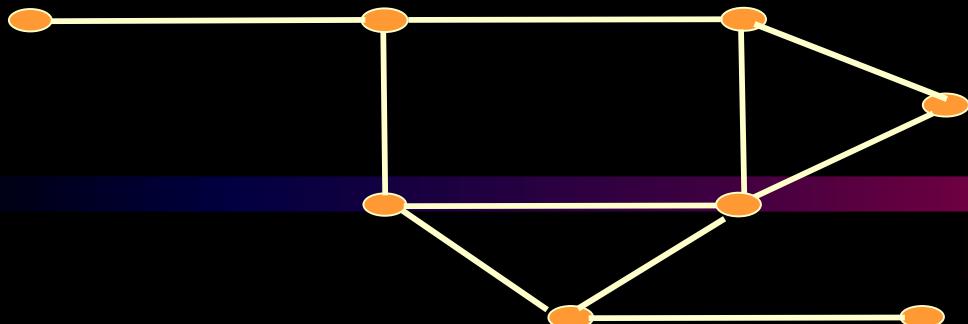


means vertex  $v$  has been processed

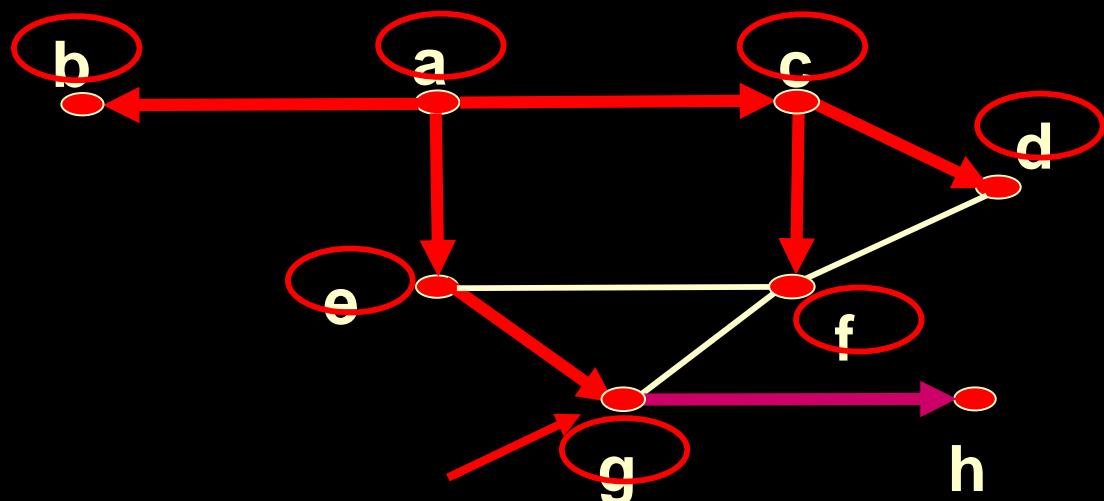
3/3/2022

Graphs

## Example execution of BFS



undirected  
graph  $G$



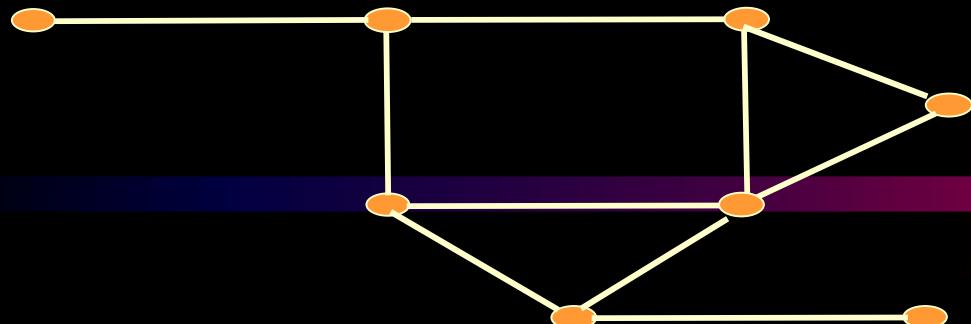
Queue  
 $= \langle h \rangle$

current vertex

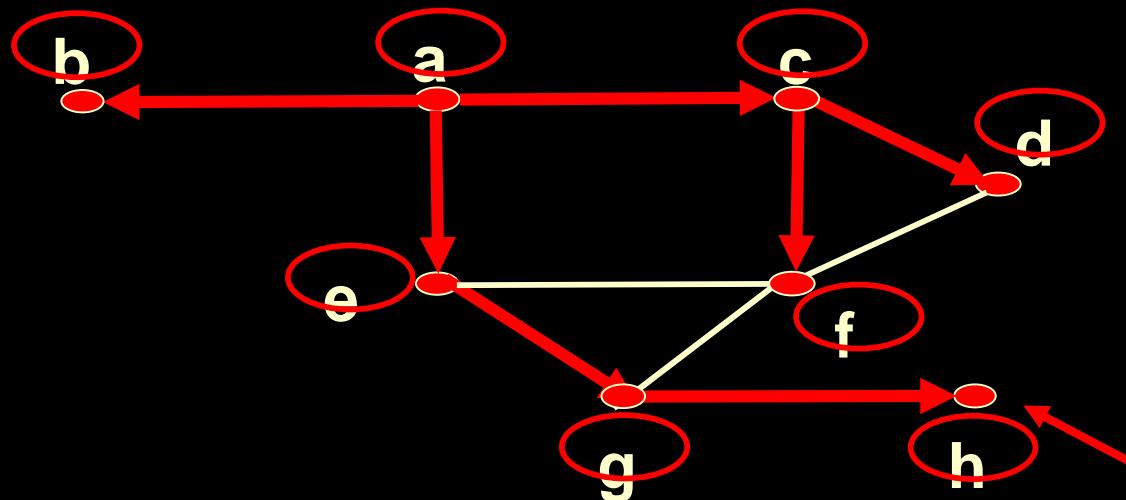


means vertex  $v$  has been processed

## Example execution of BFS



undirected  
graph  $G$



Queue  
=  $\langle \rangle$

current vertex

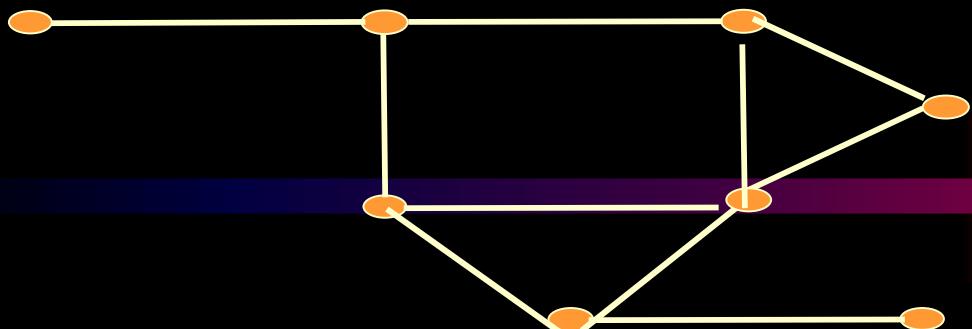
v

means vertex v has been processed

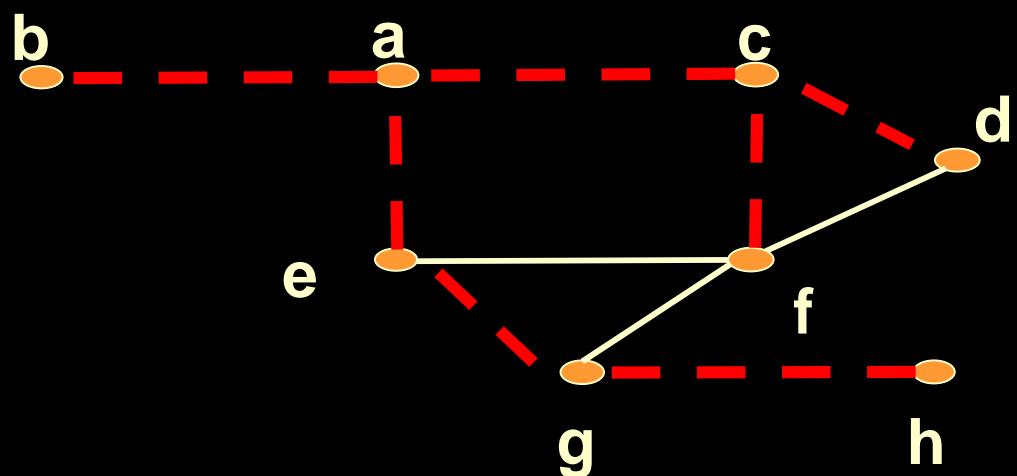
3/3/2022

Graphs

## Example execution of BFS

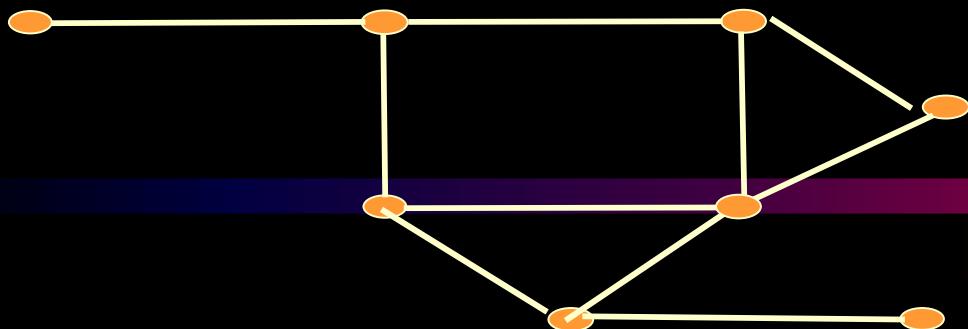


undirected  
graph  $G$

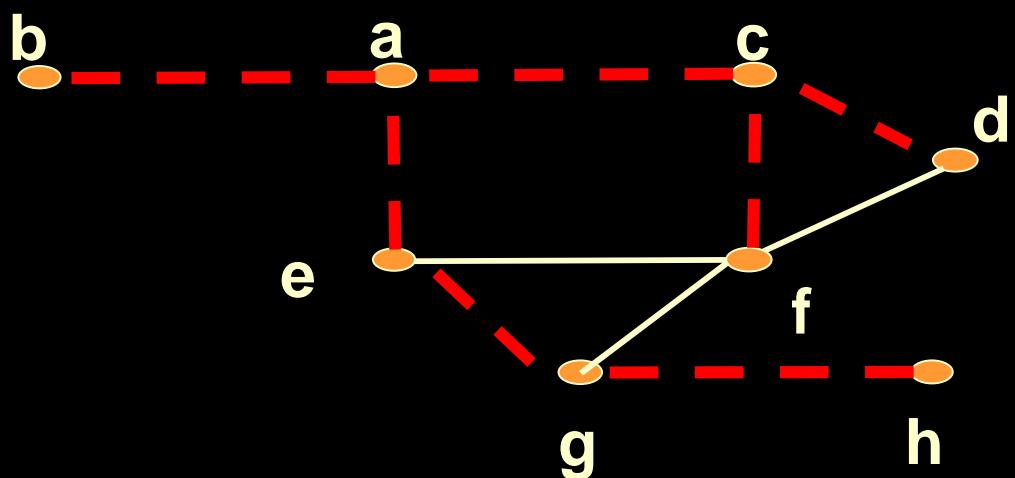


a breadth-first traversal of  $G$   
Graphs

## Example execution of BFS



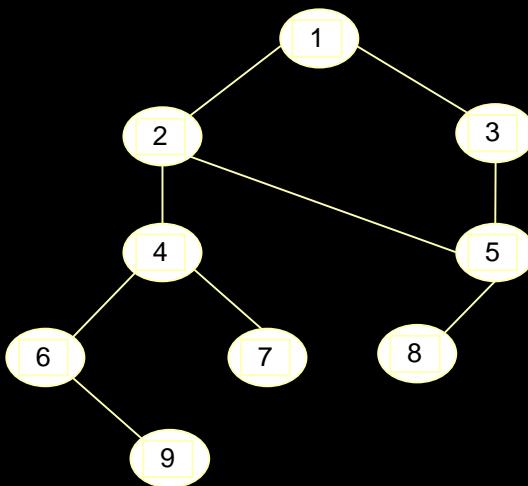
undirected  
graph  $G$



a breadth-first traversal of  $G$   
Graphs

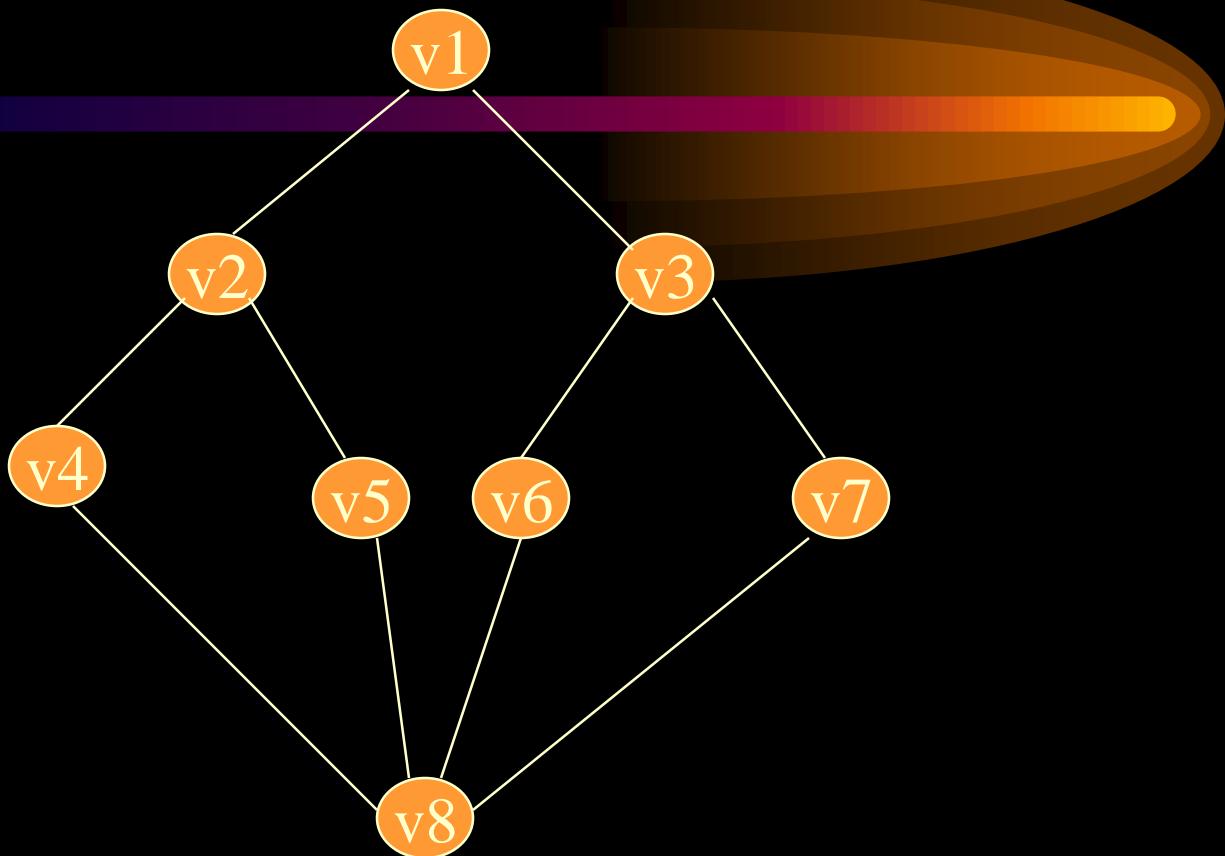
# *Breadth first search*

- Find out BFS for following graph.



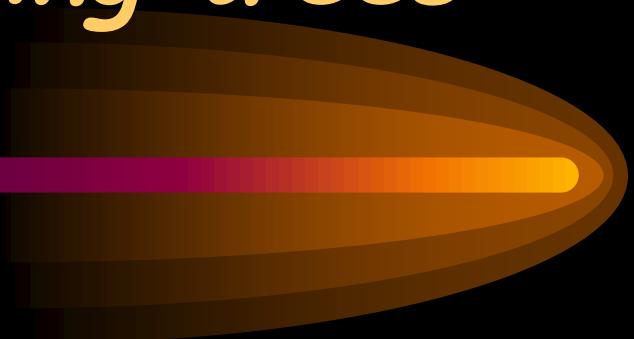
- 1 2 3 4 5 6 7 8 9

Show BFS for following graph:



# *Minimum cost spanning trees:*

- Prim's Algorithm
- Kruskal's Algorithm



## PRIM'S ALGORITHM

- Prim's algorithm finds a minimum cost spanning tree by selecting edges from the graph one-by-one as follows:
- It starts with a tree,  $T$ , consisting of the starting vertex,  $x$ .
- Then, it adds the shortest edge emanating from  $x$  that connects  $T$  to the rest of the graph.
- It then moves further and repeat the process.

