

CHAPTER-3

Non-Linear Data Structure



TREE

- Tree is used to represent and store hierarchy of data.
- Eg. Employees in company , college.
- Efficient way of storing and organizing data that is naturally hierarchical
- Tree can be also defined as collection of nodes(entities) linked together to simulate hierarchical data.
- Non-linear data structure
- Node represents the data or information and may contain link to other nodes as child.
- No bidirectional.



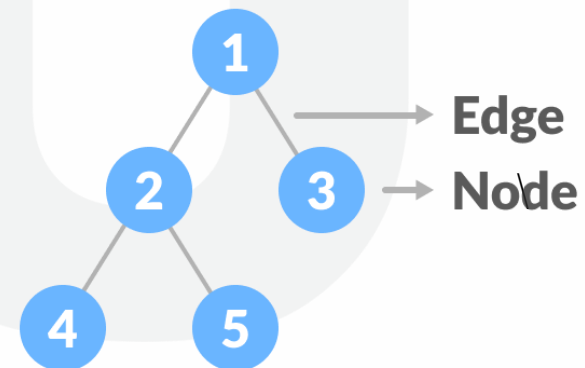
Tree Terminologies

Node

- A node is an entity that contains a key or value and pointers to its child nodes.
- The last nodes of each path are called leaf nodes or external nodes that do not contain a link/pointer to child nodes.
- The node having at least a child node is called an internal node.

Edge

- It is the link between any two nodes.



Tree Terminologies

Height of a Node

- The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node

- The depth of a node is the number of edges from the root to the node.

Height of a Tree will be height of root

- The height of a Tree is the height of the root node or the depth of the deepest node.

Degree of a Node

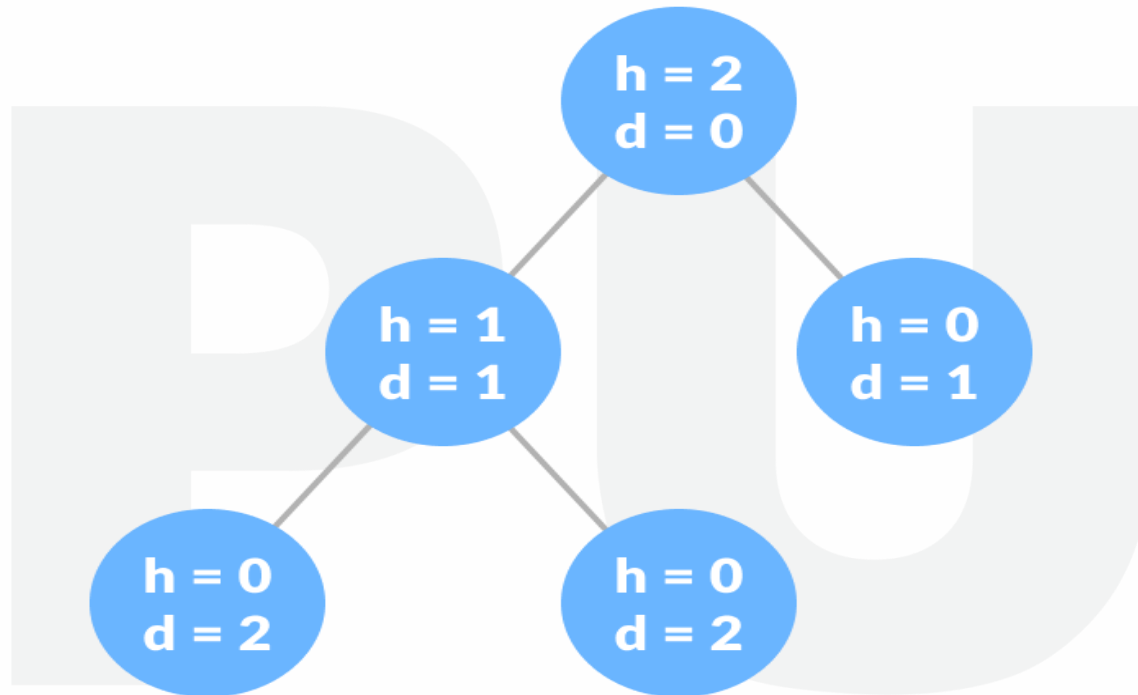
- The degree of a node is the total number of branches of that node.

Forest

- A collection of disjoint trees is called a forest.



Tree Terminologies



Applications of trees

- Storing natural hierarchical data e.g. File system.
- Organise data for quick search , insertion and deletion for e.g. Binary Search Tree.
- Trie dictionary for dynamic spell check.
- Network routing algorithm.
- And so on...



Types of Tree

1. Binary Tree
2. Binary Search Tree
3. AVL Tree
4. B-Tree

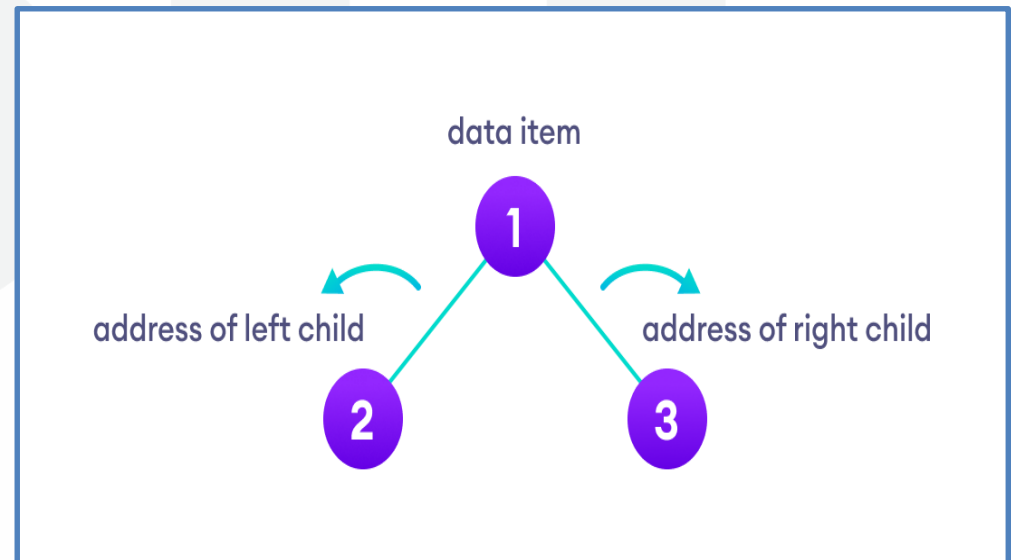
PU



Binary Tree

- A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

1. data item
2. address of left child
3. address of right child



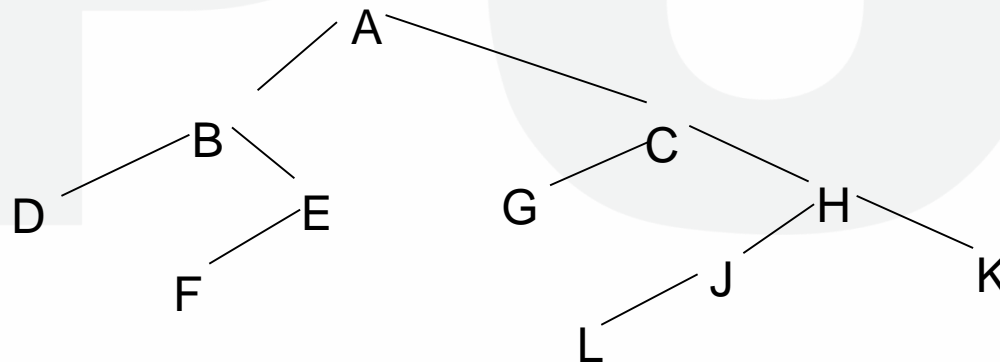
BINARY TREES

- A binary tree T is defined as a finite set of elements, called nodes, such that:
 - a) T is empty (called the null tree or empty tree), or
 - b) T contains a distinguished node R , called the root of T , and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2 .
- If T does contain a root R , then the two trees T_1 and T_2 are called, respectively, the left and right subtrees of R . If T_1 is nonempty, then its root is called the left successor of R ; similarly, if T_2 is nonempty, then its root is called the right successor of R .