Let  $T$  be a tree consisting of only the starting vertex  $x$

While ( $T$  has fewer than  $n$  vertices)

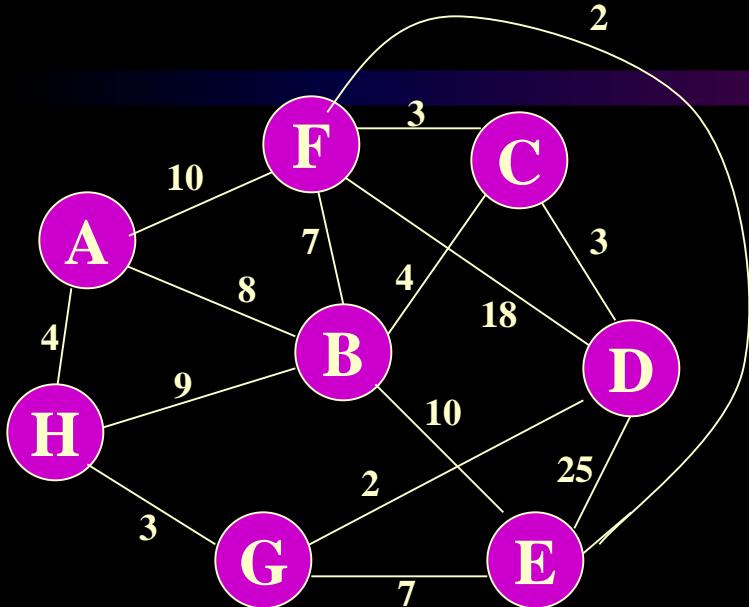
{

    find the smallest edge connecting  $T$  to  $G-T$

    add it to  $T$

}

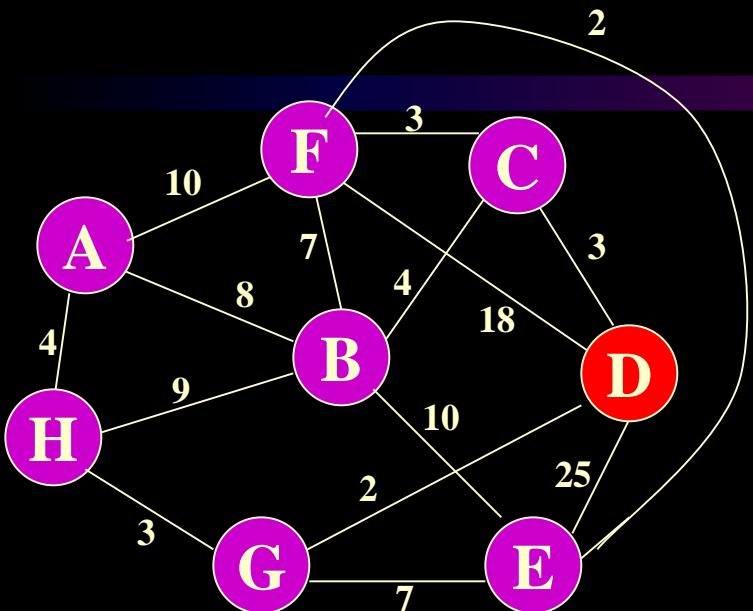
# Walk-Through



Initialize  
array

	$K$	$d_v$	$p_v$
A	False	$\infty$	-
B	False	$\infty$	-
C	False	$\infty$	-
D	False	$\infty$	-
E	False	$\infty$	-
F	False	$\infty$	-
G	False	$\infty$	-
H	False	$\infty$	-

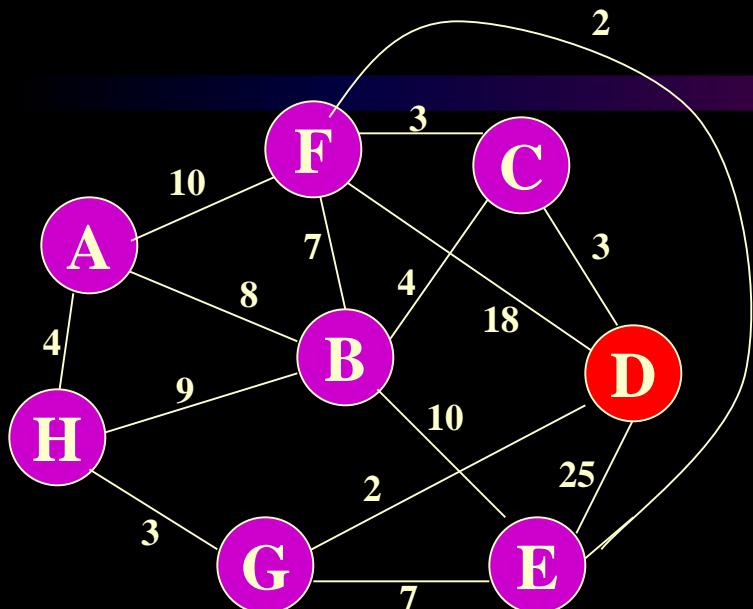
- $K$  (Vertex included in spanning tree)
- $d_v$  (Distance of Vertex)
- $p_v$  (Previous Vertex)



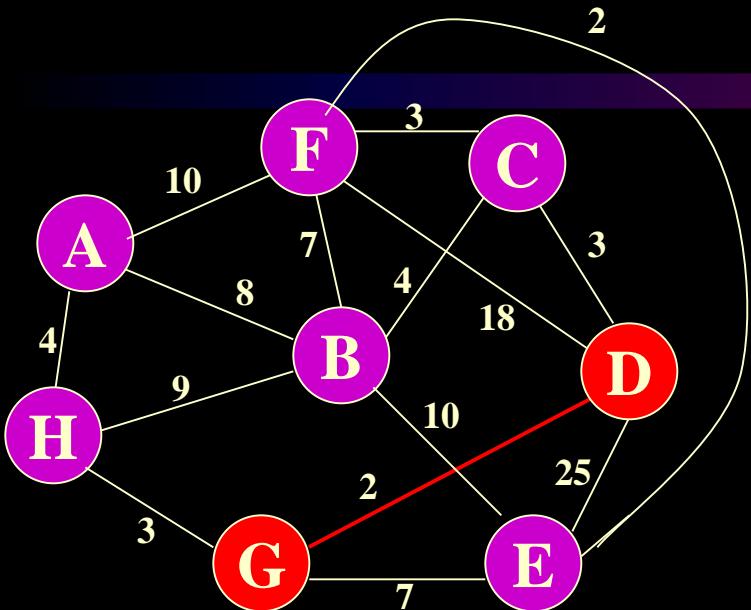
Start with any node,  
say D

	$K$	$d_v$	$p_v$
A			
B			
C			
D	T	0	-
E			
F			
G			
H			

Update distances of  
adjacent, unselected  
nodes



	$K$	$d_v$	$p_v$
A			
B			
C		3	D
D	T	0	-
E		25	D
F		18	D
G		2	D
H			

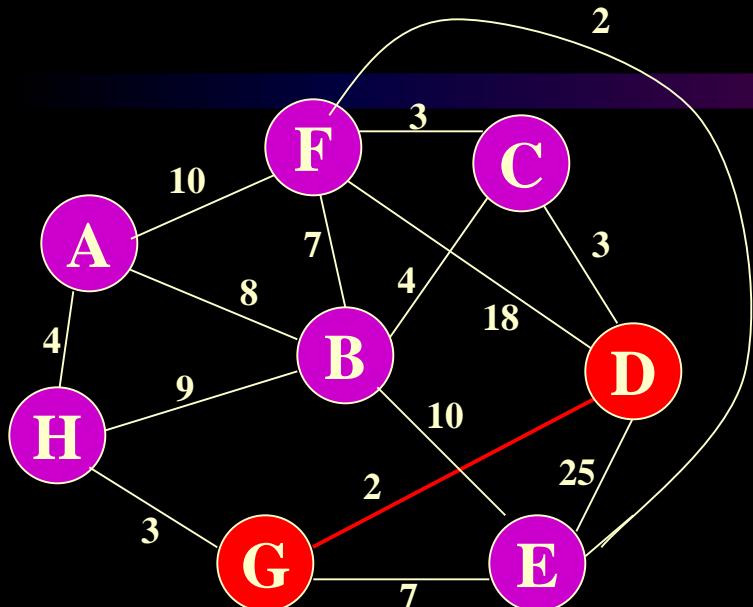


Select node with  
minimum distance

Color bar: Select node with minimum distance

	$K$	$d_v$	$p_v$
A			
B			
C		3	D
D	T	0	-
E		25	D
F		18	D
G	T	2	D
H			

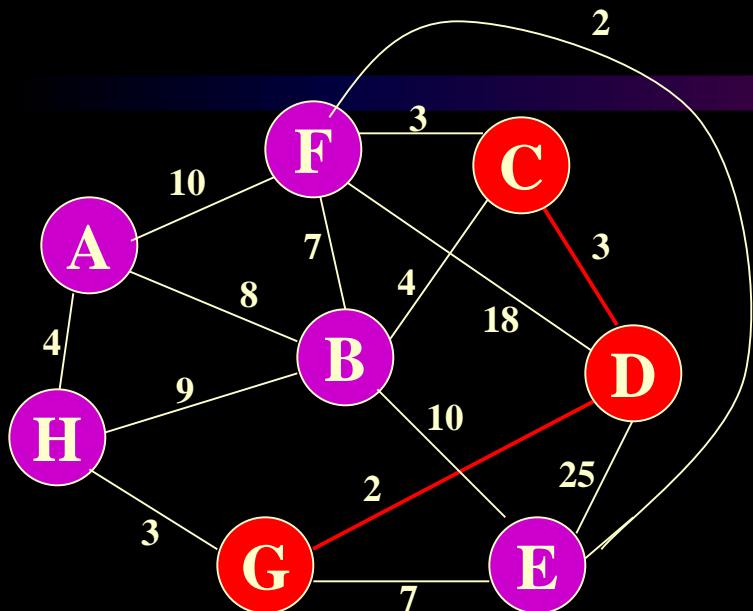
Update distances of  
adjacent, unselected  
nodes



	$K$	$d_v$	$p_v$
A			
B			
C		3	D
D	T	0	-
E		7	G
F		18	D
G	T	2	D
H		3	G

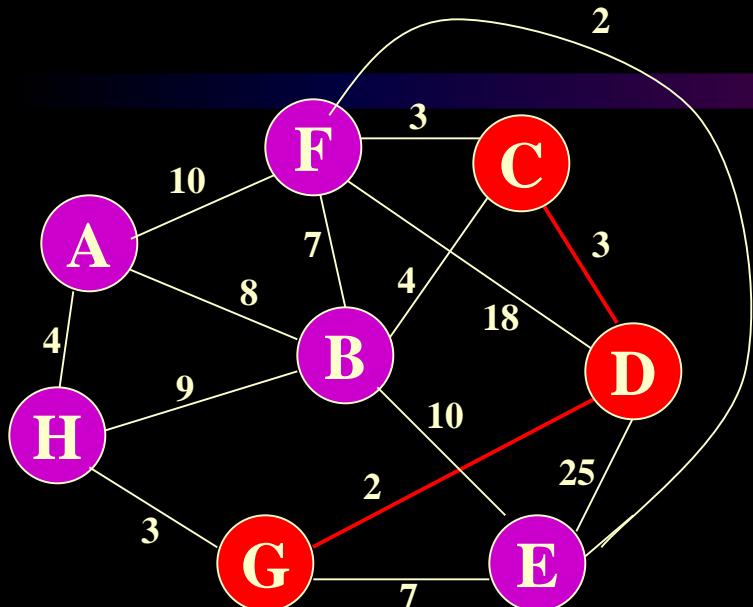
Prim's --- Selectin NEW VERTES ---till  
n vertices are obtained

Select node with  
minimum distance



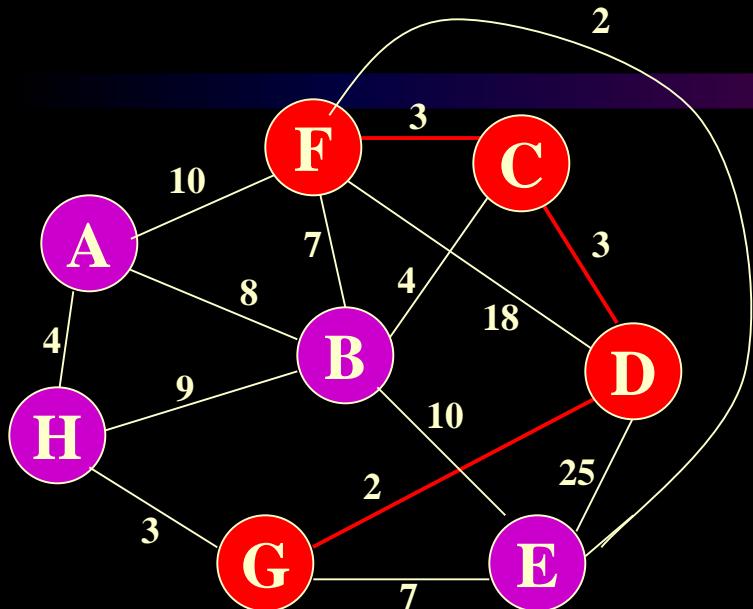
	$K$	$d_v$	$p_v$
A			
B		4	C
C	T	3	D
D	T	0	-
E		7	G
F		18	D
G	T	2	D
H		3	G

Update distances of  
adjacent, unselected  
nodes



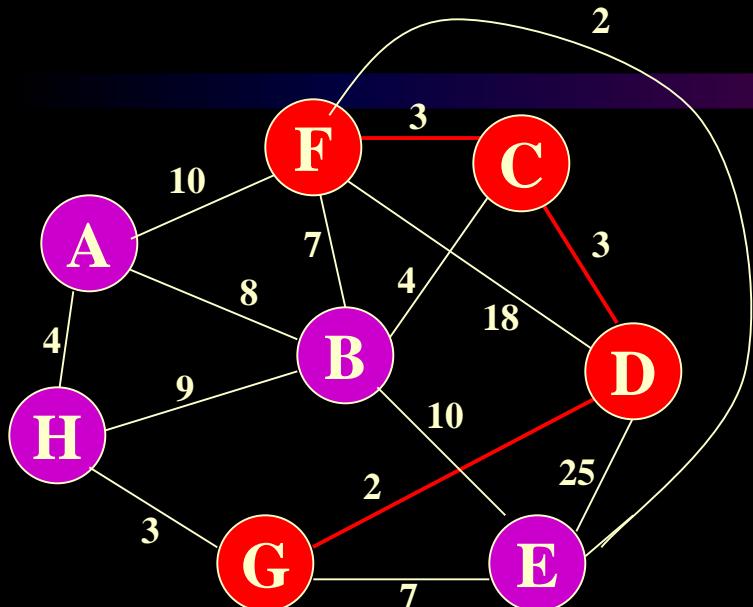
	$K$	$d_v$	$p_v$
A			
B		4	C
C	T	3	D
D	T	0	-
E		7	G
F		3	C
G	T	2	D
H		3	G

Select node with  
minimum distance



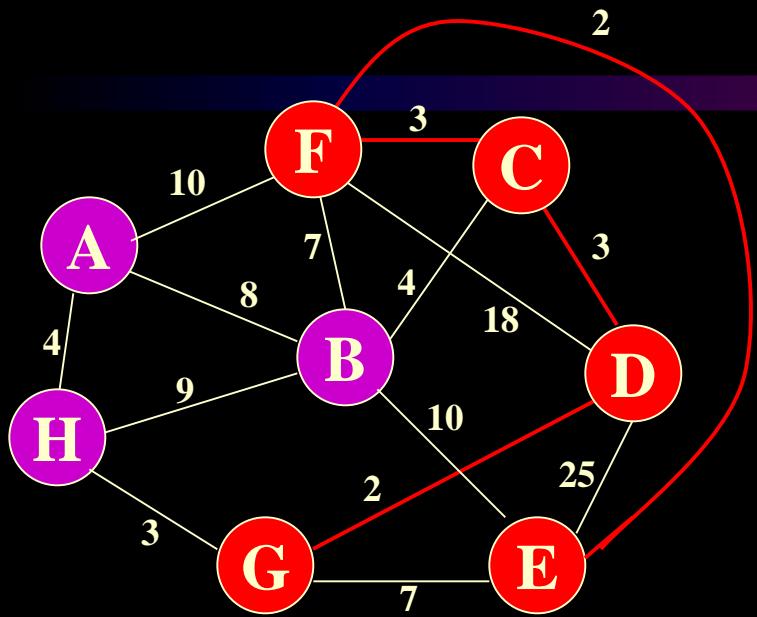
	$K$	$d_v$	$p_v$
A			
B		4	C
C	T	3	D
D	T	0	-
E		7	G
F	T	3	C
G	T	2	D
H		3	G

Update distances of  
adjacent, unselected  
nodes



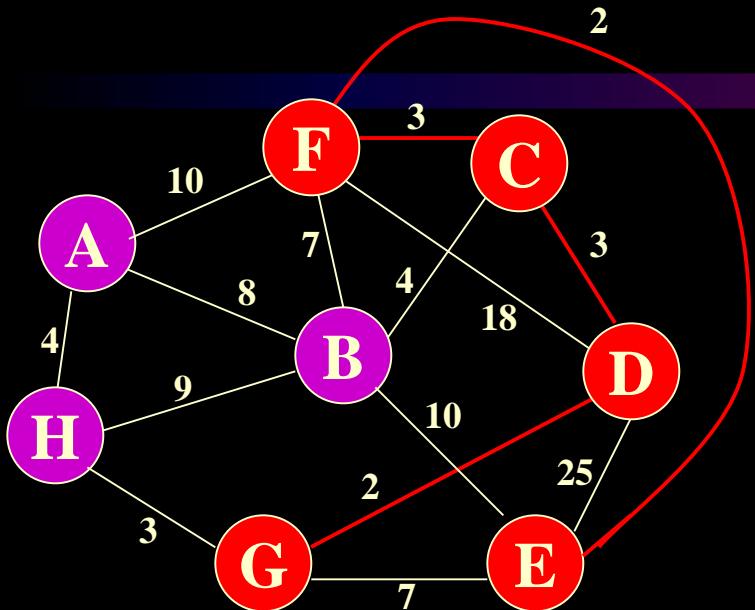
	$K$	$d_v$	$p_v$
A		10	F
B		4	C
C	T	3	D
D	T	0	-
E		2	F
F	T	3	C
G	T	2	D
H		3	G

Select node with  
minimum distance



	$K$	$d_v$	$p_v$
A		10	F
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G

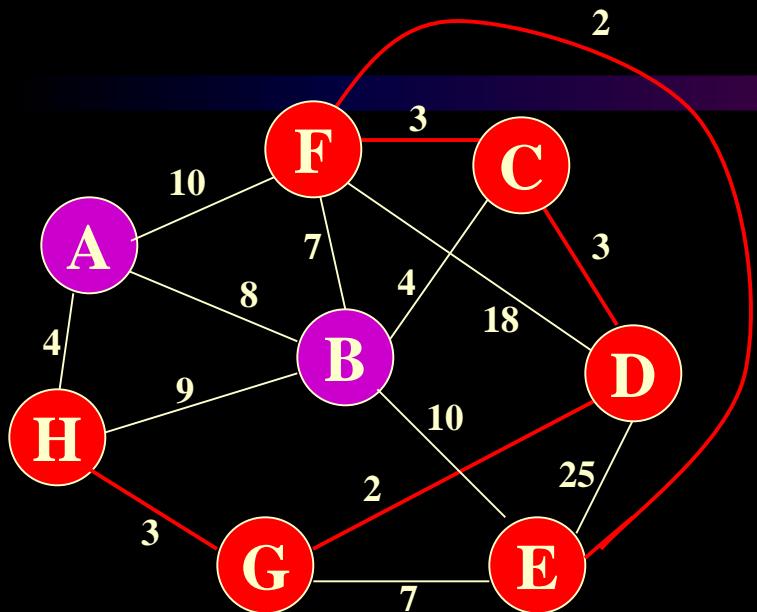
Update distances of  
adjacent, unselected  
nodes



	$K$	$d_v$	$p_v$
A		10	F
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H		3	G