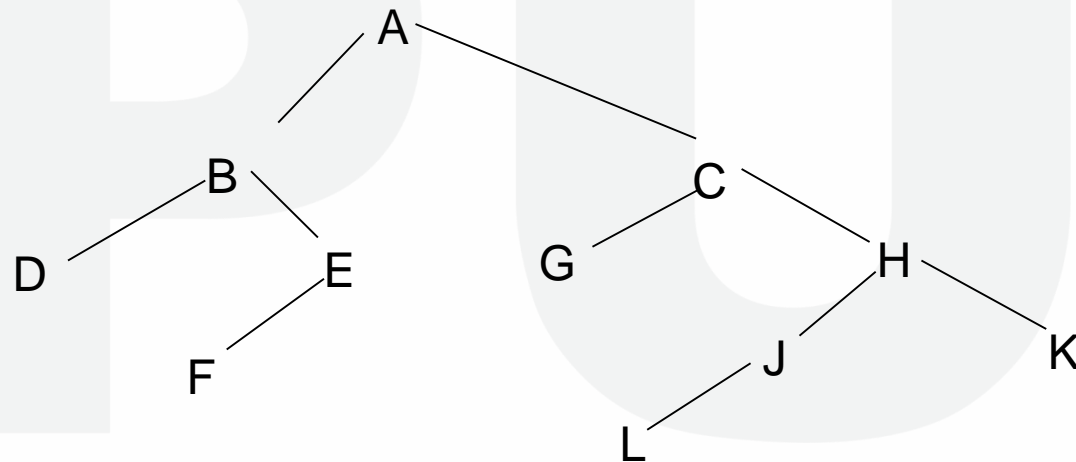
BINARY TREES

- Any node N in a binary tree T has either 0, 1 or 2 successors. The nodes A, B, C and H have two successors, the nodes E and J have only one successor, and the nodes D, F, G, L and K have no successors. The nodes with no successors are called terminal nodes.
- This means, in particular, that every node N of T contains a left and a right subtree.
- Binary tree T and T' are said to be similar if they have the same structure or, in other words, if they have the same shape. The trees are said to be copies if they are similar and if they have the same contents at corresponding nodes.



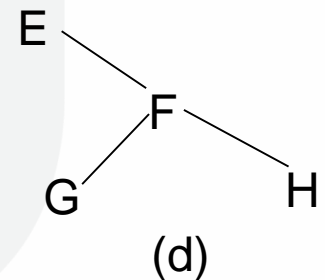
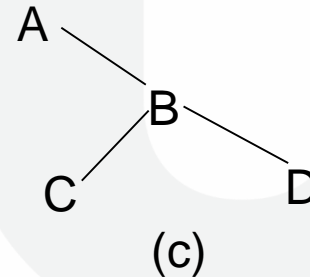
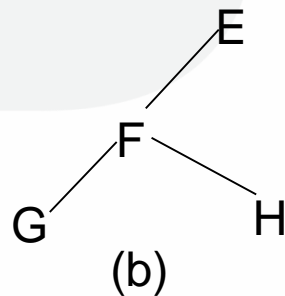
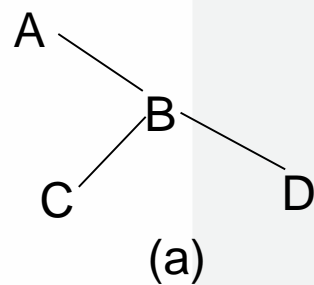
BINARY TREES

- Observe that;
 - (a) B is a left successor and C is a right successor of the node A.
 - (b) The left subtree of the root A consists of the nodes B, D, E and F, and the right subtree of A consists of the nodes C, G, H, J, K and L.



BINARY TREES

- Consider the four binary trees in fig. 7.2. The three trees (a), (c) and (d) are similar.
- In particular, the trees (a) and (c) are copies since they also have the same data at corresponding nodes. The tree (b) is neither similar nor a copy of the tree.



Complete Binary Trees

- The tree T is said to be complete if all its levels, except possibly the last, have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible.

Is This Complete?

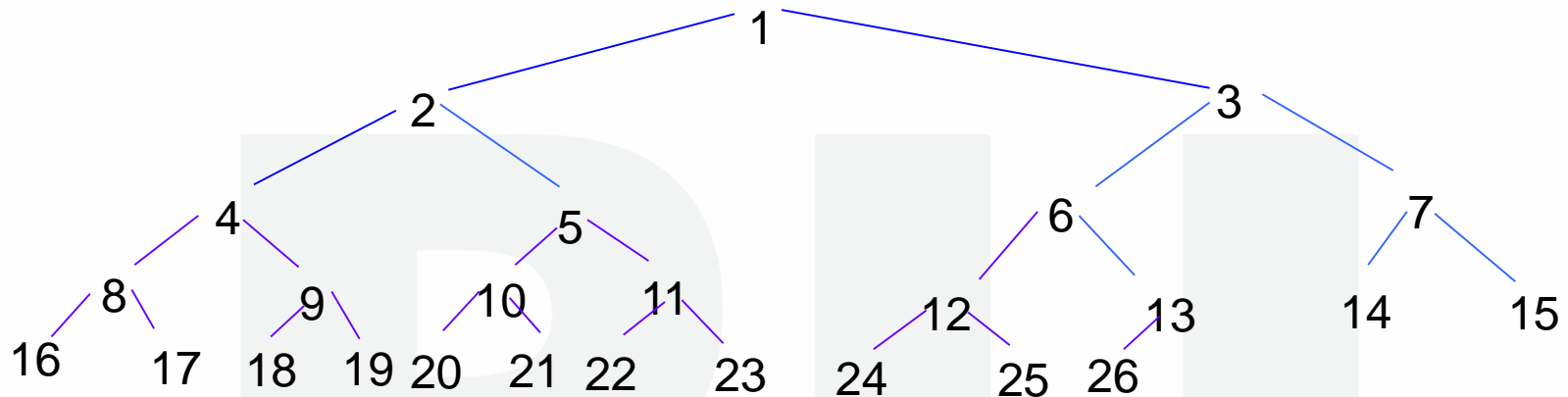
Yes!

- ✓ It is called the empty tree, and it has no nodes, not even a root.