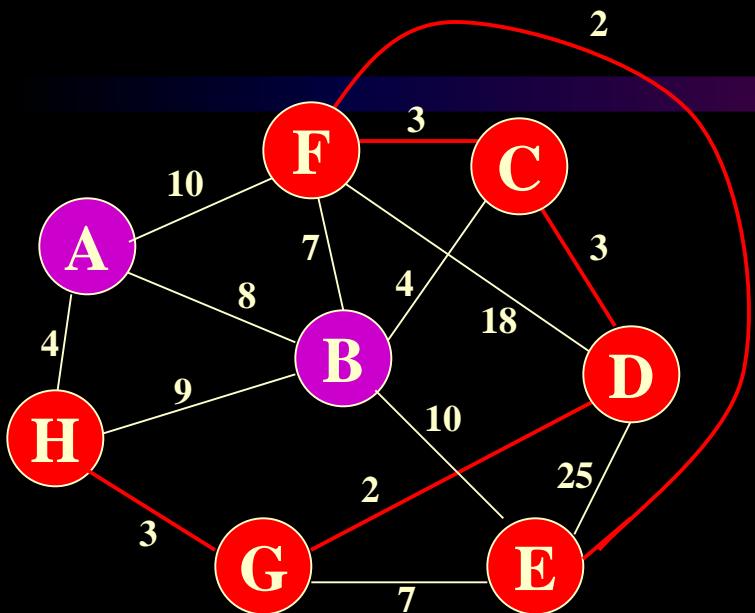
Table entries  
unchanged

Select node with  
minimum distance



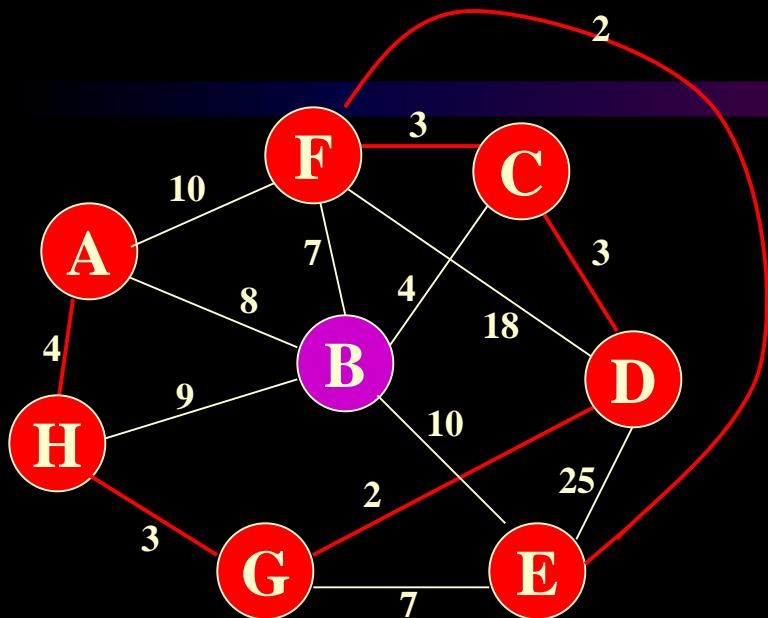
	$K$	$d_v$	$p_v$
A		10	F
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Update distances of  
adjacent, unselected  
nodes



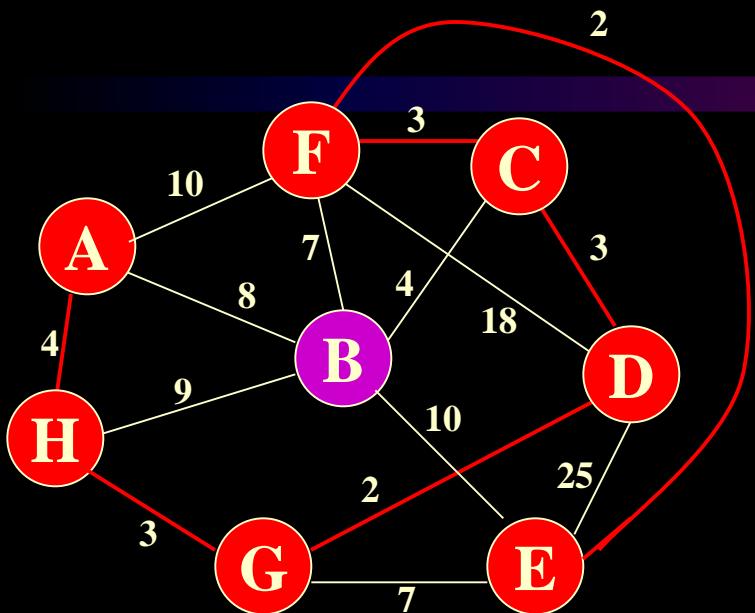
	$K$	$d_v$	$p_v$
A		4	H
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Select node with  
minimum distance



	$K$	$d_v$	$p_v$
A	T	4	H
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

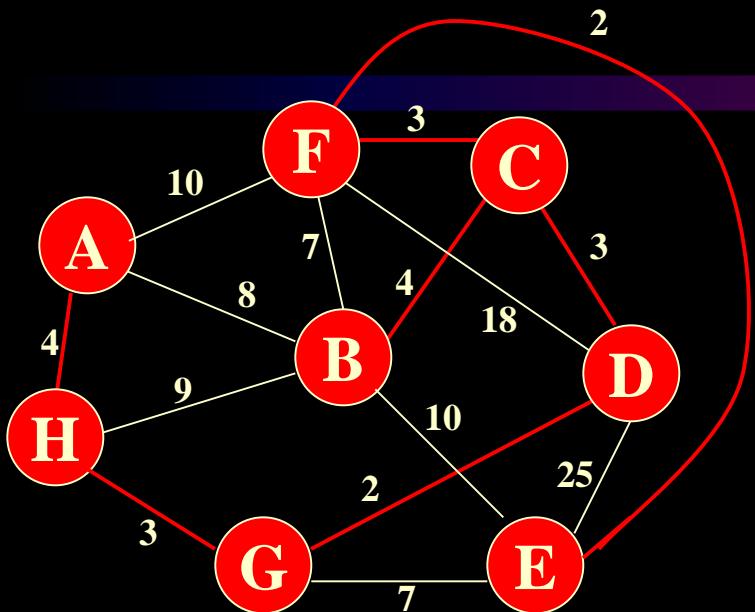
Update distances of  
adjacent, unselected  
nodes



	$K$	$d_v$	$p_v$
A	T	4	H
B		4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

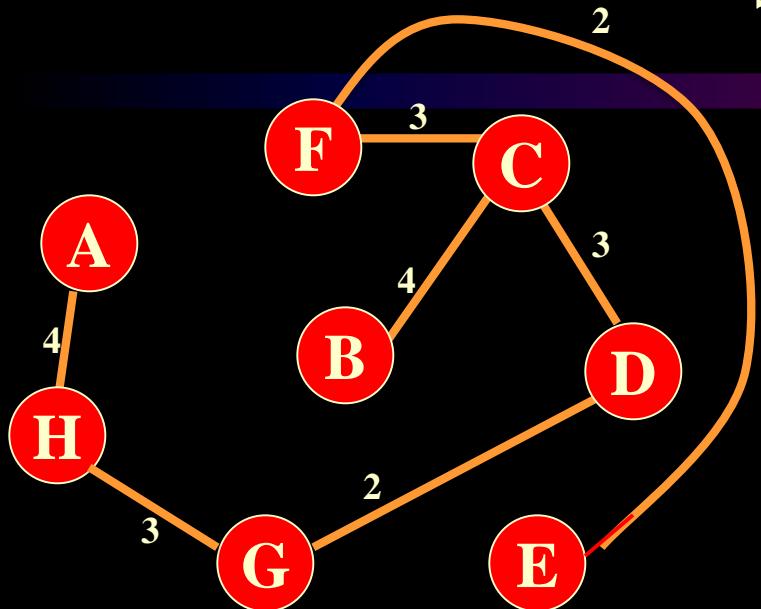
Table entries  
unchanged

Select node with  
minimum distance



	$K$	$d_v$	$p_v$
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Cost of Minimum  
Spanning Tree =  $\sum d_v =$   
**21**

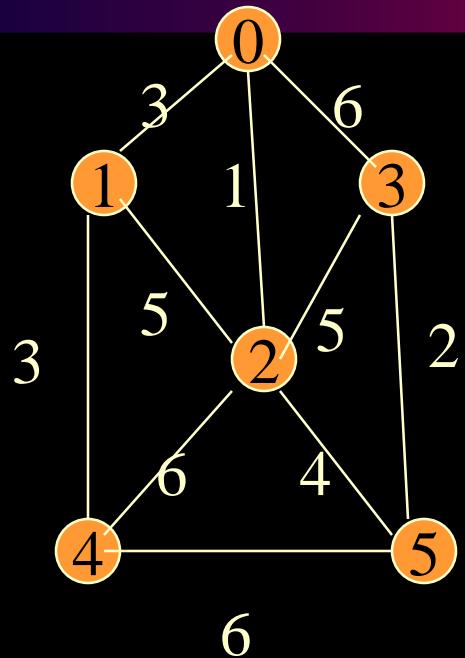


	$K$	$d_v$	$p_v$
A	T	4	H
B	T	4	C
C	T	3	D
D	T	0	-
E	T	2	F
F	T	3	C
G	T	2	D
H	T	3	G

Done

Visited Order: D,G,C,F,E,H,A,B

# Solve using prim's algo.

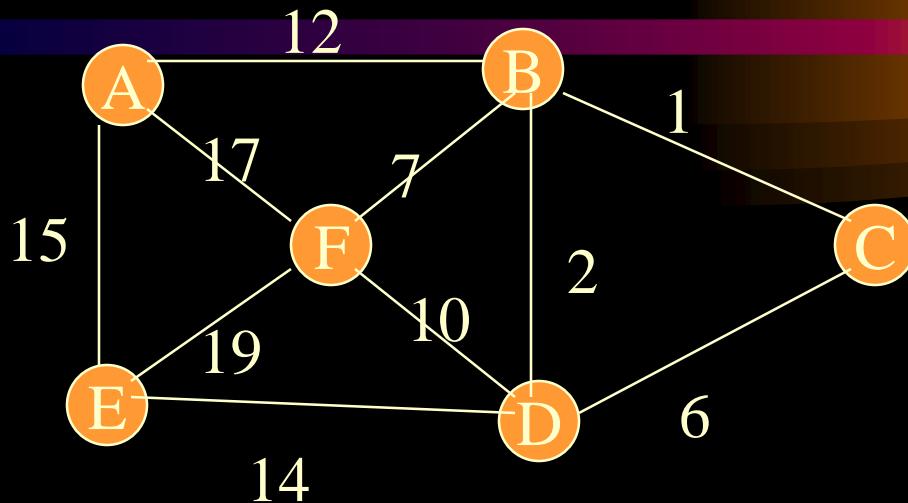


	$K$	$d_v$	$p_v$
0	T	0	
1	T	3	0
2	T	1	0
3	T	2	5
4	T	3	1
5	T	4	2

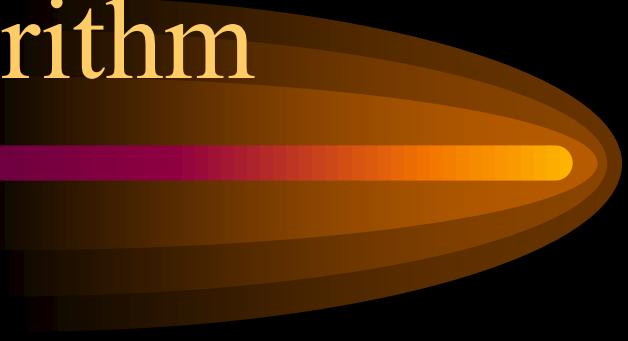
Solution using Prims

Visited Order: 0,2,1,4,5,2

# *Solve using prim's algo.*



# Kruskal's Algorithm



- Work with edges, rather than nodes
- Two steps:
  - Sort edges by increasing edge weight
  - Select the first  $|V| - 1$  edges that do not generate a cycle

## KRUSKAL'S ALGORITHM.

- Kruskal's algorithm also finds the minimum cost spanning tree of a graph by adding edges one-by-one.
- However, Kruskal's algorithm forms a forest of trees, which are joined together incrementally to form the minimum cost spanning tree.

# *The Kruskal Algorithm*

```
// input: a graph G with n nodes and m edges  
// output: E: a MST for G
```

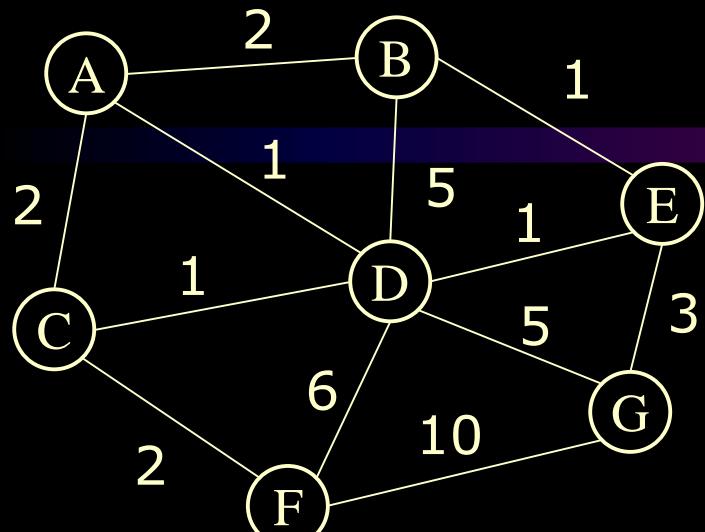
1.  $EG[1..m] \leftarrow$  Sort the m edges in G in increasing weight order
2.  $E \leftarrow \{ \}$  //the edges in the MST
3.  $i \leftarrow 1$  //counter for EG
4. **While**  $|E| < n - 1$  **do**  
    **if** adding  $EG[i]$  to E does not add a cycle **then**  
         $E \leftarrow E \cup \{EG[i]\}$   
         $i \leftarrow i + 1$
5. **return** E

Is this algorithm Greedy?

Yes

Complexity:  $O(|E|\log_2 |E|)$

# Example: Kruskal's Algorithm



Edges in sorted order:

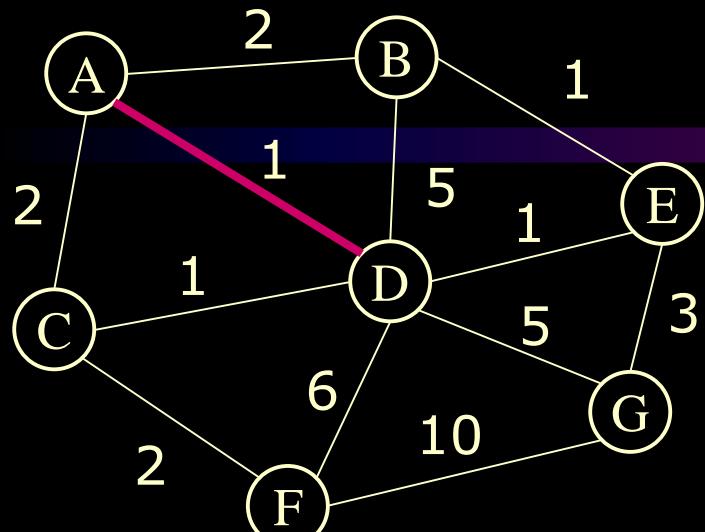
- 1: (A,D) (C,D) (B,E) (D,E)
- 2: (A,B) (C,F) (A,C)
- 3: (E,G)
- 5: (D,G) (B,D)
- 6: (D,F)
- 10: (F,G)

Sets: (A) (B) (C) (D) (E) (F) (G)

Output:

At each step, the union/find sets are the trees in the forest

# Example: Kruskal's Algorithm



Edges in sorted order:

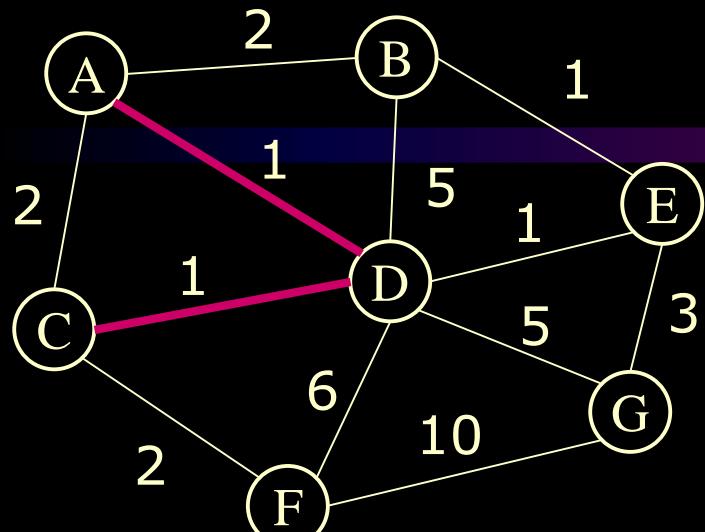
- 1: ~~(A,D)~~ (C,D) (B,E) (D,E)
- 2: (A,B) (C,F) (A,C)
- 3: (E,G)
- 5: (D,G) (B,D)
- 6: (D,F)
- 10: (F,G)

Sets: (A,D) (B) (C) (E) (F) (G)

Output: (A,D)

At each step, the union/find sets are the trees in the forest

# Example: Kruskal's Algorithm



Edges in sorted order:

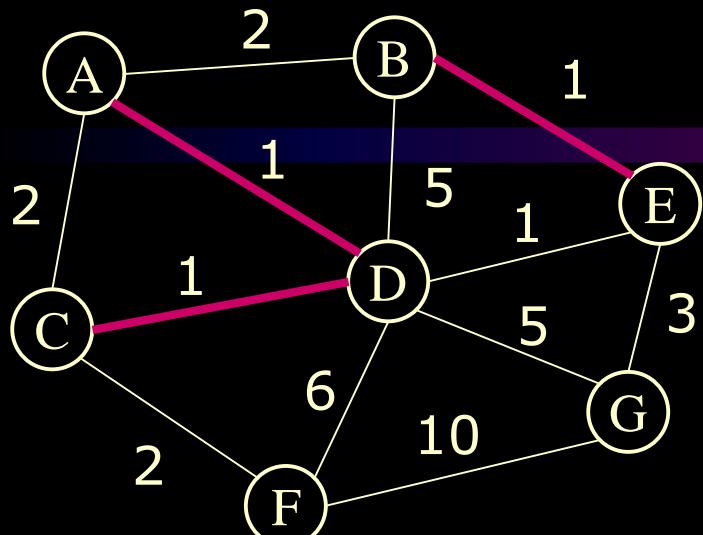
- 1: ~~(A,D)~~ ~~(C,D)~~ (B,E) (D,E)
- 2: (A,B) (C,F) (A,C)
- 3: (E,G)
- 5: (D,G) (B,D)
- 6: (D,F)
- 10: (F,G)

Sets: (A, C, D) (B) (E) (F) (G)

Output: (A, D) (C, D)

At each step, the union/find sets are the trees in the forest

# Example: Kruskal's Algorithm



Edges in sorted order:

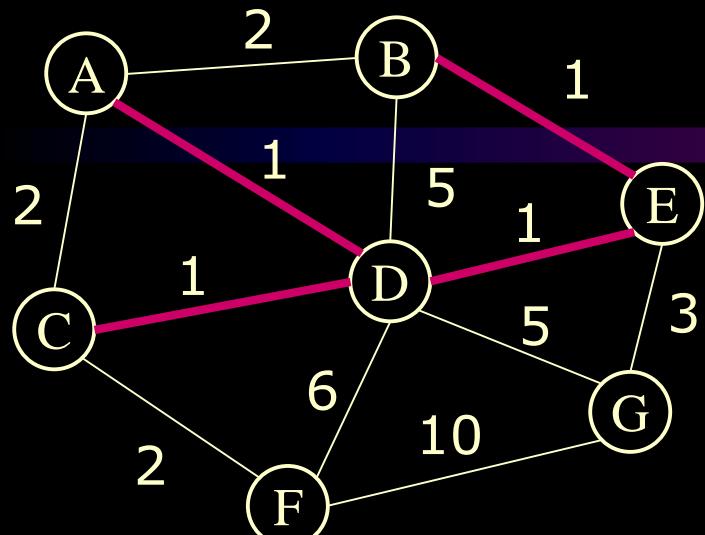
- 1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ (D,E)
- 2: (A,B) (C,F) (A,C)
- 3: (E,G)
- 5: (D,G) (B,D)
- 6: (D,F)
- 10: (F,G)