This binary tree is also complete, .
You see, this binary tree has no nodes at all. It is called the empty tree, and it is considered to be a complete binary tree.



Complete Binary tree



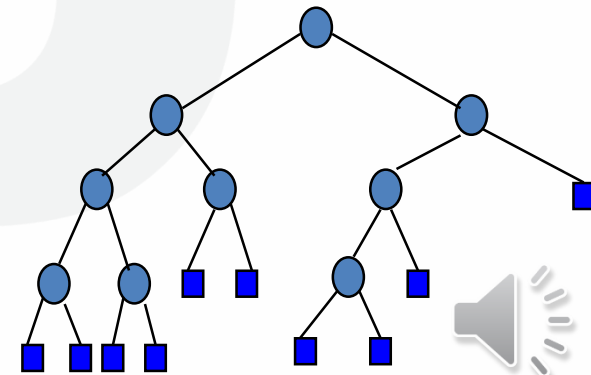
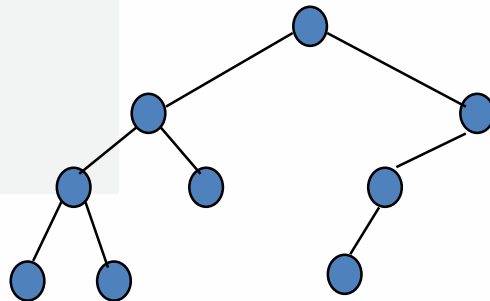
The left and right children of the node K are , respectively, $2 * K$ and $2 * K + 1$, and the parent of K is the node $[K/2]$.

The children of node 9 are the nodes 18, 19, and its parent is the node $[9/2] = 4$.



Extended Binary trees: 2 trees

- A binary tree T is said to be a 2 – tree or extended binary tree if each node N has either 0 or 2 children. The nodes with two children are called internal nodes, and the nodes with 0 children are called external nodes.
- Circles for internal nodes.**
 - Squares for external nodes.**
- Start with any binary tree and add an external node wherever there is an empty subtree.
 - Result is an extended binary tree.

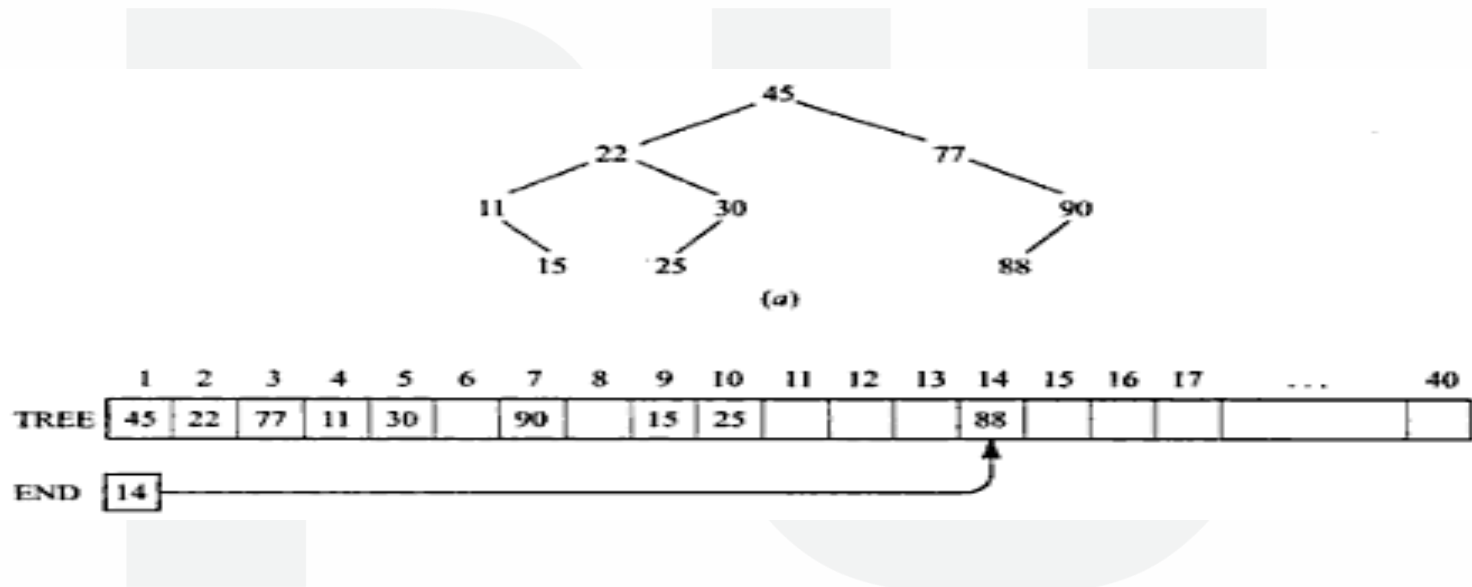


Binary Trees: Array representation

- Sequential Representation of binary Trees uses only a single linear array TREE together with a pointer variable END as follows:
- The binary tree is represented in an array by storing each element at the array position corresponding to the number assigned to it.
 - (a) The root R of T is stored in TREE[1].
 - (b) If a node occupies TREE[k], then its left child is stored in TREE[2 * K] and its right child is stored in TREE[2*k+1]
 - (c) END contains the location of the last node of T.



Binary Trees: Array representation



Linked Representation of Binary Tree

- The most popular way to present a binary tree
- Each element is represented by a node that has two link fields (leftChild and rightChild) plus an Info field
- The space required by an n node binary tree is $n * \text{sizeof}(\text{binaryTreeNode})$



Linked Representation of Binary trees

- INFO[K] contains the data at the node N
- LEFT[K] contains the location of the left child of node N
- RIGHT[K] contains the location of the right child of node N.

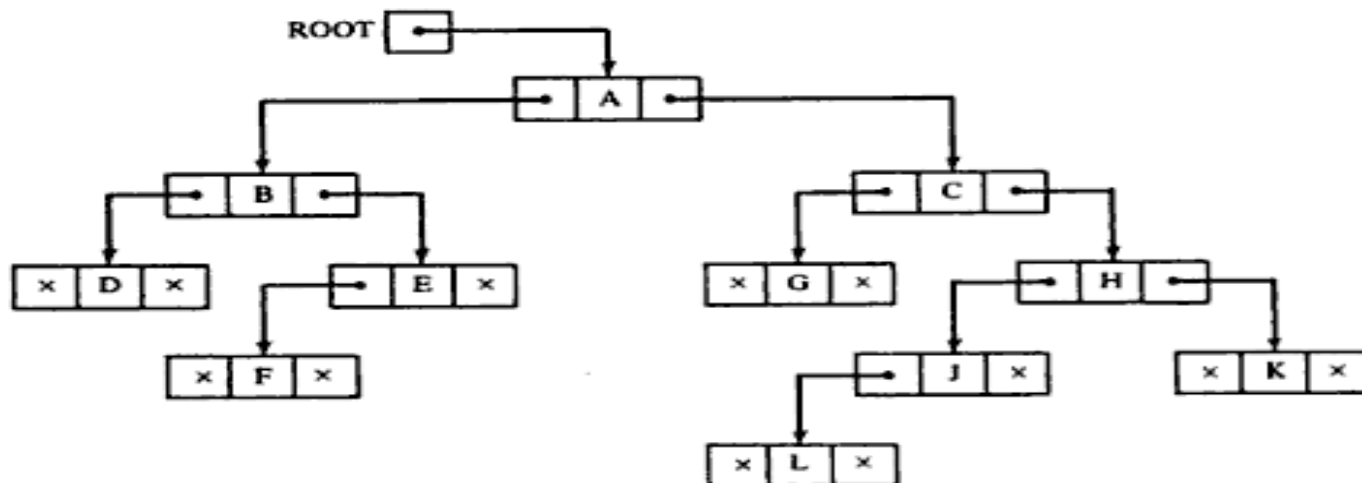
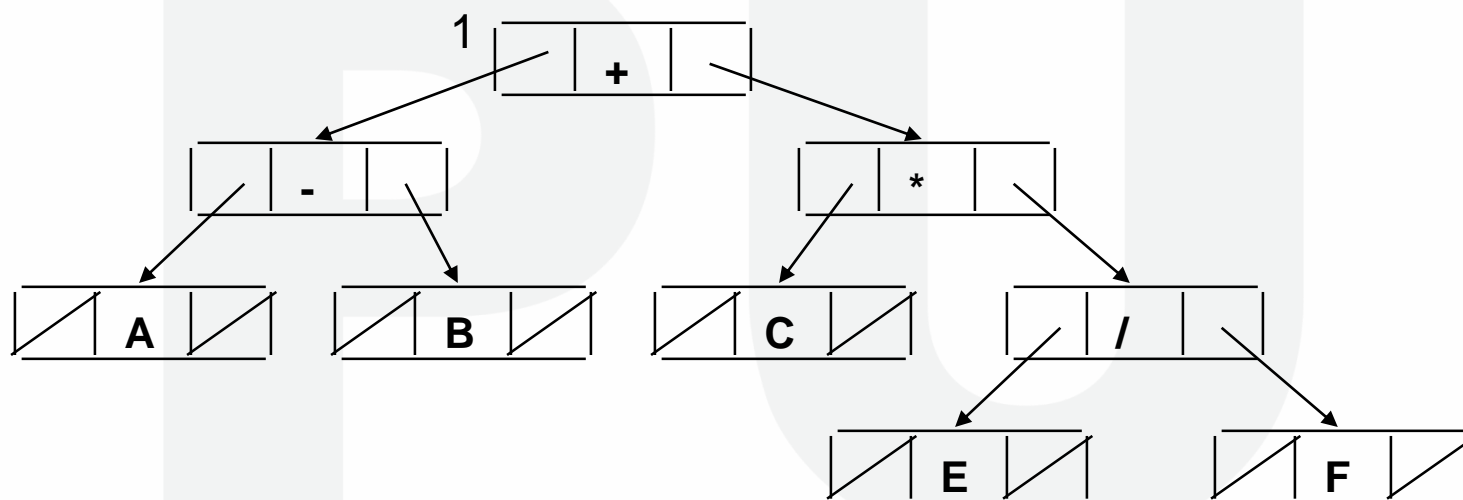


Fig. 10-6



Linked representation of a binary tree

- Linked representation uses explicit links to connect the nodes. Example:



Binary tree (linked representation)

- We can use a positional sequence ADT to implement a binary tree. Our
- example tree, in this case, we be represented as follows:

position	1	2	3	4	5	6	7	8	9
data	+	-	A	B	*	C	/	E	F
leftChild	2	3	null	null	6	null	8	null	null
rightChild	5	4	null	null	7	null	9	null	null
parent	null	1	2	2	1	5	5	7	7



Linear representation of a binary tree

- **Advantages of linear representation:**

1. Simplicity.
2. Given the location of the child (say, k), the location of the parent is easy to determine ($k / 2$).

- **Disadvantages of linear representation:**

1. Additions and deletions of nodes are inefficient, because of the data movements in the array.
 2. Space is wasted if the binary tree is not complete. That is, the linear representation is useful if the number of missing nodes is small.
- Linear representation of a binary tree can be implemented by means of a linked list instead of an array
 - This way the above mentioned disadvantages of the linear representation is resolved.



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

Traversal of Binary Trees

There are three ways to traverse a tree, and these are:

- In-Order (Left-Root-Right)
- Pre-Order (Root-Left-Right)
- Post-Order (Left-Right-Root)

These traversal methods can be implemented in a recursive manner as you will notice:

Traversals

To Traverse a non-empty binary tree in In-Order, we need

to perform the following operations:

1. Visit the Left Subtree in In-Order
2. Visit the Root
3. Visit the Right Subtree in In-Order

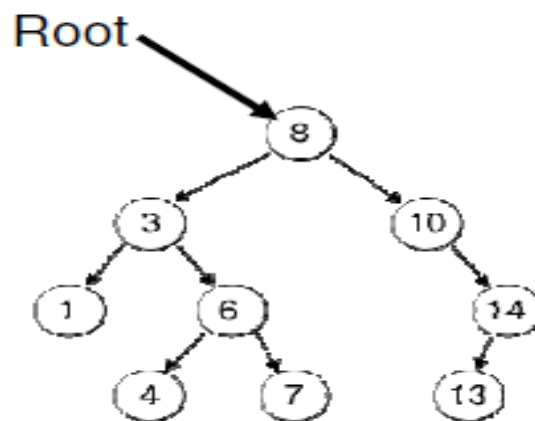
To Traverse a non-empty binary tree in Pre-Order, we need to perform the following operations:

1. Visit the Root
2. Visit the Left Subtree in Pre-Order
3. Visit the Right Subtree in Pre-Order

Tree Traversal

To Traverse a non-empty binary tree in Post-Order, we need to perform the following operations:

1. Visit the Left Subtree in Post-Order
2. Visit the Right Subtree in Post-Order
3. Visit the Root



Question

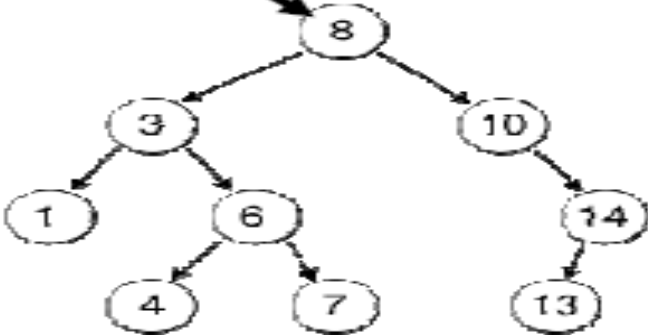
Print the value in each node after an:

- In-Order Traversal
- Pre-Order Traversal
- Post-Order Traversal



Tree Traversal

Root



```
graph TD; Root((Root)) --> 8((8)); 8 --> 3((3)); 8 --> 10((10)); 3 --> 1((1)); 3 --> 6((6)); 6 --> 4((4)); 6 --> 7((7)); 10 --> 14((14)); 14 --> 13((13));
```