Sets: (A, C, D) (B, E) (F) (G)

Output: (A, D) (C, D) (B, E)

At each step, the union/find sets are the trees in the forest

# Example: Kruskal's Algorithm



Edges in sorted order:

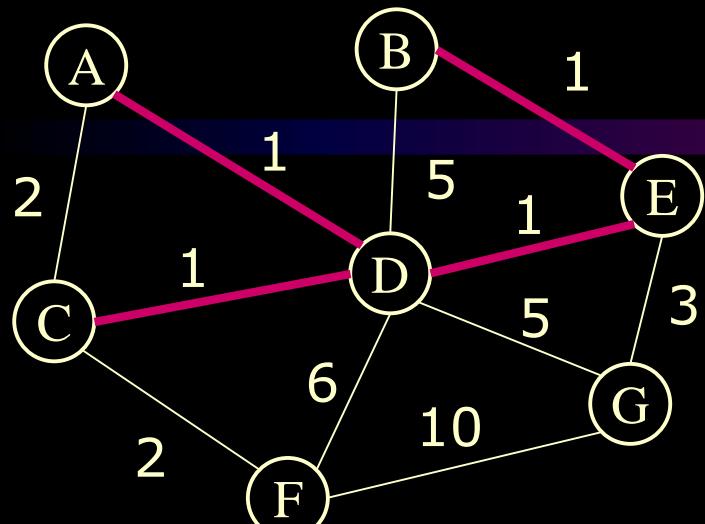
- 1: ~~(A,D) (C,D) (B,E) (D,E)~~
- 2: (A,B) (C,F) (A,C)
- 3: (E,G)
- 5: (D,G) (B,D)
- 6: (D,F)
- 10: (F,G)

Sets:  $(A, B, C, D, E) (F) (G)$

Output:  $(A, D) (C, D) (B, E) (D, E)$

At each step, the union/find sets are the trees in the forest

# Example: Kruskal's Algorithm



Edges in sorted order:

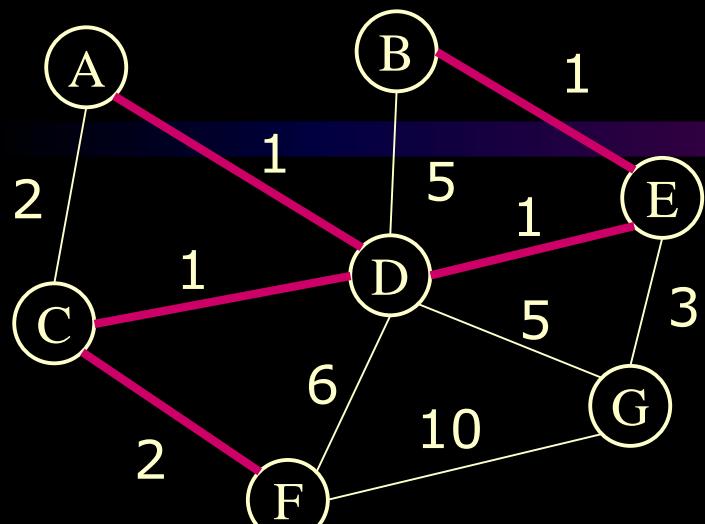
- 1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~
- 2: ~~(A,B)~~ (C,F) (A,C)
- 3: (E,G)
- 5: (D,G) (B,D)
- 6: (D,F)
- 10: (F,G)

Sets:  $(A, B, C, D, E) (F) (G)$

Output:  $(A,D) (C,D) (B,E) (D,E)$

At each step, the union/find sets are the trees in the forest

# Example: Kruskal's Algorithm



Edges in sorted order:

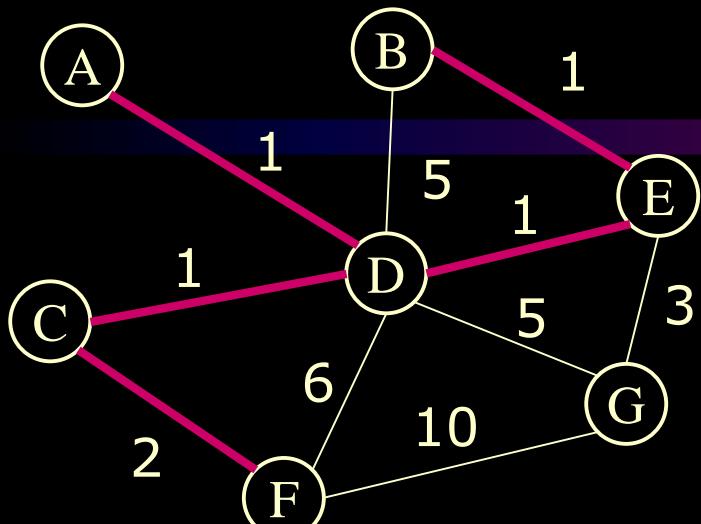
- 1: ~~(A,D) (C,D) (B,E) (D,E)~~
- 2: ~~(A,B) (C,F) (A,C)~~
- 3: (E,G)
- 5: (D,G) (B,D)
- 6: (D,F)
- 10: (F,G)

Sets:  $(A, B, C, D, E, F) (G)$

Output:  $(A,D) (C,D) (B,E) (D,E) (C,F)$

At each step, the union/find sets are the trees in the forest

# Example: Kruskal's Algorithm



Edges in sorted order:

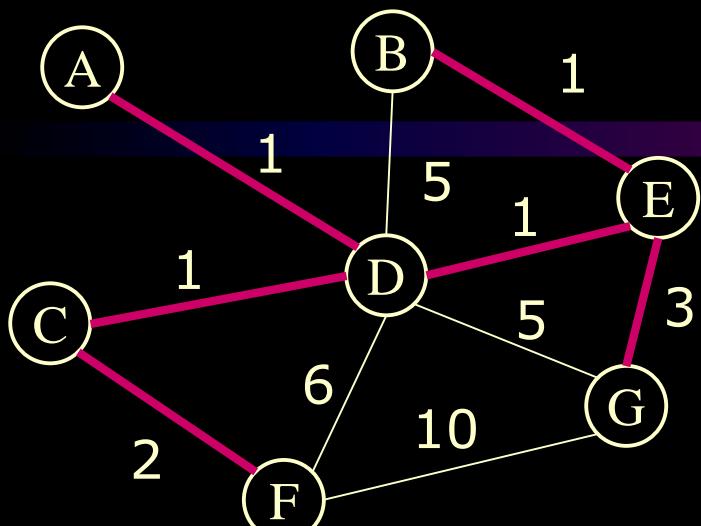
- 1: ~~(A,D) (C,D) (B,E) (D,E)~~
- 2: ~~(A,B) (C,F) (A,C)~~
- 3: (E,G)
- 5: (D,G) (B,D)
- 6: (D,F)
- 10: (F,G)

Sets:  $(A, B, C, D, E, F) (G)$

Output: (A,D) (C,D) (B,E) (D,E) (C,F)

At each step, the union/find sets are the trees in the forest

# Example: Kruskal's Algorithm



Edges in sorted order:

- 1: ~~(A,D) (C,D) (B,E) (D,E)~~
- 2: ~~(A,B) (C,F) (A,C)~~
- 3: ~~(E,G)~~
- 5: (D,G) (B,D)
- 6: (D,F)
- 10: (F,G)

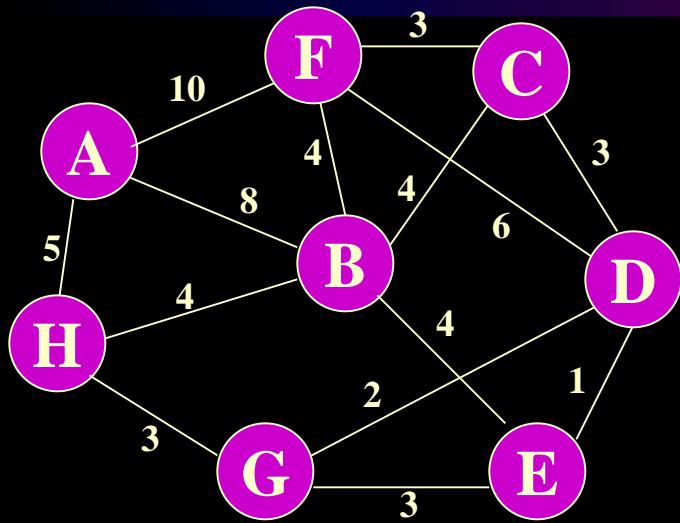
Sets:  $(A, B, C, D, E, F, G)$

Output:  $(A, D) (C, D) (B, E) (D, E) (C, F) (E, G)$

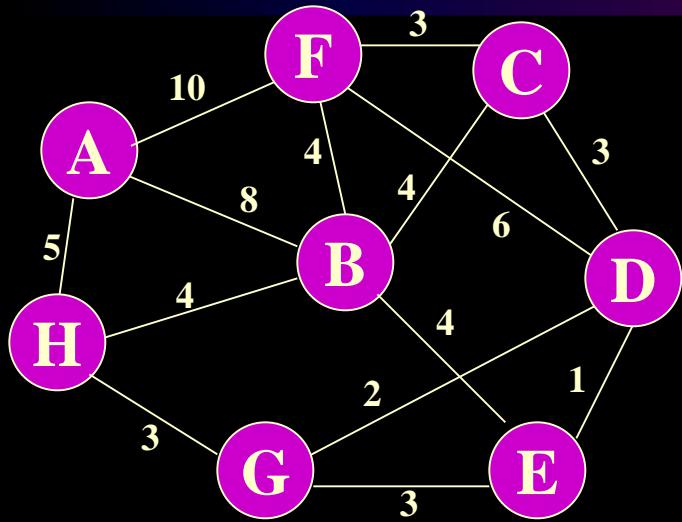
At each step, the union/find sets are the trees in the forest

# Walk-Through

Consider an undirected, weight graph

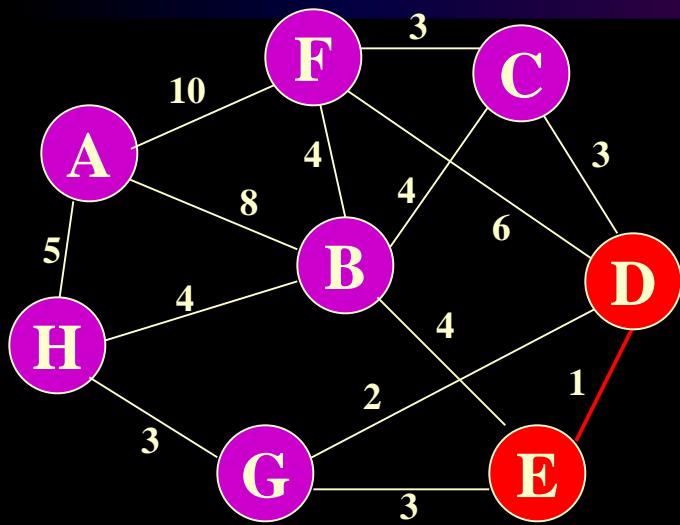


Sort the edges by increasing edge weight



$edge$	$d_v$		$edge$	$d_v$	
(D,E)	1		(B,E)	4	
(D,G)	2		(B,F)	4	
(E,G)	3		(B,H)	4	
(C,D)	3		(A,H)	5	
(G,H)	3		(D,F)	6	
(C,F)	3		(A,B)	8	
(B,C)	4		(A,F)	10	

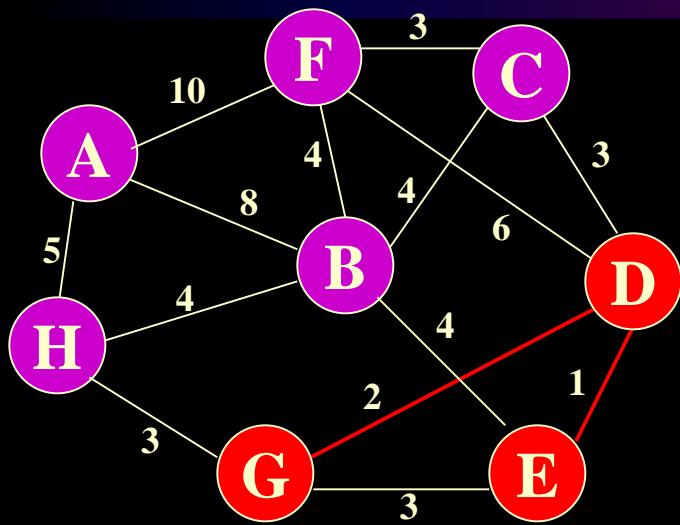
Select first  $|V|-1$  edges which  
do not generate a cycle



<i>edge</i>	$d_v$	
(D,E)	1	✓
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

<i>edge</i>	$d_v$	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

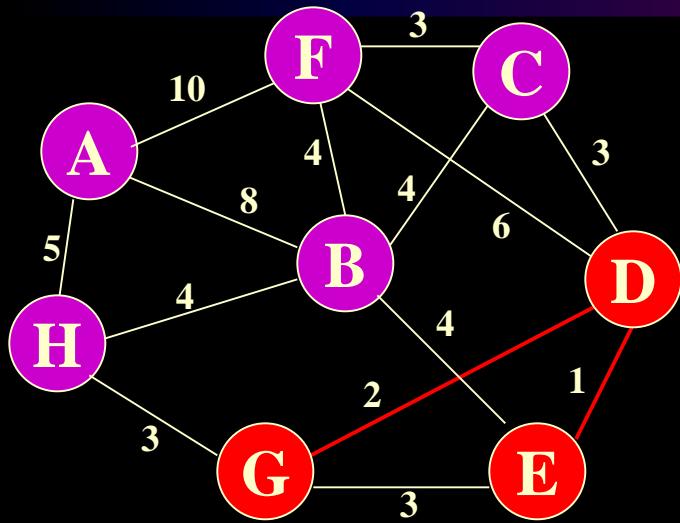
Select first  $|V|-1$  edges which  
do not generate a cycle



<i>edge</i>	$d_v$	
(D,E)	1	✓
(D,G)	2	✓
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

<i>edge</i>	$d_v$	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Select first  $|V|-1$  edges which  
do not generate a cycle

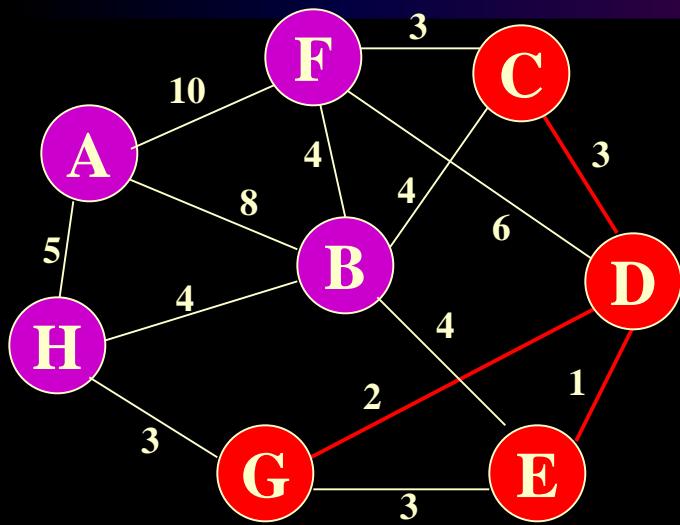


<i>edge</i>	$d_v$	
(D,E)	1	✓
(D,G)	2	✓
(E,G)	3	✗
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

<i>edge</i>	$d_v$	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Accepting edge (E,G) would create  
a cycle

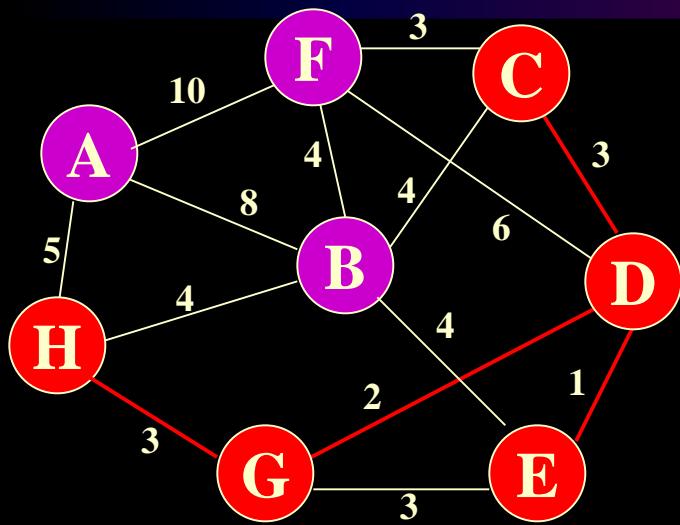
Select first  $|V|-1$  edges which  
do not generate a cycle



<i>edge</i>	$d_v$	
(D,E)	1	✓
(D,G)	2	✓
(E,G)	3	✗
(C,D)	3	✓
(G,H)	3	
(C,F)	3	
(B,C)	4	

<i>edge</i>	$d_v$	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

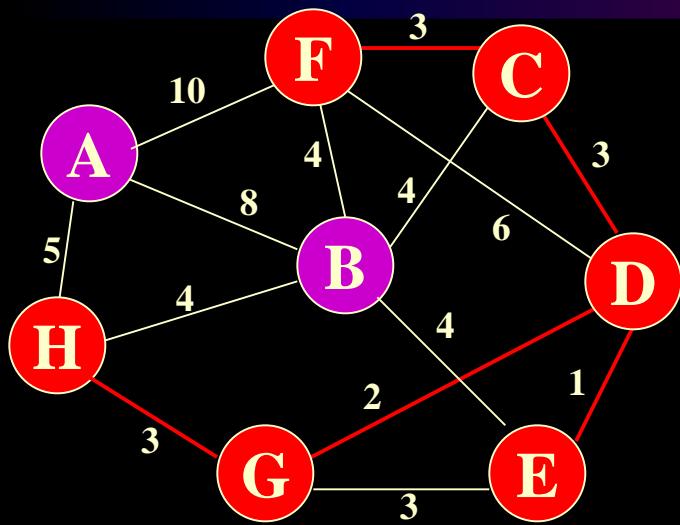
Select first  $|V|-1$  edges which  
do not generate a cycle



<i>edge</i>	$d_v$	
(D,E)	1	✓
(D,G)	2	✓
(E,G)	3	✗
(C,D)	3	✓
(G,H)	3	✓
(C,F)	3	
(B,C)	4	