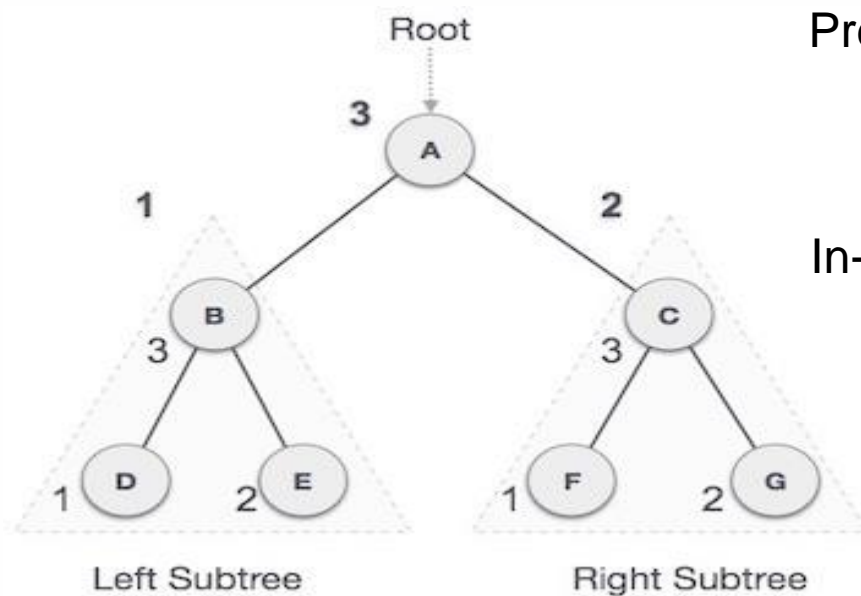
Answer

Print the value in each node after an:

- In-Order Traversal
{1,3,4,6,7,8,10,13,14}
- Pre-Order Traversal
{8,3,1,6,4,7,10,14,13}
- Post-Order Traversal
{1,4,7,6,3,13,14,10,8}



Tree Traversal



Pre-order Traversal

$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

In-order Traversal

$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$

Post-order Traversal

$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

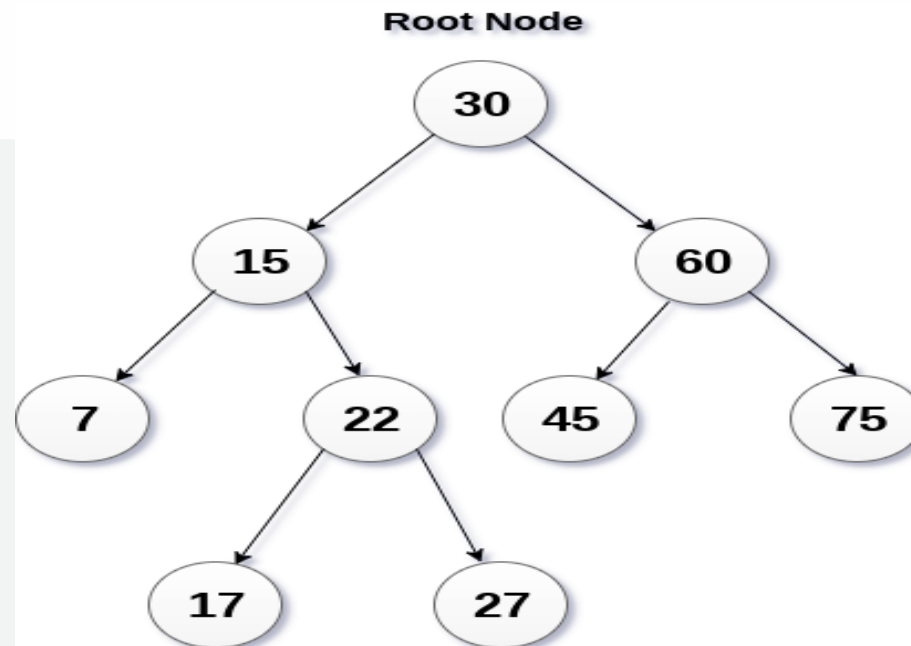


Binary Search Tree

- Binary Search tree can be defined as a class of binary trees, in which the nodes are arranged in a specific order. This is also called ordered binary tree.
- In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
- Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
- This rule will be recursively applied to all the left and right sub-trees of the root.



Binary Search Tree



Binary Search Tree



Advantages of using binary search tree

- Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
- The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes $O(\log_2 n)$ time. In worst case, the time it takes to search an element is $O(n)$.
- It also speed up the insertion and deletion operations as compare to that in array and linked list.





Operations on Binary Tree

SN	Operation	Description
1	Searching in BST	Finding the location of some specific element in a binary search tree.
2	Insertion in BST	Adding a new element to the binary search tree at the appropriate location so that the property of BST do not violate.
3	Deletion in BST	Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have.





Insertion

- Lexically Ordered Binary tree
- In Lexically ordered Binary Tree, the creation of a BT could be based on information associated with each node. This ordering could be numerical (either ascending or descending), or it could be a list of names to be kept in lexicographical order.
- That is, the left subtree of the tree is to contain nodes whose associated names are lexically less than the name associated with the root node of the tree. Similarly, the right subtree of the tree is to contain nodes whose associated names are lexically greater than the name associated with the root node of the tree.



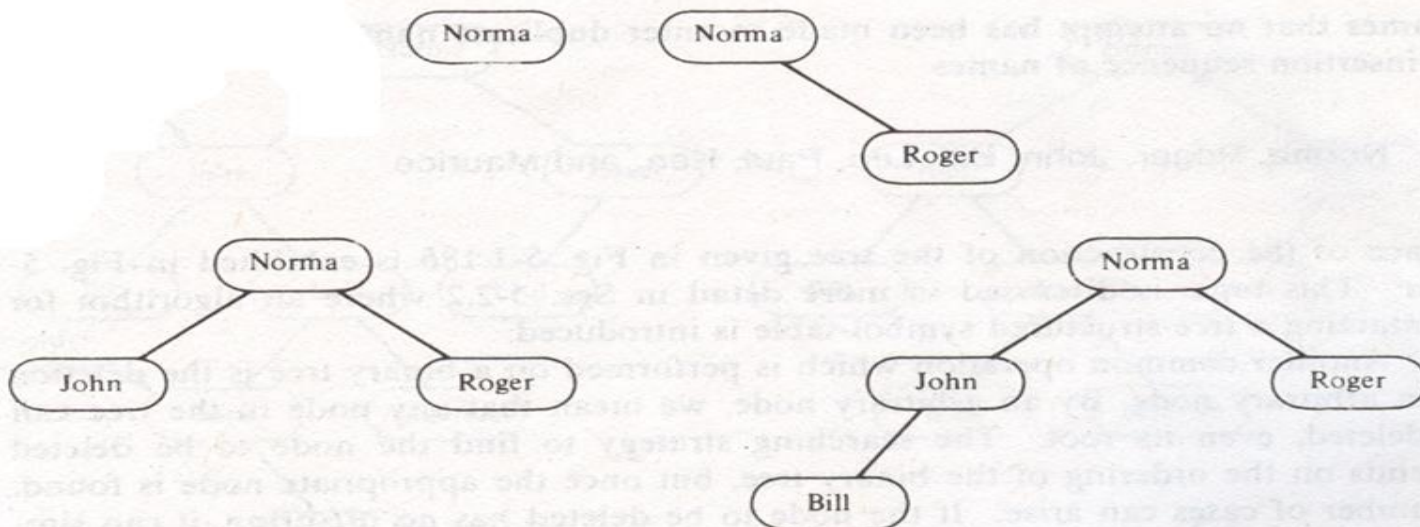
Insertion

- Given preorder traversal of a binary search tree, construct the BST.
- For example, if the given traversal is
- {10, 5, 1, 7, 40, 50}.
- {7, 4, 12, 3, 6, 8, 1, 5, 10}

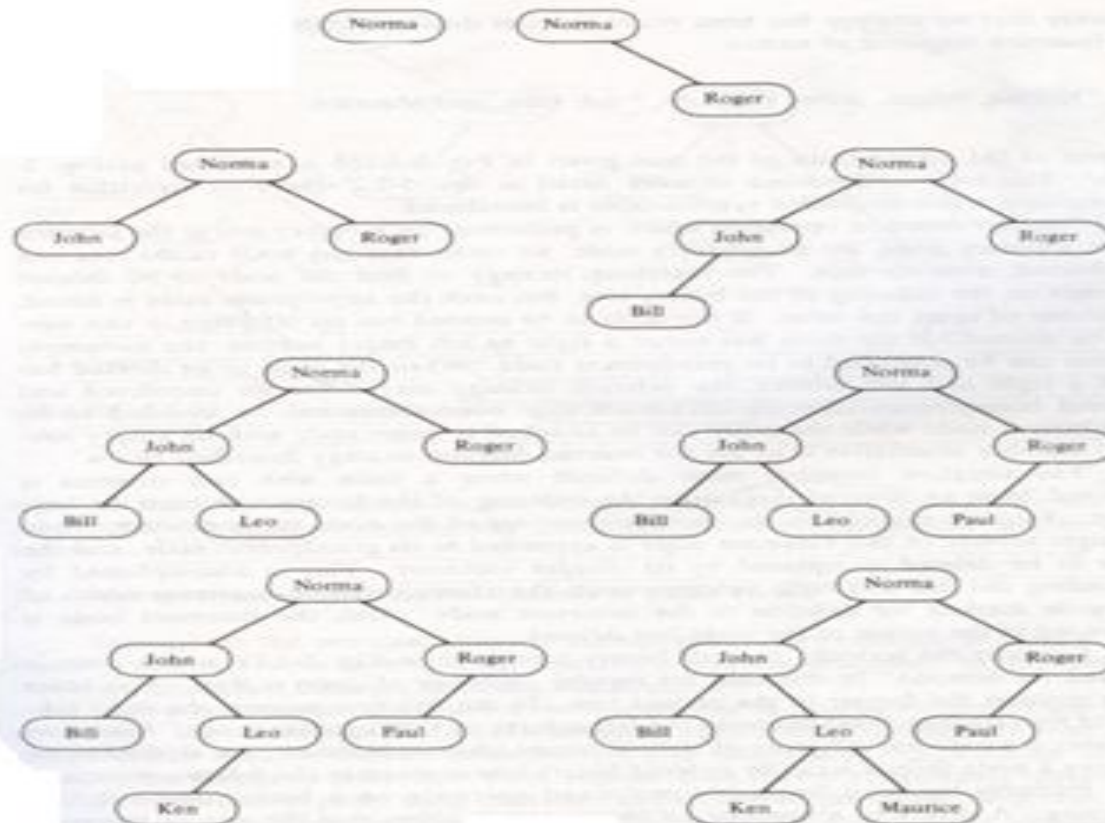


Insertion

- Construct the Lexically Ordered Binary Tree for Norma, Roger, John, bill, Leo, Paul, Ken and Maurice



Insertion - Norma, Roger, John, bill, Leo, Paul, Ken and Maurice



Deletion

- Deletion of an arbitrary node from the given binary tree. The searching strategy to find the node to be deleted depends on the ordering of the binary tree, if the node is found then,

Case 1 : If the node to be deleted has no offspring, it can simply be deleted.

Case 2 : If the node has either a right or left empty subtree, the nonempty subtree can be appended to its grandparent node.

Case 3 : When the node to be deleted has both right and left subtrees, then deletion strategy may differ for unordered and ordered binary tree.



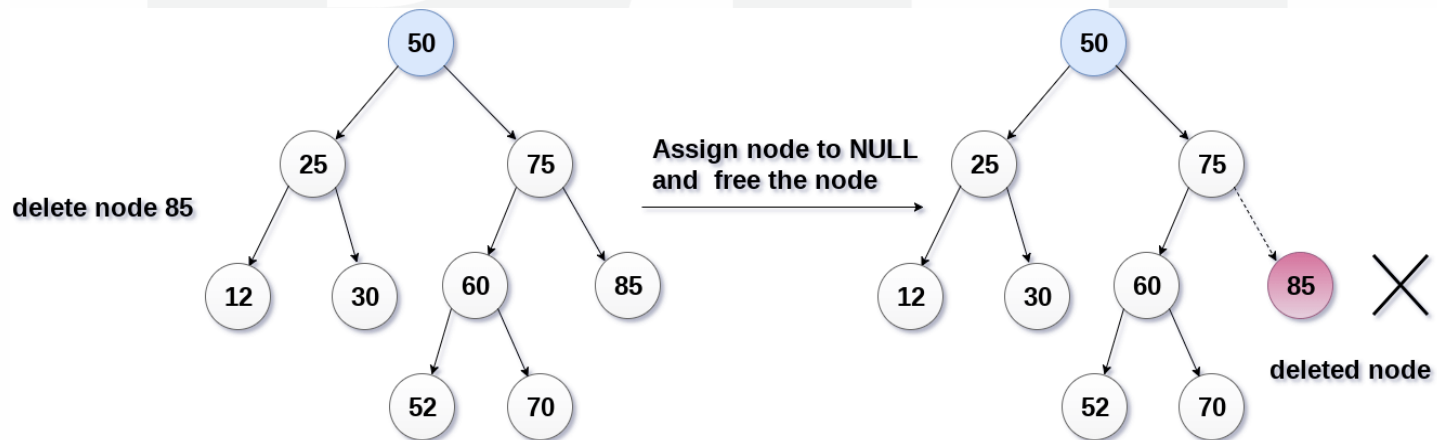
Deletion

- For an unordered tree, one subtree can be attached to its grandparent node while the other can be attached to some node with an empty subtree.
- For an ordered tree,
 1. Obtain the inorder successor of the node to be deleted.
 2. Then the right subtree of this successor node is appended to its grandparent node.
 3. The node to be deleted is replaced by its inorder successor.
 4. The successor node is appended to the parent of the node just deleted.