<i>edge</i>	$d_v$	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

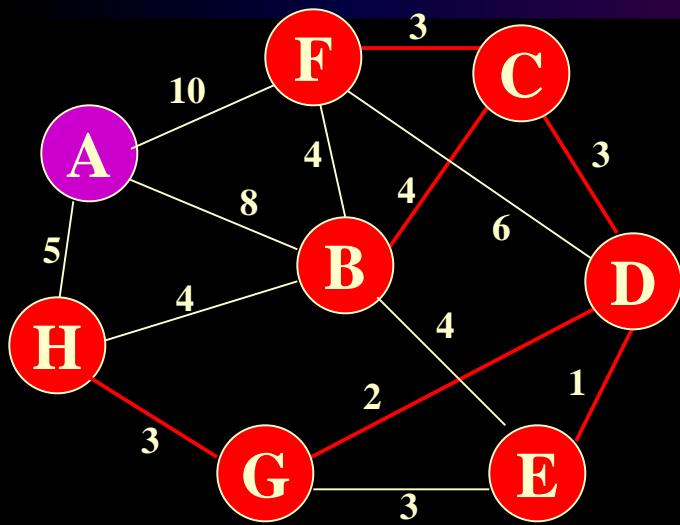
Select first  $|V|-1$  edges which  
do not generate a cycle



<i>edge</i>	$d_v$	
(D,E)	1	✓
(D,G)	2	✓
(E,G)	3	✗
(C,D)	3	✓
(G,H)	3	✓
(C,F)	3	✓
(B,C)	4	

<i>edge</i>	$d_v$	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

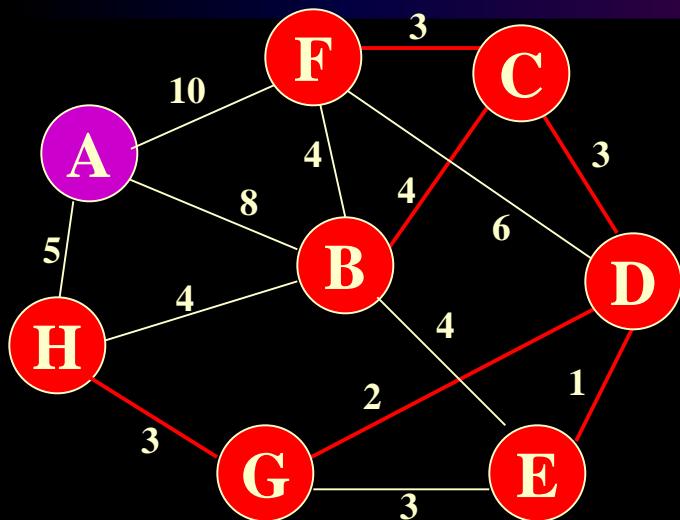
Select first  $|V|-1$  edges which  
do not generate a cycle



<i>edge</i>	$d_v$	
(D,E)	1	✓
(D,G)	2	✓
(E,G)	3	✗
(C,D)	3	✓
(G,H)	3	✓
(C,F)	3	✓
(B,C)	4	✓

<i>edge</i>	$d_v$	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

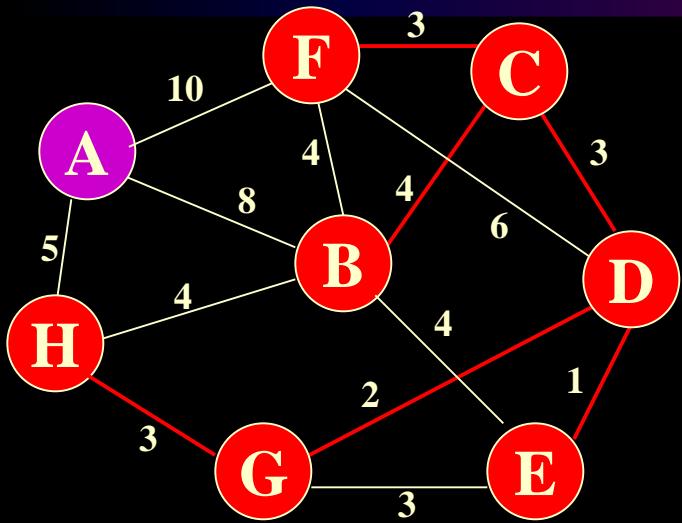
Select first  $|V|-1$  edges which  
do not generate a cycle



<i>edge</i>	$d_v$	
(D,E)	1	✓
(D,G)	2	✓
(E,G)	3	✗
(C,D)	3	✓
(G,H)	3	✓
(C,F)	3	✓
(B,C)	4	✓

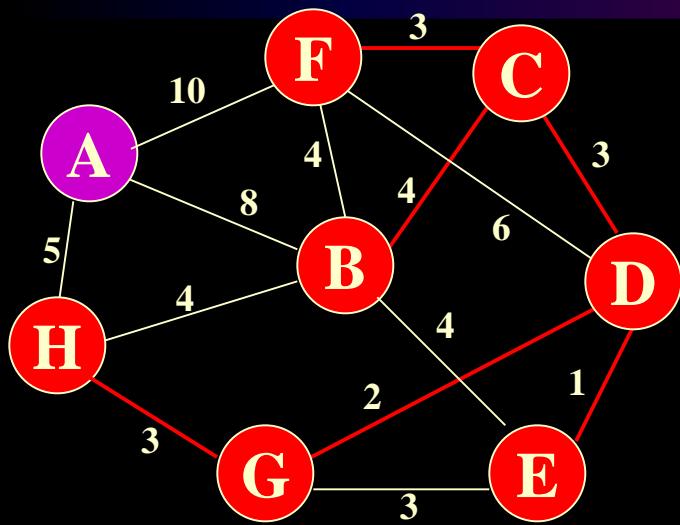
<i>edge</i>	$d_v$	
(B,E)	4	✗
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Select first  $|V|-1$  edges which  
do not generate a cycle



$edge$	$d_v$		$edge$	$d_v$	
(D,E)	1	✓	(B,E)	4	✗
(D,G)	2	✓	(B,F)	4	✗
(E,G)	3	✗	(B,H)	4	
(C,D)	3	✓	(A,H)	5	
(G,H)	3	✓	(D,F)	6	
(C,F)	3	✓	(A,B)	8	
(B,C)	4	✓	(A,F)	10	

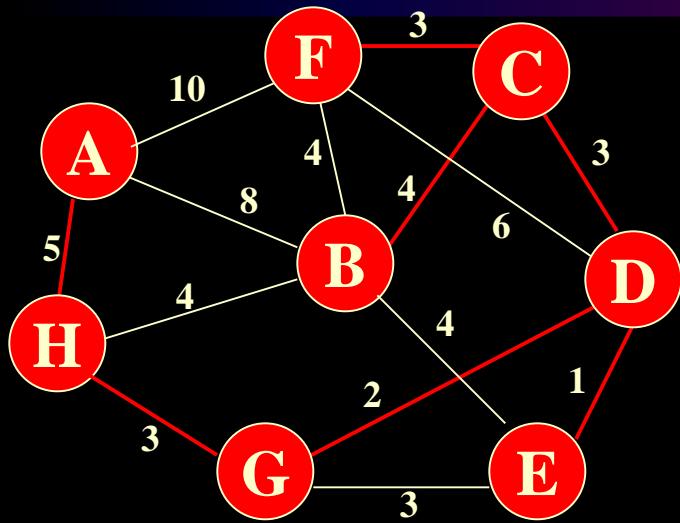
Select first  $|V|-1$  edges which  
do not generate a cycle



<i>edge</i>	$d_v$	
(D,E)	1	✓
(D,G)	2	✓
(E,G)	3	✗
(C,D)	3	✓
(G,H)	3	✓
(C,F)	3	✓
(B,C)	4	✓

<i>edge</i>	$d_v$	
(B,E)	4	✗
(B,F)	4	✗
(B,H)	4	✗
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

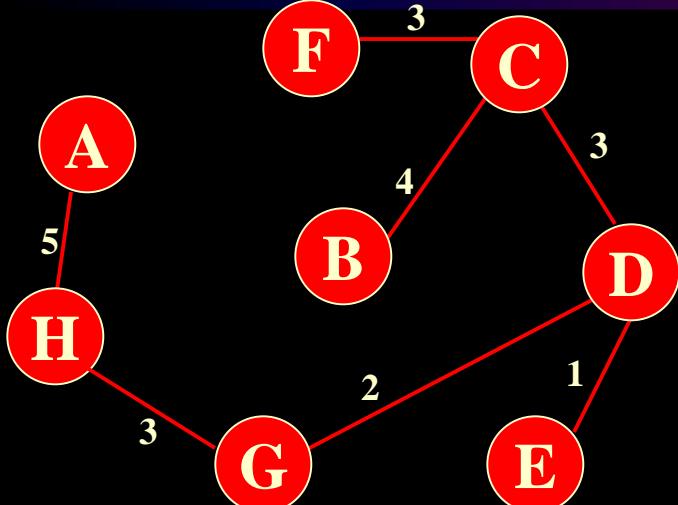
Select first  $|V|-1$  edges which  
do not generate a cycle



<i>edge</i>	$d_v$	
(D,E)	1	✓
(D,G)	2	✓
(E,G)	3	✗
(C,D)	3	✓
(G,H)	3	✓
(C,F)	3	✓
(B,C)	4	✓

<i>edge</i>	$d_v$	
(B,E)	4	✗
(B,F)	4	✗
(B,H)	4	✗
(A,H)	5	✓
(D,F)	6	
(A,B)	8	
(A,F)	10	

Select first  $|V|-1$  edges which  
do not generate a cycle



<i>edge</i>	$d_v$		<i>edge</i>	$d_v$	
(D,E)	1	✓	(B,E)	4	✗
(D,G)	2	✓	(B,F)	4	✗
(E,G)	3	✗	(B,H)	4	✗
(C,D)	3	✓	(A,H)	5	✓
(G,H)	3	✓	(D,F)	6	
(C,F)	3	✓	(A,B)	8	
(B,C)	4	✓	(A,F)	10	

}

Done

not  
considered

Total Cost =  $\sum d_v =$   
**21**

- **Kruskal's algorithm**

1. Select the shortest edge in a network
2. Select the next shortest edge which does not create a cycle
3. Repeat step 2 until all vertices have been connected

- **Prim's algorithm**

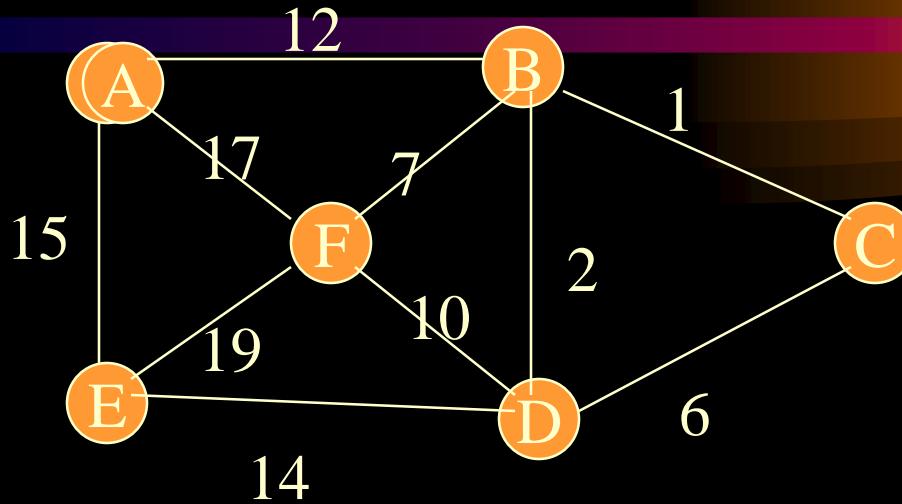
1. Select any vertex
2. Select the shortest edge connected to that vertex
3. Select the shortest edge connected to any vertex already connected
4. Repeat step 3 until all vertices have been connected

- Some points to note

- Both algorithms will always give solutions with the same length.
- They will usually select edges in a different order
- Occasionally they will use different edges – this may happen when you have to choose between edges with the same length. In this case there is more than one minimum connector for the network.

# *Construct minimum spanning tree using kruskal's algorithm:*

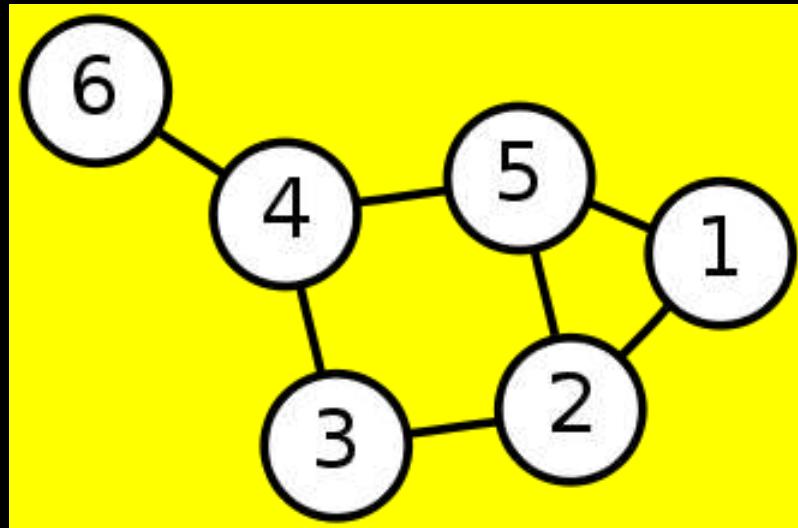
- 



# *Dijkstra's Algorithm*

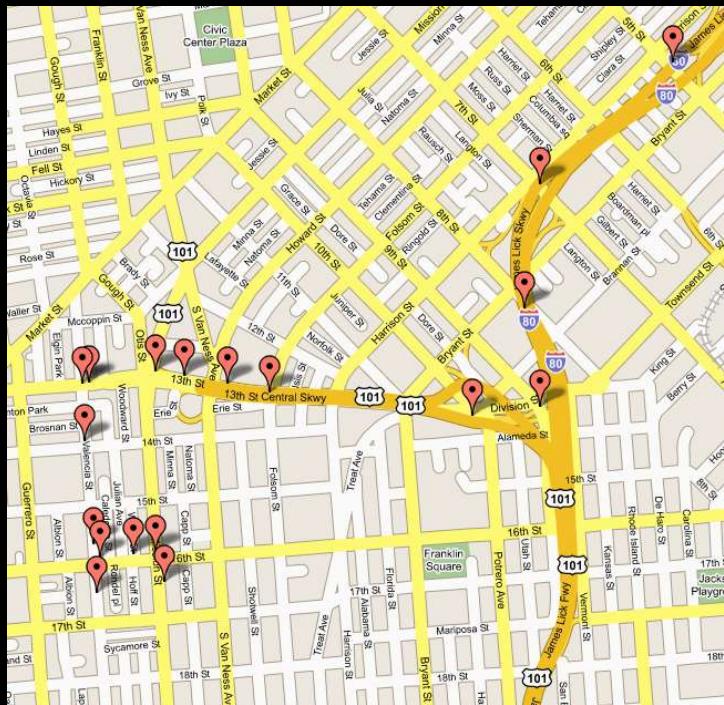
# *Single-Source Shortest Path*

**Single-Source Shortest Path Problem** - The problem of finding shortest paths from a source vertex  $v$  to all other vertices in the graph.



# Applications

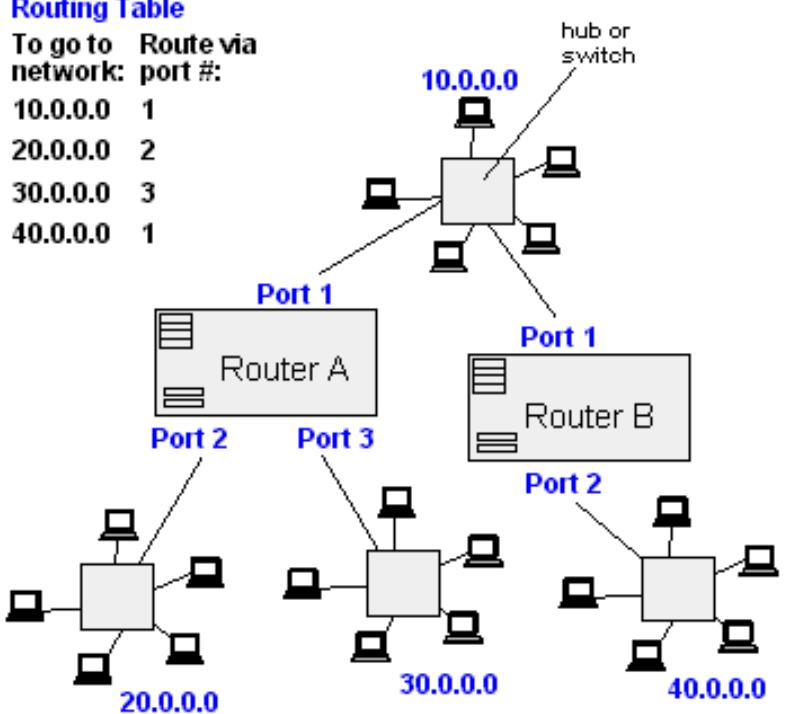
- Maps (Google Maps)
- Routing Systems



From Computer Desktop Encyclopedia  
© 1998 The Computer Language Co. Inc.