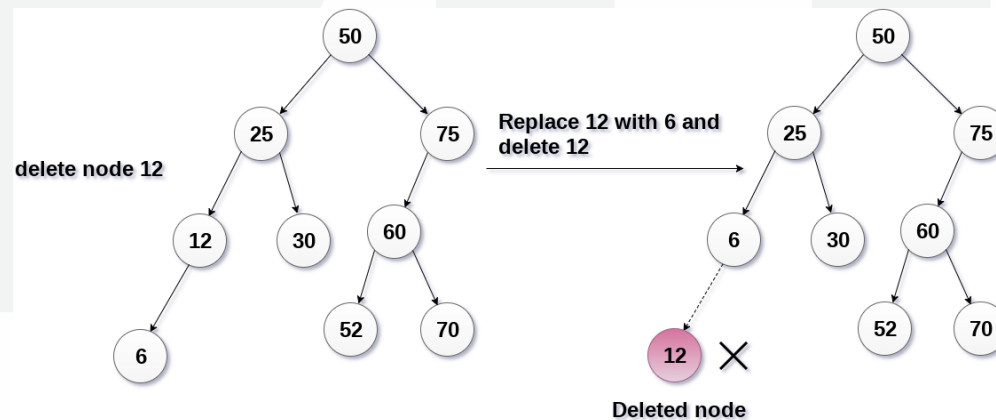
The node to be deleted is a leaf node

- It is the simplest case, in this case, replace the leaf node with the NULL and simply free the allocated space.
- In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



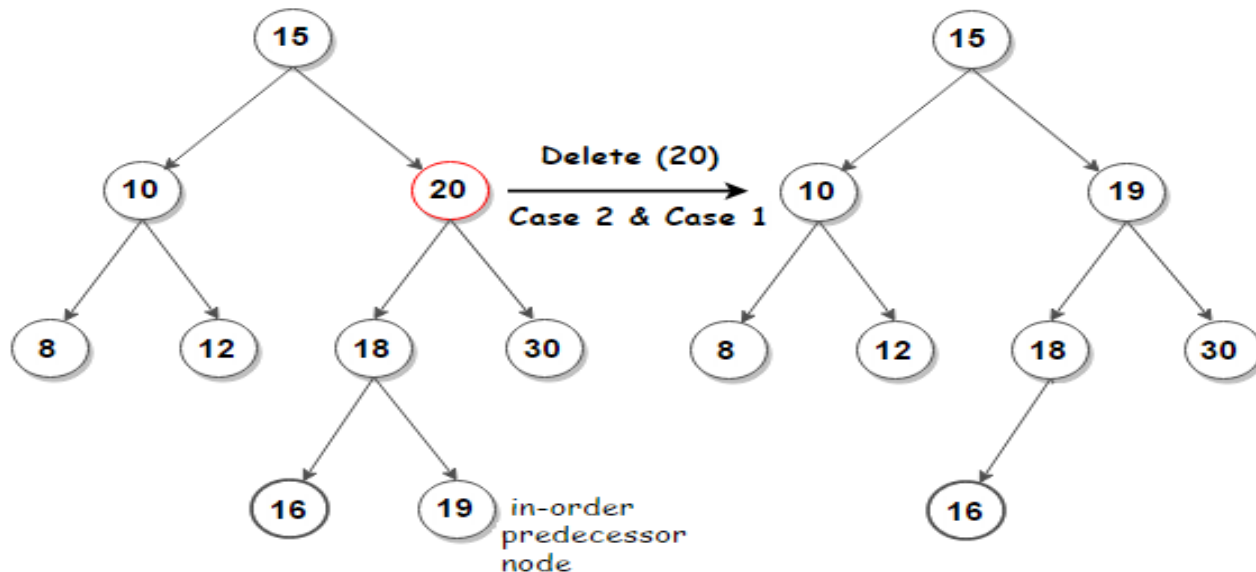
The node to be deleted has only one child.

- In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.
- In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



The node to be deleted has two children.

- It is a bit complex case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.





Search

- Case 1: If the tree is non ordered Binary tree then traversed using any traversal method – Preorder, Inorder, Postorder, converse Preorder, converse inorder, converse Postorder until the appropriate node is found.
- Case 2: If the tree is lexically ordered tree, then the number of comparisons are reduced than the non ordered Binary tree because compare the search element with the root node, if it is lexically greater than the root node then examined only right sided binary subtree. The search continue until either the item I found or an empty subtree is encountered. If the search ends with an empty subtree, then the item is not present in the tree.
- Apply the same method for left subtree, if the search element is lexically less than the root node.



Storage representation of Binary Tree

- There are two methods for representing the Binary tree.
- Linear method and Linked Method
- 1. Linear Method**
 - This method uses one dimensional array of size $2^d + 1 - 1$. where d is the depth of the tree. (Note: Maximum level no. of the tree)
 - Once the size of the array is finalized then use the following steps to represent the Binary tree.
 - First store the root at the first position of array.
 - If the node is at location N of the array then store its left child at $2N$ location and store its right child at $2N+1$ location.



1. Linear Method (Cont)

Advantages:

- It is very simple method.
- From any child, it's parent node can be easily determined. If the child node is at location N then its parent node is at $N/2$ location.
- It can be easily implemented even in older languages like BASIC, FORTRAN etc.

Disadvantages:

- Wastage of memory due to partially filled tree.
- Insertion and deletion of a node required more data movement and it takes excessive amount of processing time.





2. Linked Method

- In a Binary tree, each node may have two child nodes, therefore a node in a linked representation has two pointer fields. One for its right child and another for its left child and one more data field which contains the specific information about node itself.
- When the node has no children then the corresponding pointer field is NULL.

Advantages:

- It is more efficient method than Linear method.

Disadvantages:

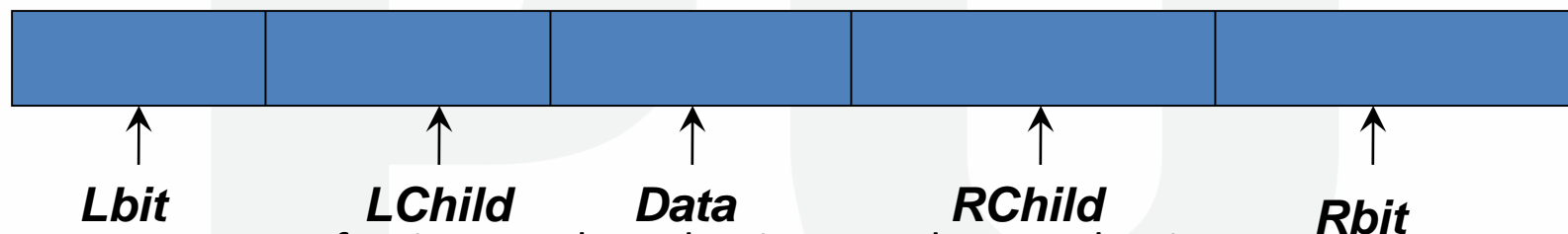
- Wastage of memory space due to NULL pointers.
- From a given node, it is difficult to find its parent node.
- We can't implement Linked method in older languages such as FORTRAN.





Threaded Storage Representation

- In Linked representation method, there are more no. of NULL links than actual pointers. So all the null links in linked storage representation can be replaced by thread pointers.
- Binary tree is threaded according to particular traversal order.
- Storage representation for thread node.



There are two types of pointers: Thread Pointer and Normal Pointer.

$LBIT(P) = 1$, if $Lchild(p)$ is a normal pointer

$= 0$, if $Lchild(p)$ is a thread pointer

$RBIT(P) = 1$, if $Rchild(p)$ is a normal pointer

$= 0$, if $Rchild(p)$ is a thread pointer



Threaded Storage Representation

- If $Rchild(x)$ is NULL then replace NULL link by a pointer to the node which is the inorder successor.
- If $Lchild(x)$ is NULL then replace NULL link by a pointer to the node which is the inorder predecessor.