## Router A Routing Table

To go to network:	Route via port #:
10.0.0.0	1
20.0.0.0	2
30.0.0.0	3
40.0.0.0	1



# *Dijkstra's algorithm*

**Dijkstra's algorithm** - is a solution to the **single-source shortest path problem** in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Input: Weighted graph  $G=\{E,V\}$  and source vertex  $v \in V$ , such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex  $v \in V$  to all other vertices

# Approach

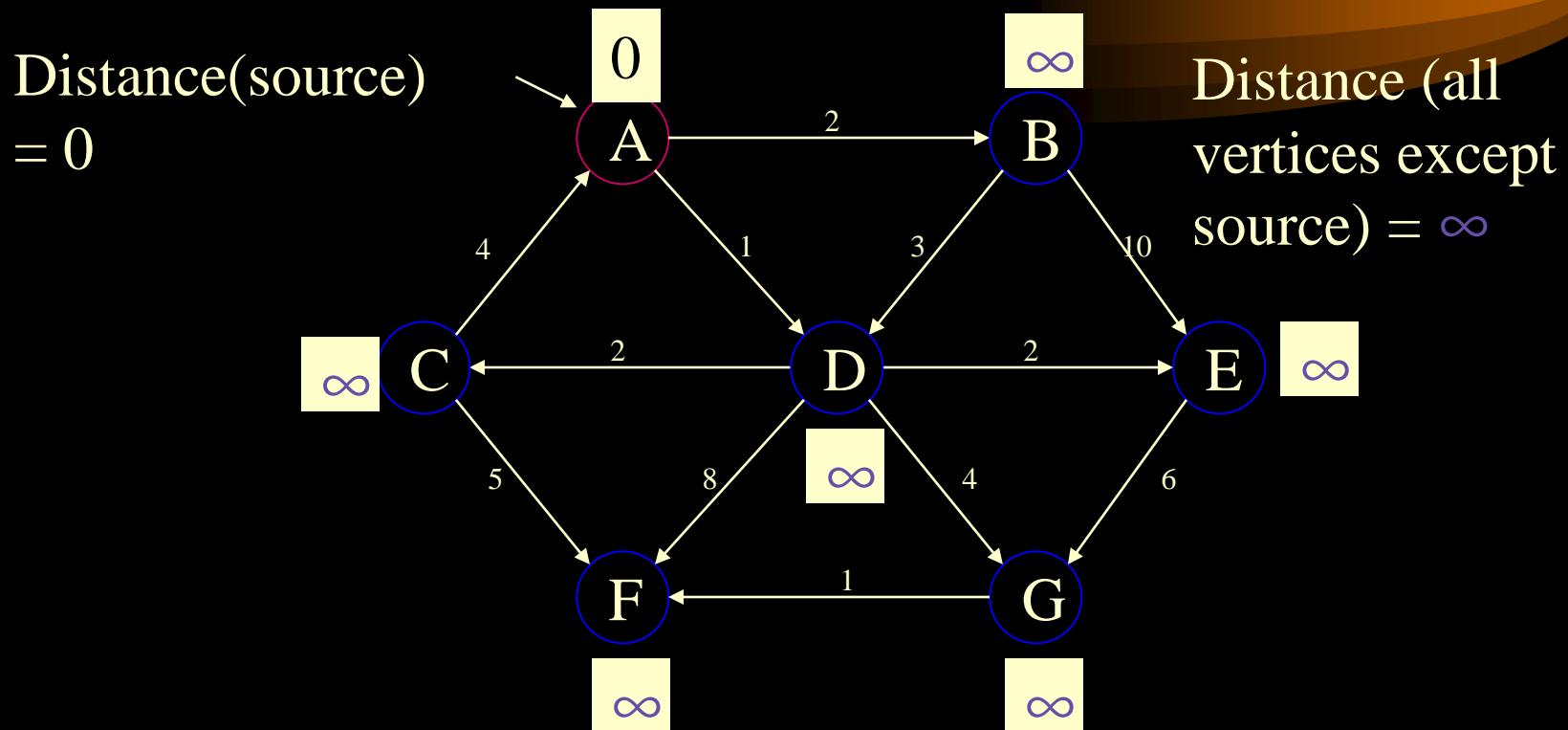
- The algorithm computes for each vertex  $u$  the **distance** to  $u$  from the start vertex  $v$ , that is, the weight of a shortest path between  $v$  and  $u$ .
- the algorithm keeps track of the set of vertices for which the distance has been computed, called the **cloud  $C$**
- Every vertex has a label  $D$  associated with it. For any vertex  $u$ ,  $D[u]$  stores an approximation of the distance between  $v$  and  $u$ . The algorithm will update a  $D[u]$  value when it finds a shorter path from  $v$  to  $u$ .
- When a vertex  $u$  is added to the cloud, its label  $D[u]$  is equal to the actual (final) distance between the starting vertex  $v$  and vertex  $u$ .

- Dijkstra\_Algorithm(source, G):
- /\*parameters: source node--> source, graph--> G
- return: List of cost from source to all other nodes--> cost \*/
- unvisited\_list = [] // List of unvisited vertices
- cost = []
- cost[source] = 0 // Distance (cost) from source to source will be 0
- for each vertex v in G: // Assign cost as INFINITY to all vertices
  - if v ≠ source
    - cost[v] = INFINITY
    - add v to unvisited\_list // All nodes pushed to unvisited\_list initially

while unvisited\_list is not empty: // Main loop

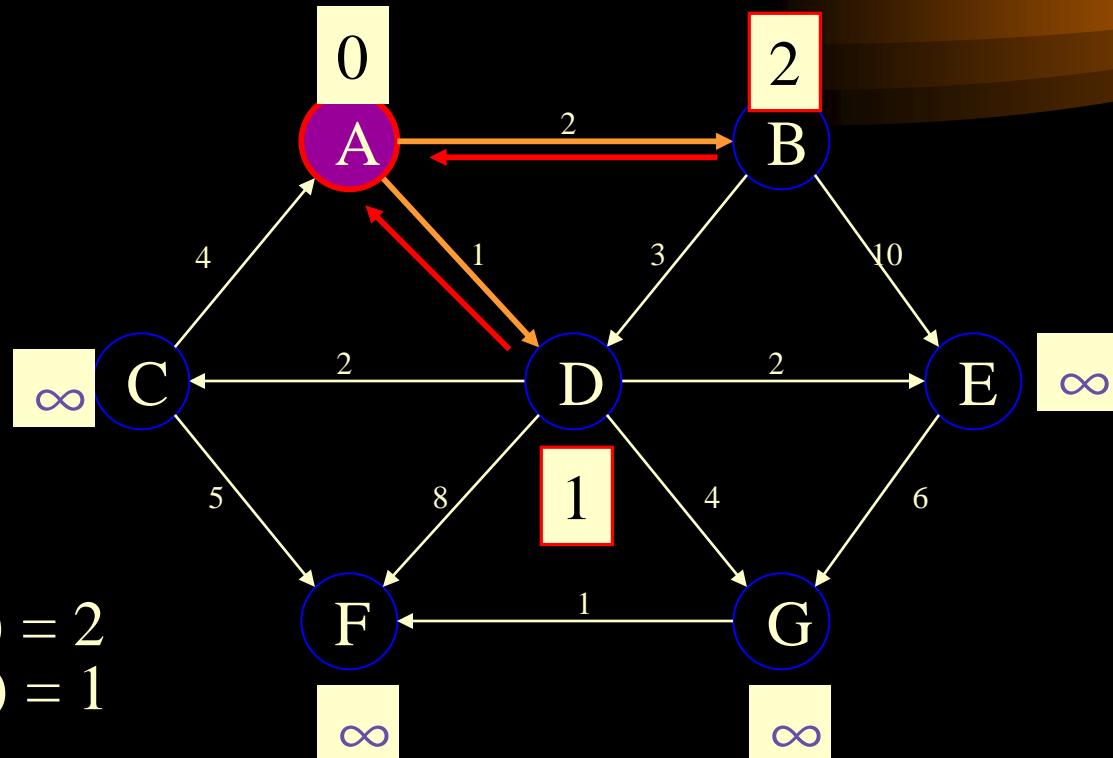
- v = vertex in unvisited\_list with min cost[v] // v is the source node for first iteration
  - remove v from unvisited\_list // Marking node as visited
  - for each neighbor u of v: // Assign shorter path cost to neighbour u
    - cost\_value = Min( cost[u], cost[v] + edge\_cost(v, u)]
    - cost[u] = cost\_value // Update cost of vertex u
- return cost

# Example: Initialization



Pick vertex in List with minimum distance.

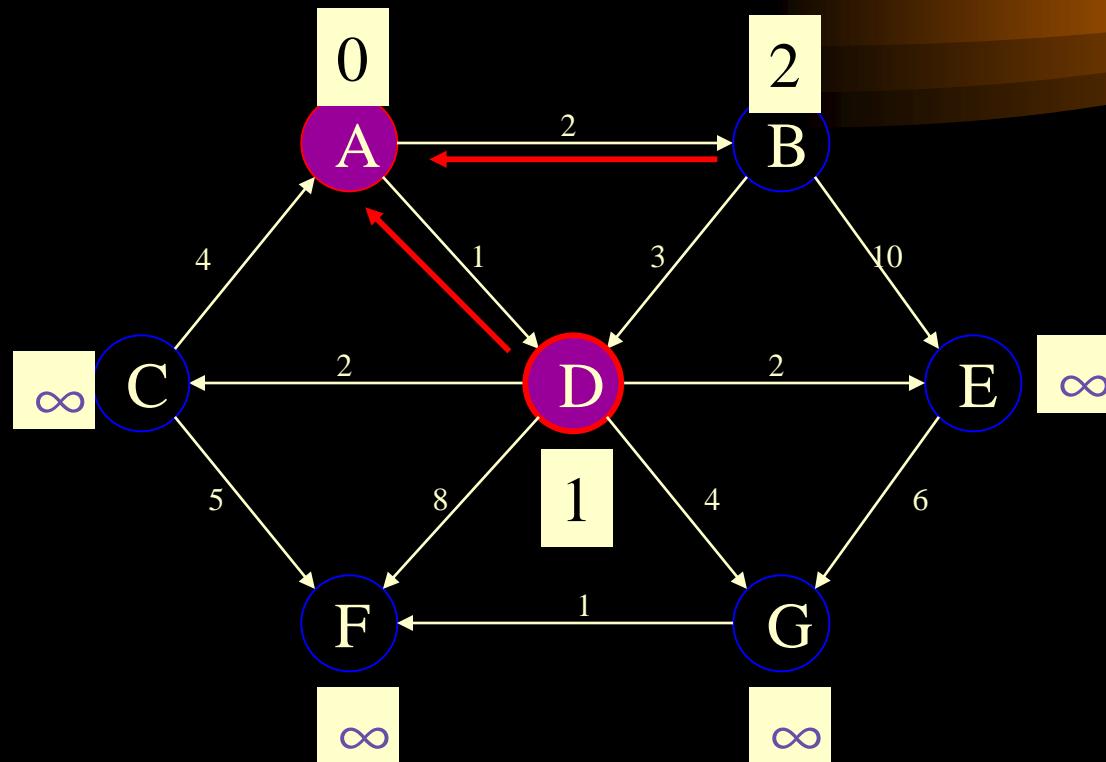
# Example: Update neighbors' distance



$$\text{Distance}(B) = 2$$

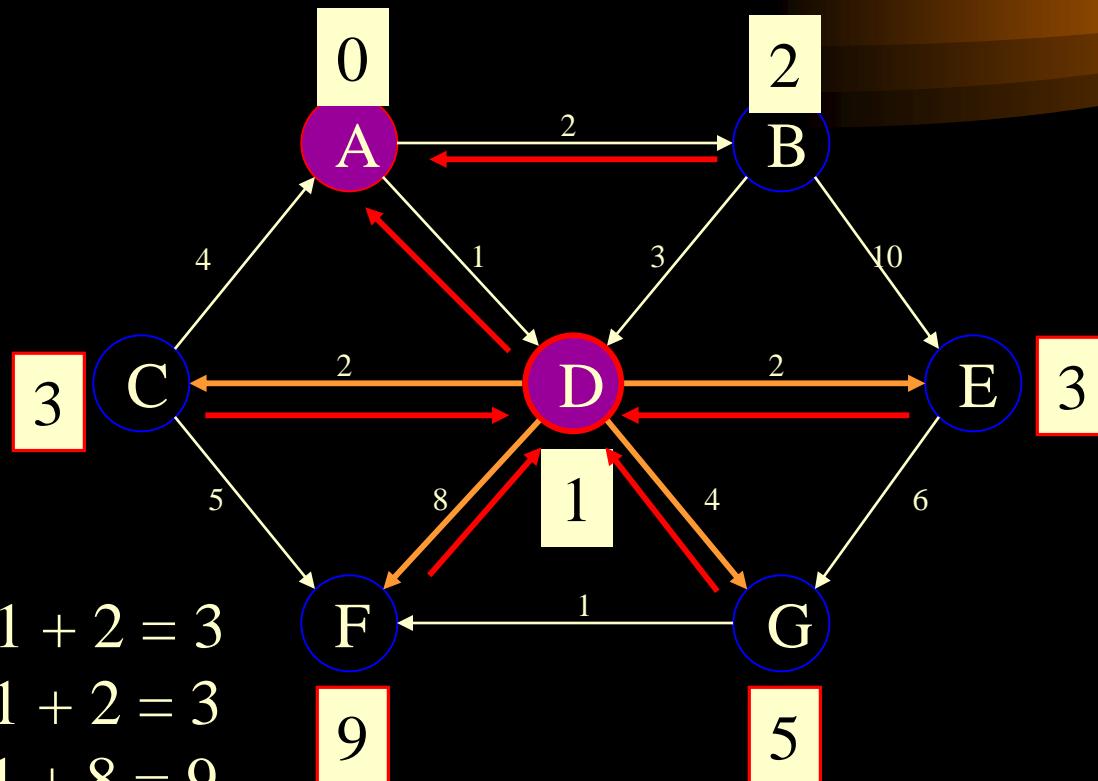
$$\text{Distance}(D) = 1$$

# Example: Remove vertex with minimum distance



Pick vertex in List with minimum distance, i.e., D

# Example: Update neighbors



$$\text{Distance}(C) = 1 + 2 = 3$$

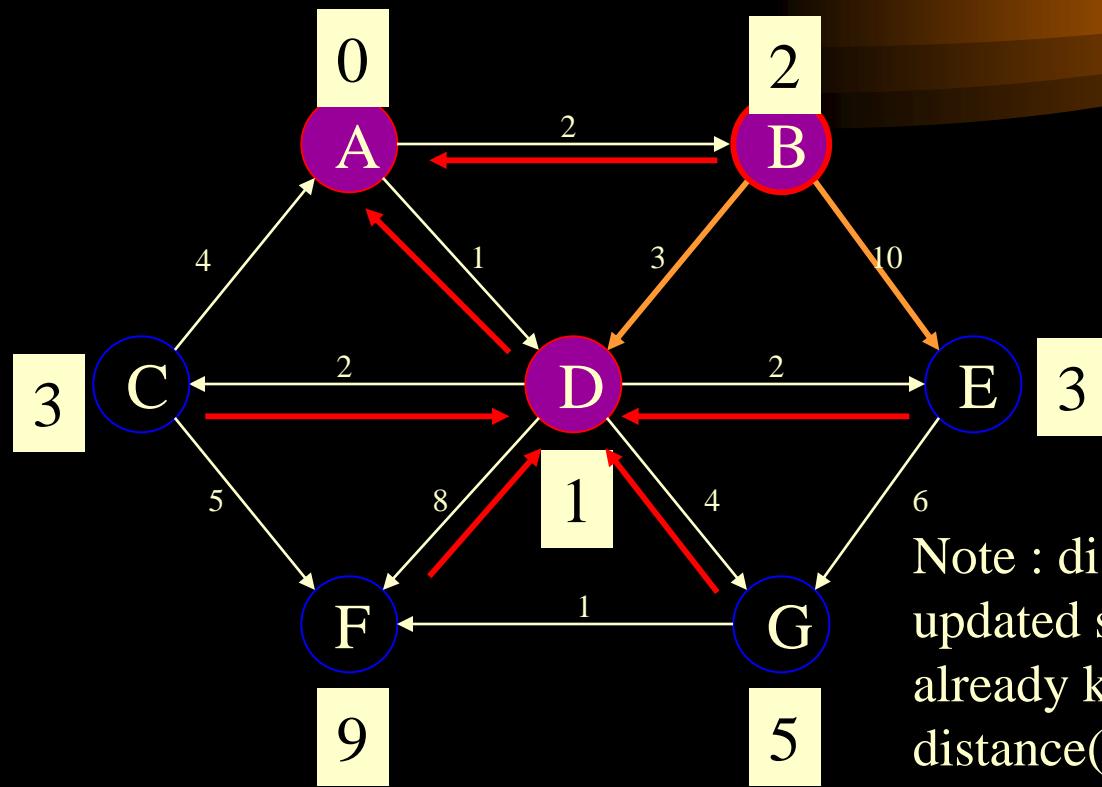
$$\text{Distance}(E) = 1 + 2 = 3$$

$$\text{Distance}(F) = 1 + 8 = 9$$

$$\text{Distance}(G) = 1 + 4 = 5$$

# Example: Continued...

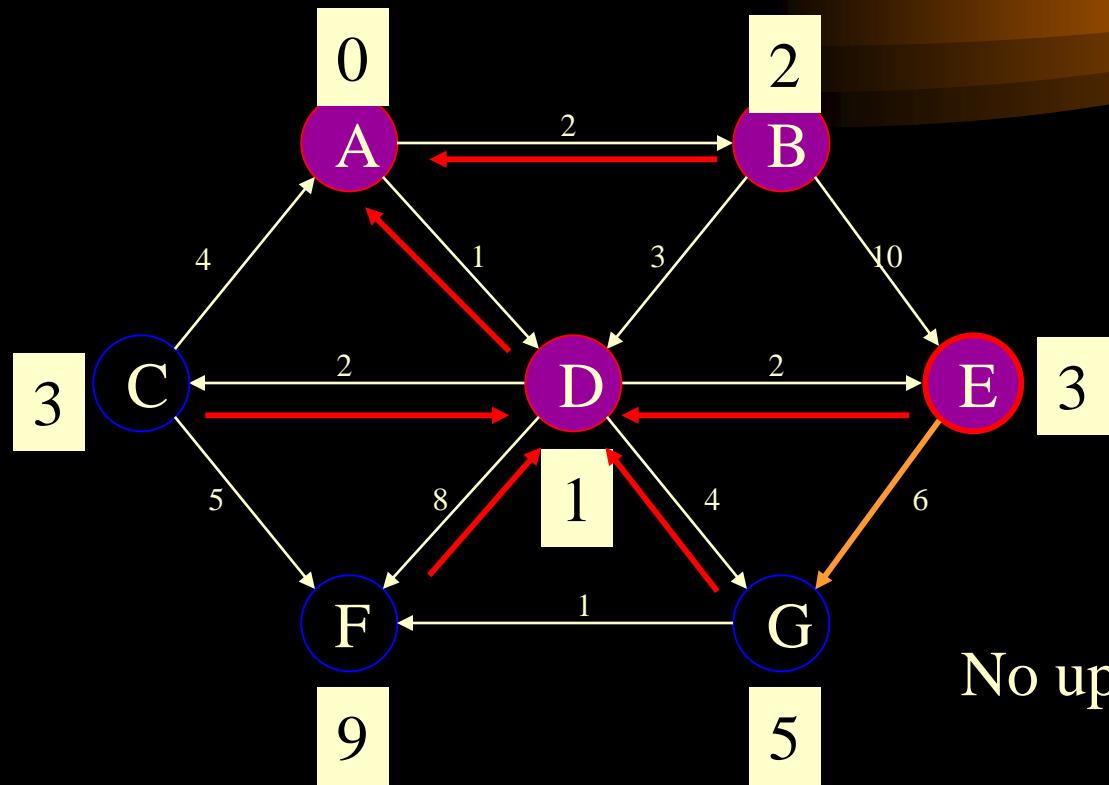
Pick vertex in List with minimum distance (B) and update neighbors



Note : distance(D) not updated since D is already known and distance(E) not updated since it is larger than previously computed

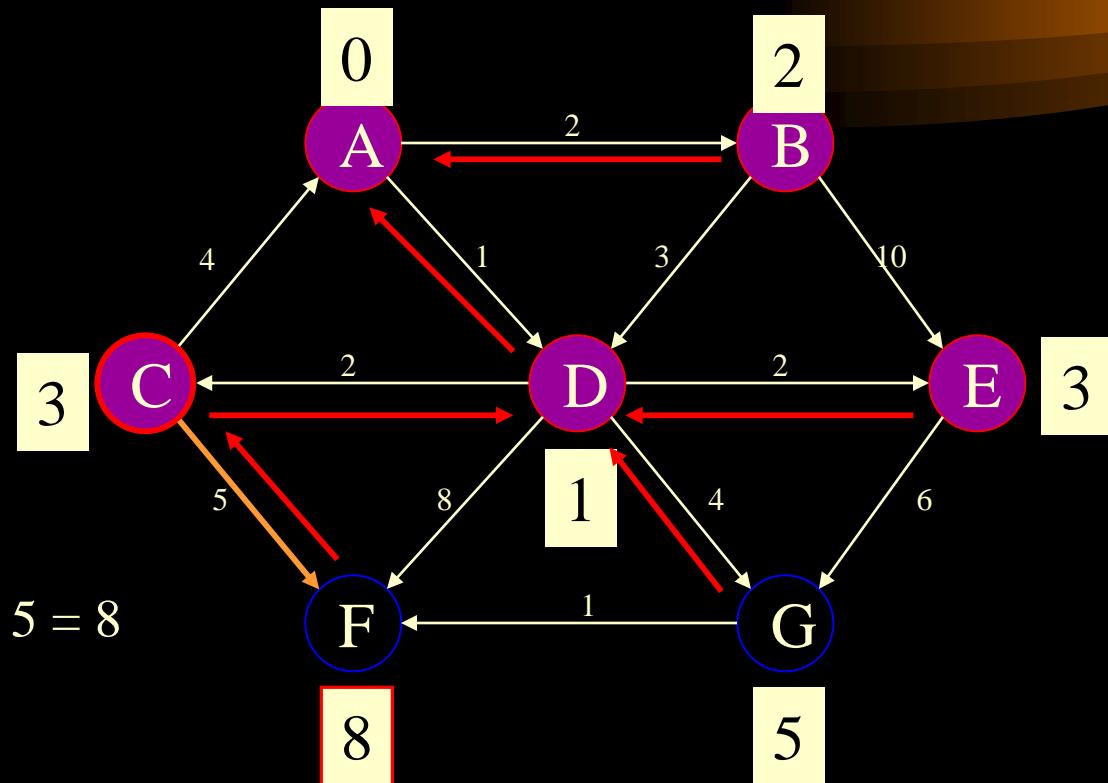
# *Example: Continued...*

Pick vertex List with minimum distance (E) and update neighbors



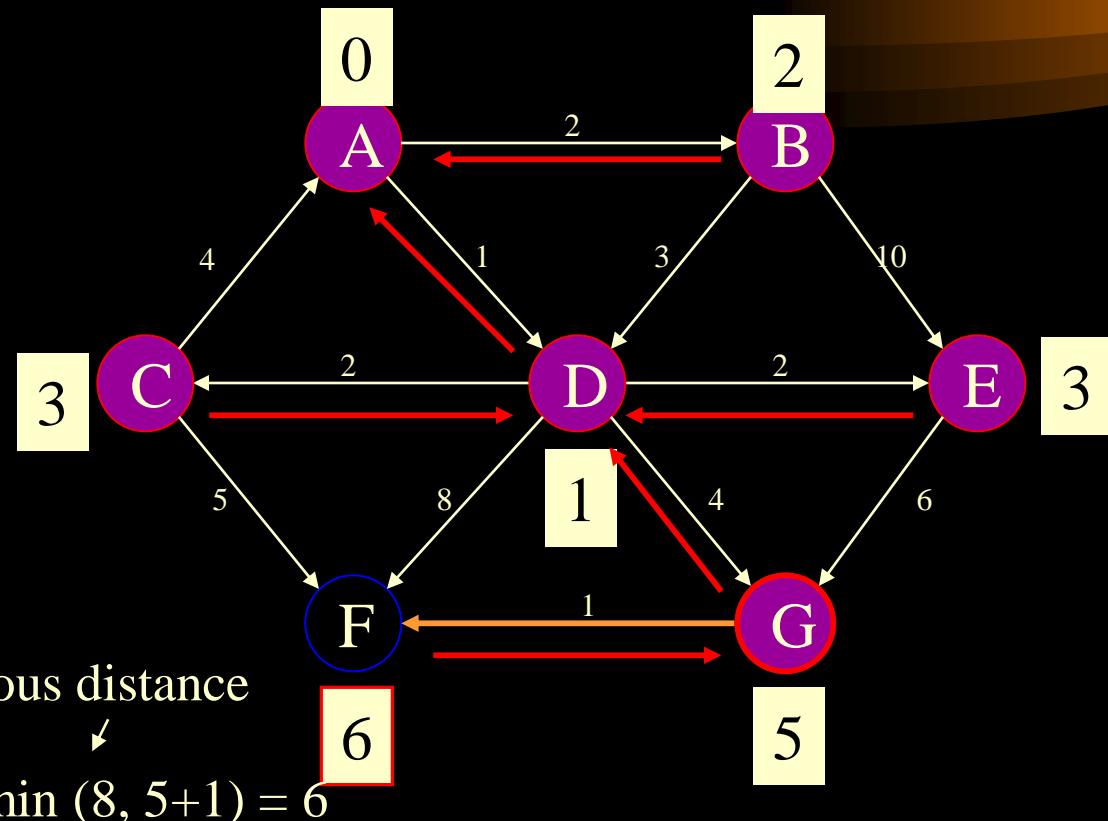
No updating

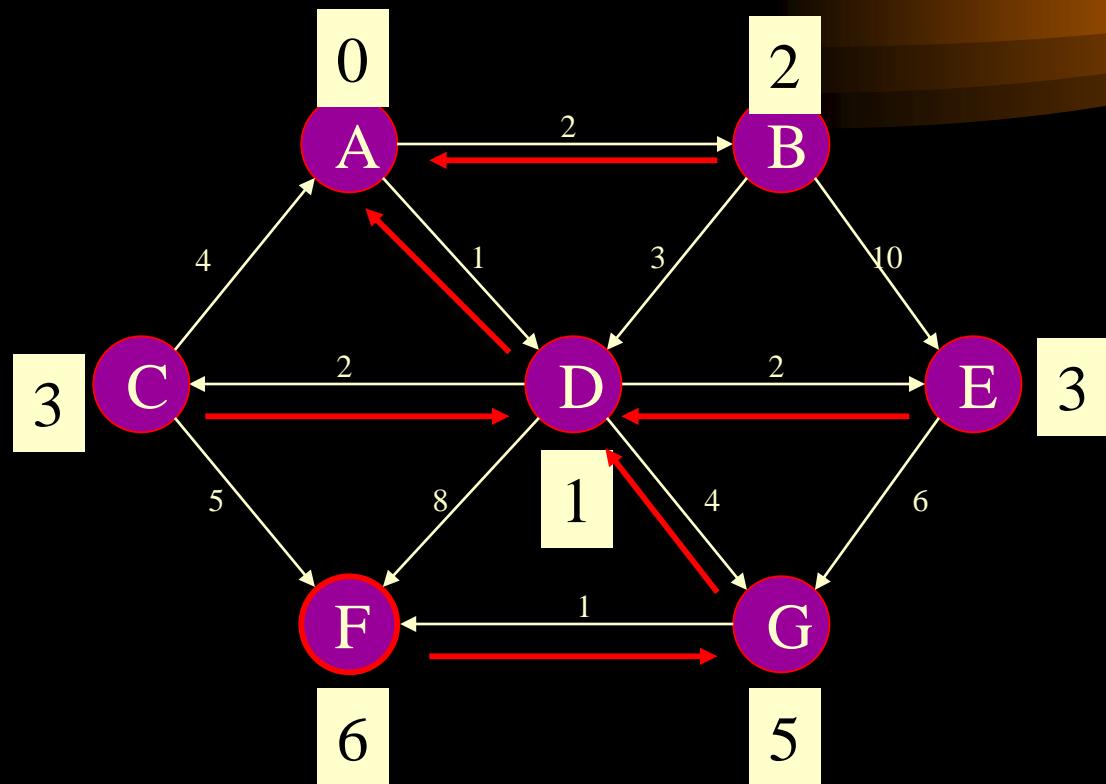
Pick vertex List with minimum distance (C) and update neighbors



$$\text{Distance}(F) = 3 + 5 = 8$$

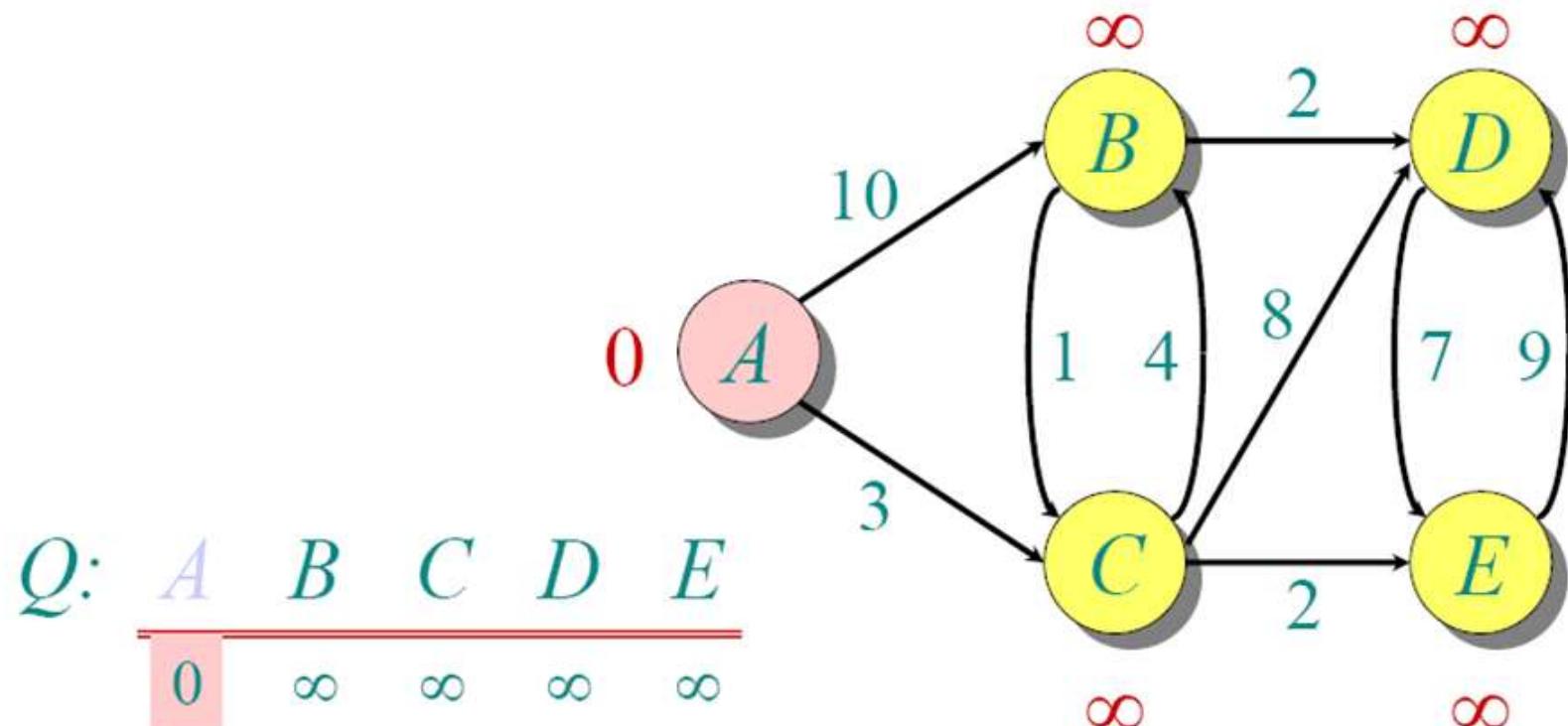
Pick vertex List with minimum distance (G) and update neighbors



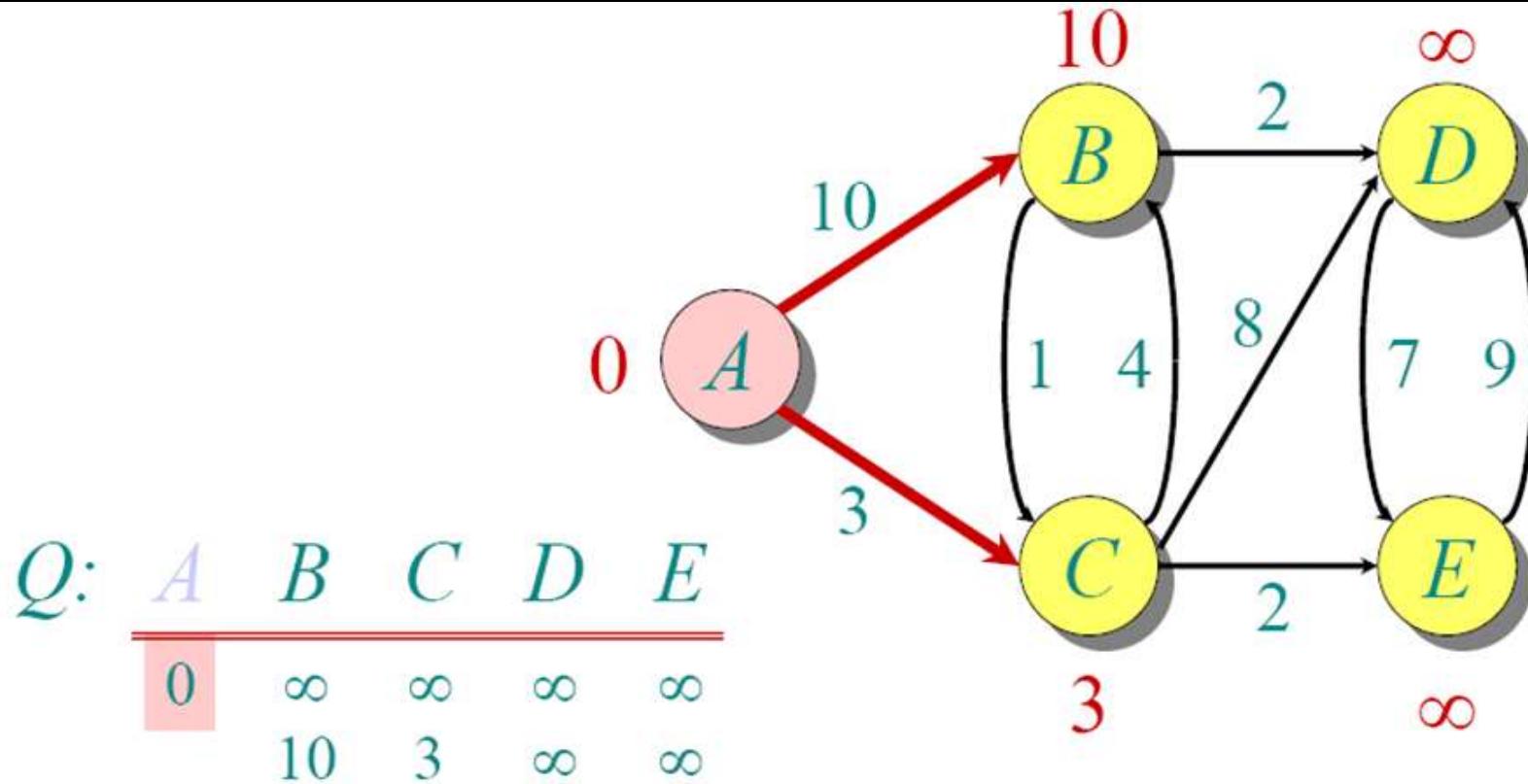


Pick vertex not in S with lowest cost (F) and update neighbors

# Another Example

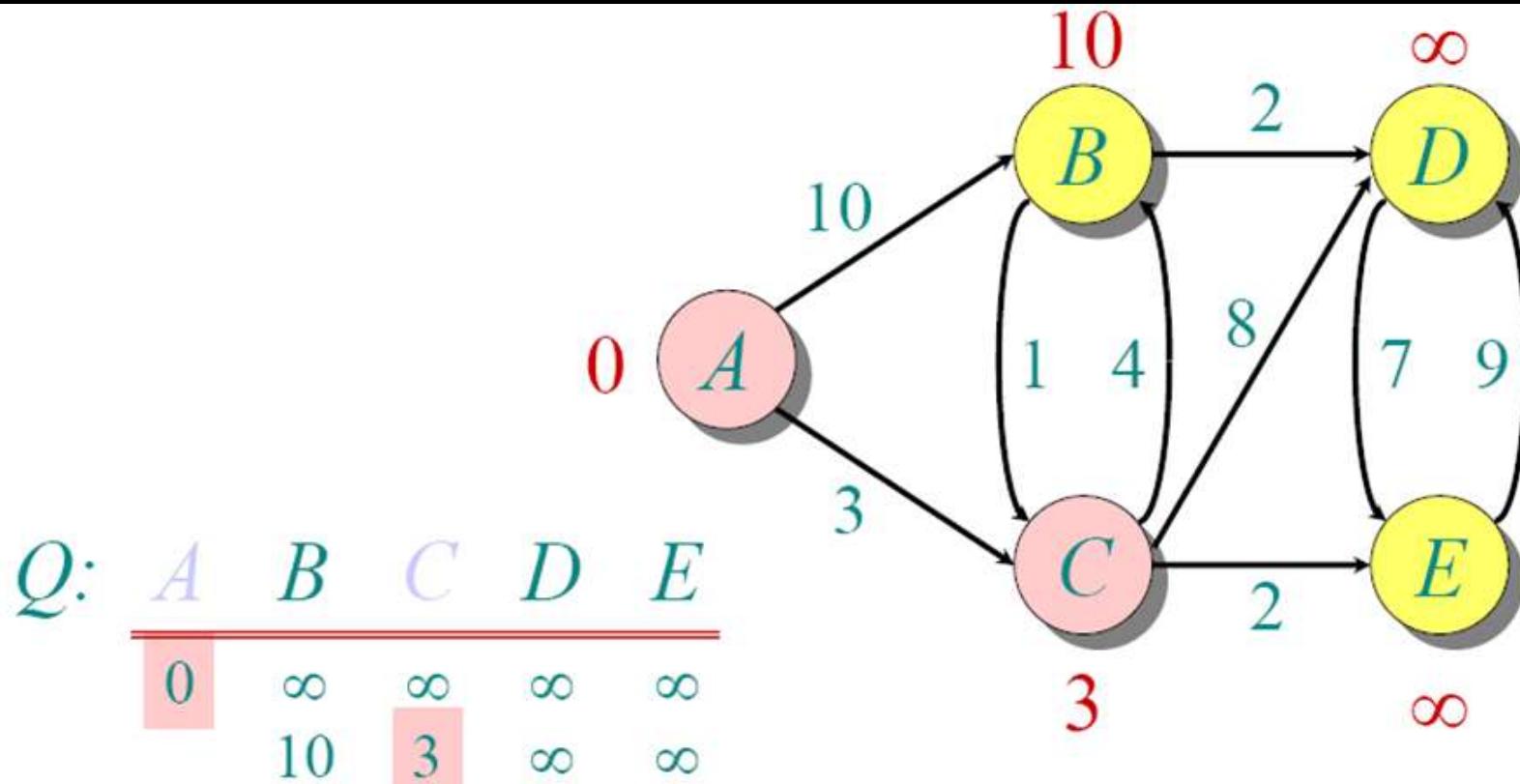


# Another Example



$S: \{ A \}$

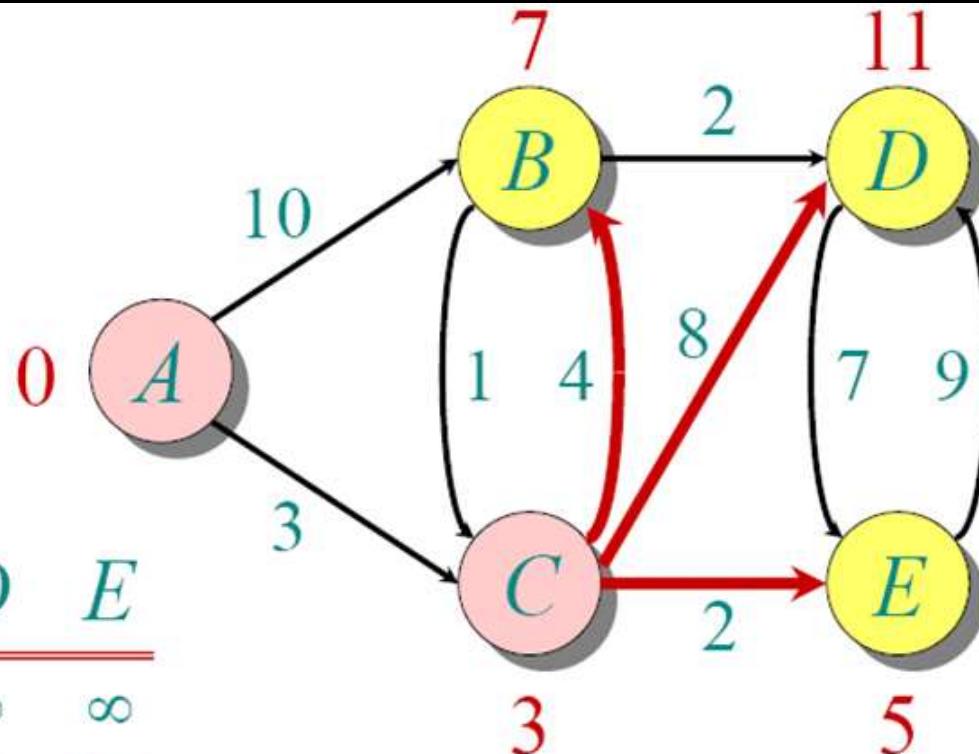
# Another Example



$S: \{ A, C \}$

# Another Example

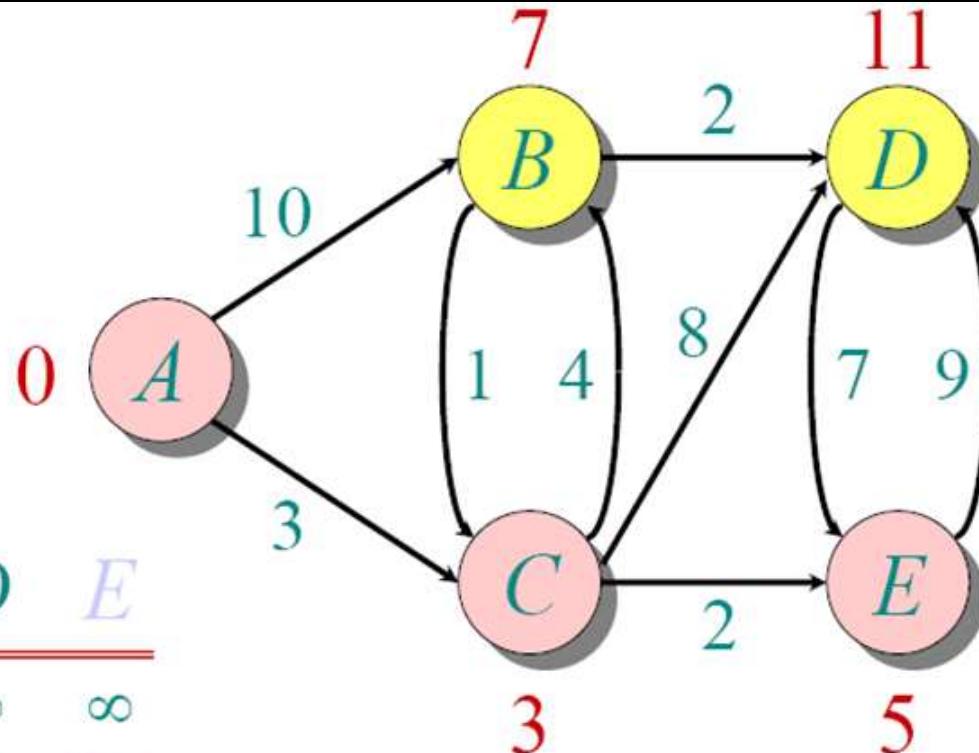
$Q:$	$A$	$B$	$C$	$D$	$E$
	0	$\infty$	$\infty$	$\infty$	$\infty$
	10	3	$\infty$	$\infty$	
	7		11	5	



$S: \{ A, C \}$

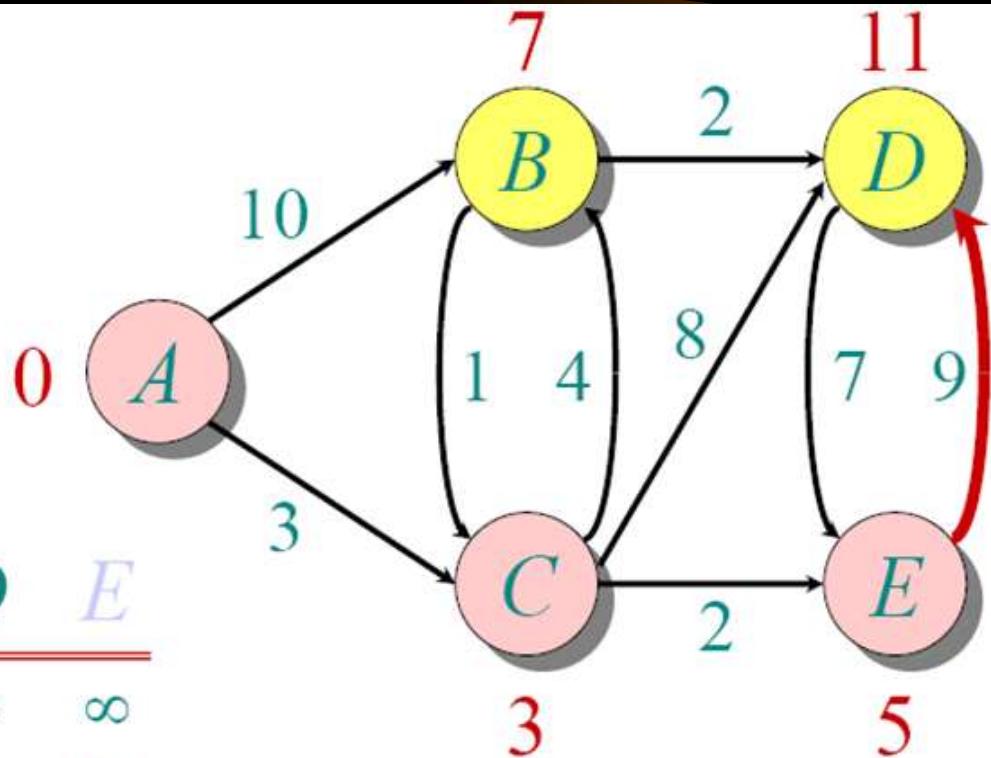
# Another Example

$Q:$	$A$	$B$	$C$	$D$	$E$
	0	$\infty$	$\infty$	$\infty$	$\infty$
	10	3	$\infty$	$\infty$	
	7		11	5	



$S: \{ A, C, E \}$

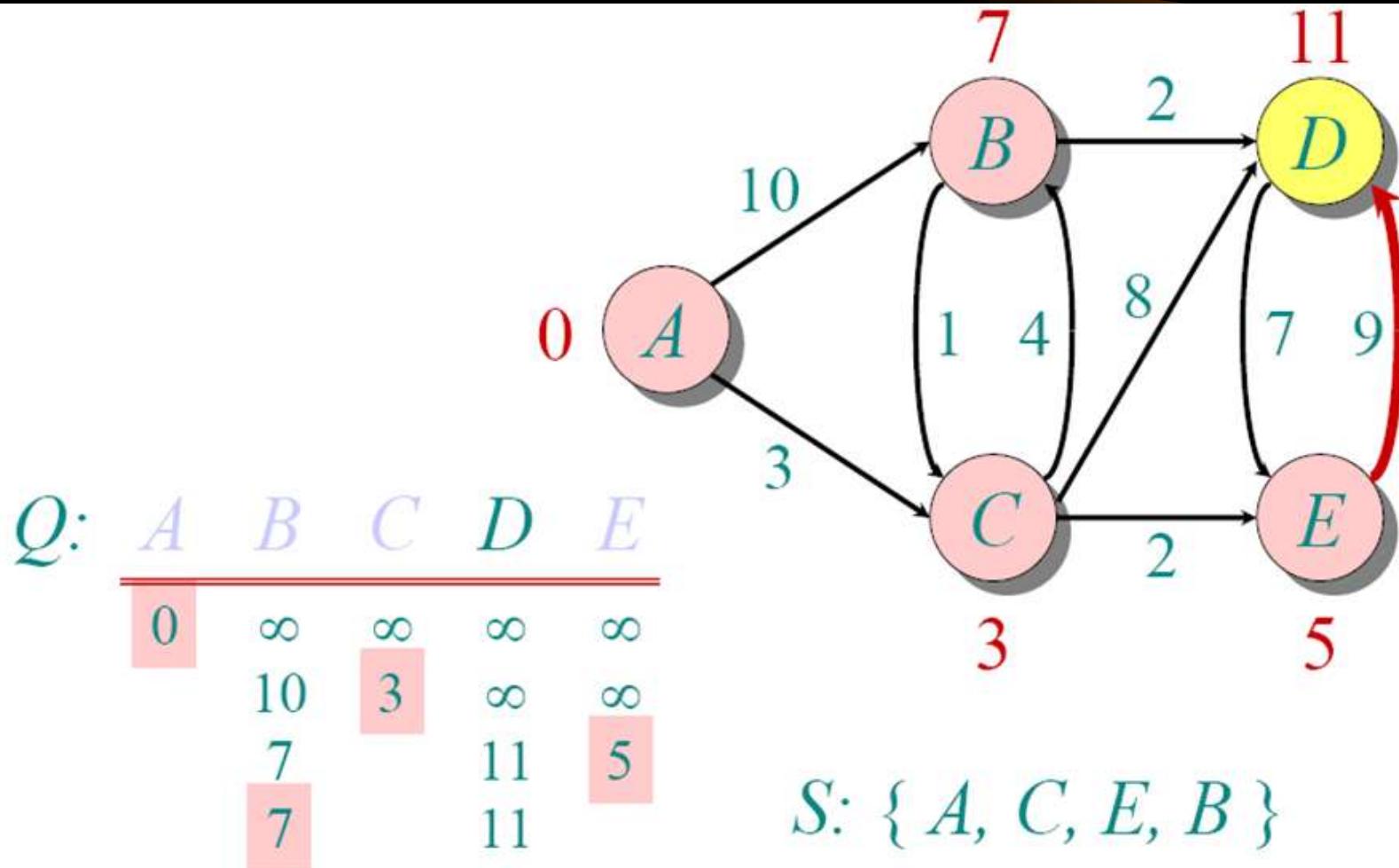
# Another Example



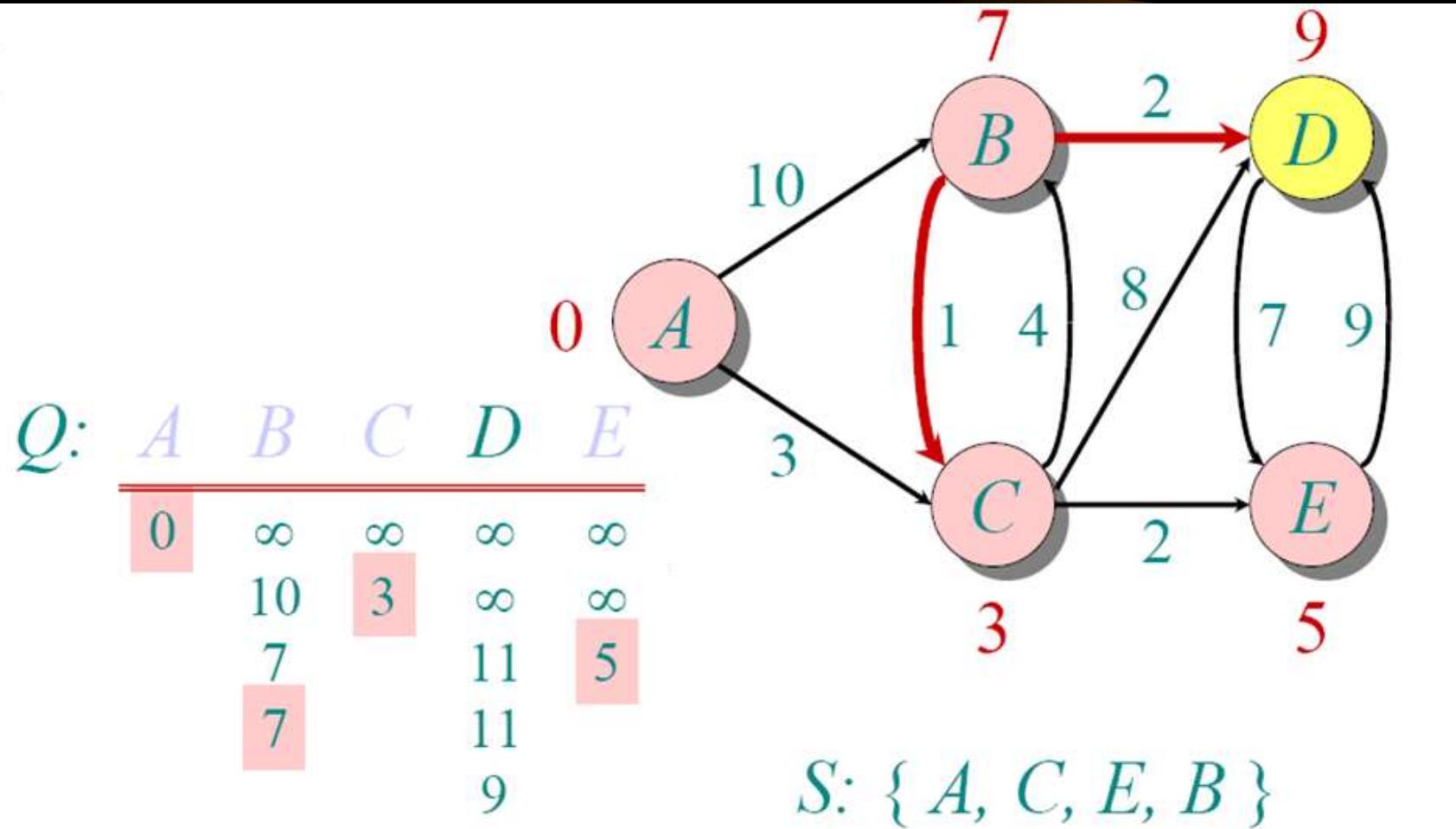
$Q:$	$A$	$B$	$C$	$D$	$E$
0	$\infty$	$\infty$			
10		3	$\infty$	$\infty$	
7		11	5		
7		11			

$S: \{ A, C, E \}$

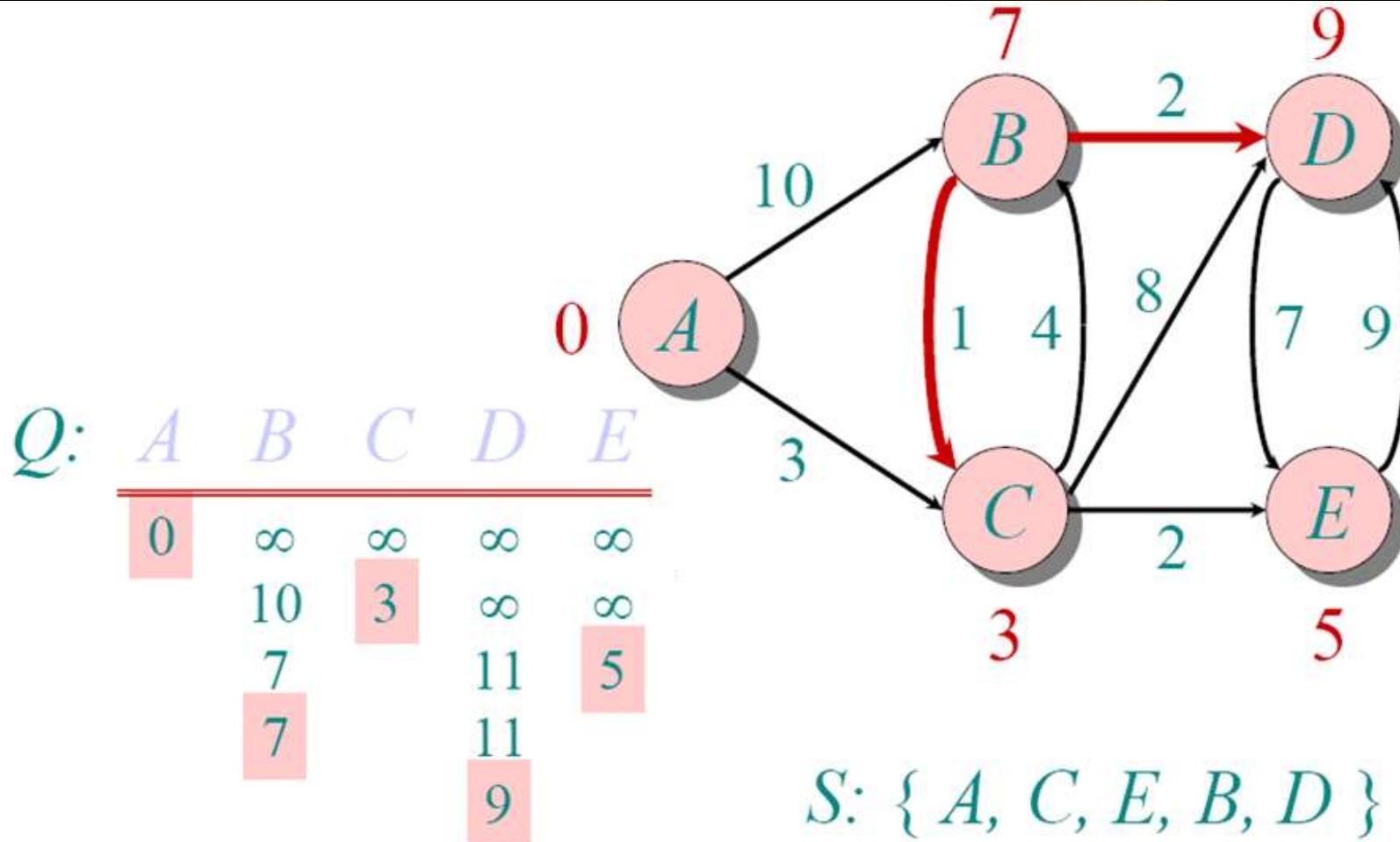
# Another Example



# Another Example



# Another Example



*Find out shortest path from A using Dijkstra's algorithm.*