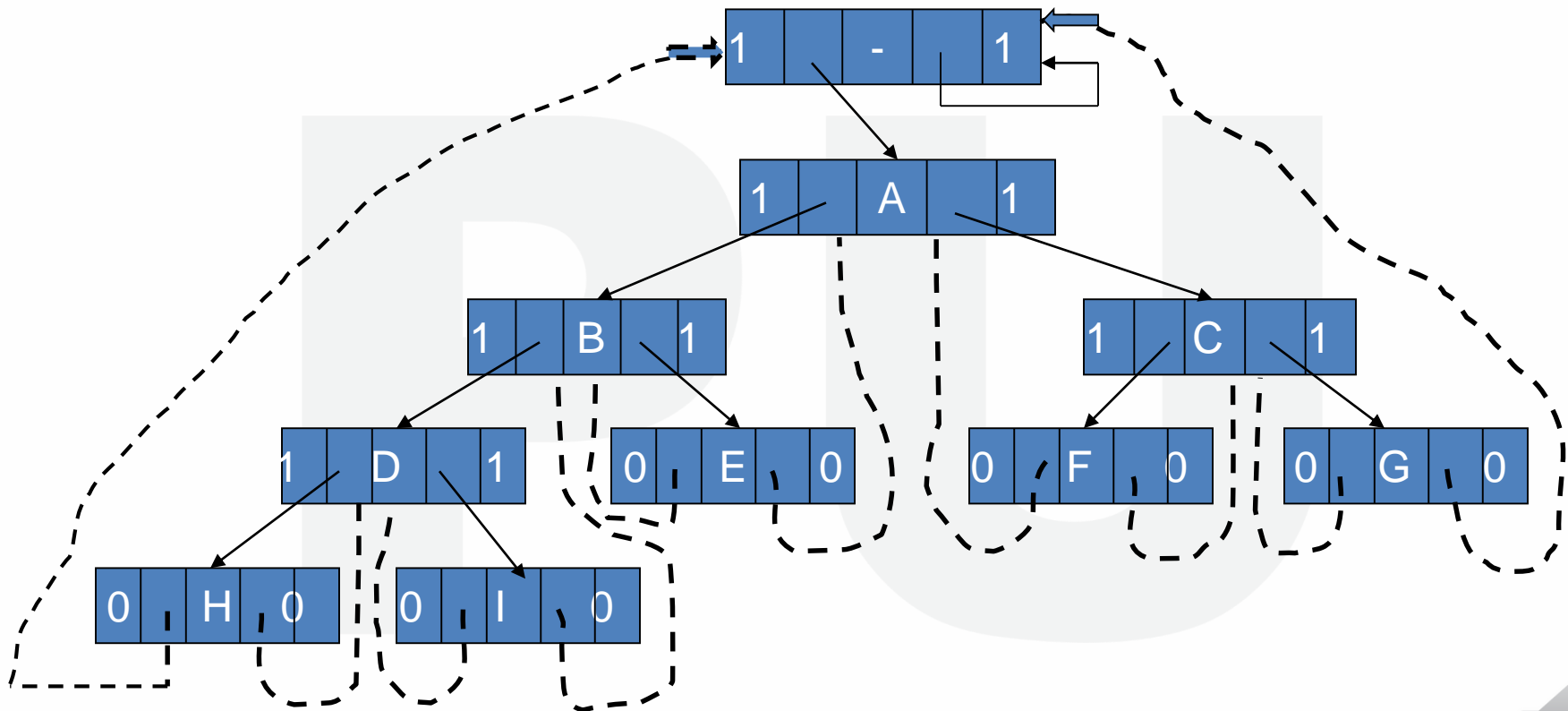
Threaded Storage Representation

For constructing the Threaded Storage Tree follow the following steps:

- When tree is empty then use only HEAD node. It is known as an empty threaded storage representation.
- Form a HEAD node and connect the Rchild with the HEAD node itself.
- Decide the particular traversal order of the given tree and find the predecessor and successor of each node.
- The predecessor of the 1st node and the successor of the last node will be pointing to HEAD node.
- The root node of a given tree is pointed by the LChild of the HEAD node.
- If the left link of any node is NULL then it can be threaded with the address of predecessor of that node for a particular traversal order.
- If the right link of any node is NULL then it can be threaded with the address of successor of that node for a particular traversal order.



Threaded Storage Representation





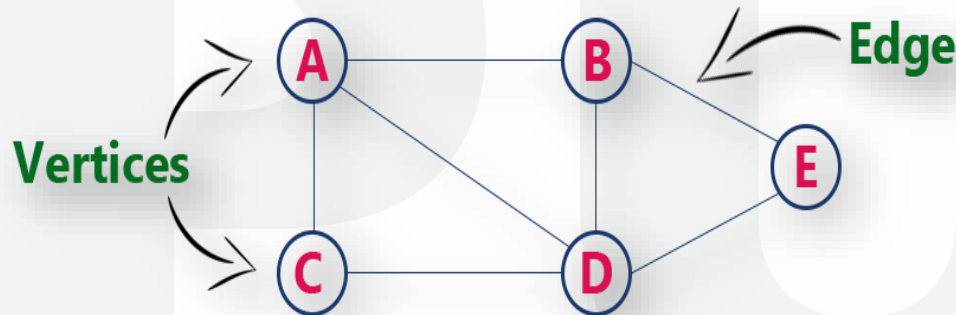
Graph

- Non-Linear Data Structure
- A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.
- Graph is a collection of nodes and edges which connects nodes in the graph
- Generally, a graph G is represented as $G = (V, E)$, where V is set of vertices and E is set of edges.



Graph

- The following is a graph with 5 vertices and 7 edges.
This graph G can be defined as $G = (V, E)$
Where $V = \{A, B, C, D, E\}$ and $E = \{(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (D, E)\}$.

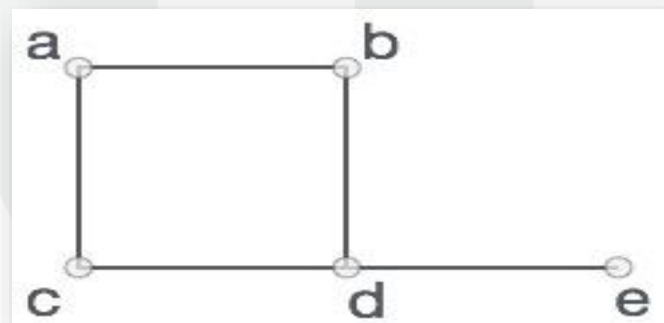


Types of Graph - Undirected Graph

- Undirected graphs have edges that do not have a direction.
- The edges indicate a two-way relationship, in that each edge can be traversed in both directions.
- Formally, a graph is a pair of sets (V, E) , where V is the set of vertices and E is the set of edges, connecting the pairs of vertices.
- Take a look at the following graph –

$V = \{a, b, c, d, e\}$

$E = \{(b, a) (a, c) (d, b) (c, d) (d, e)\}$



Directed Graphs.

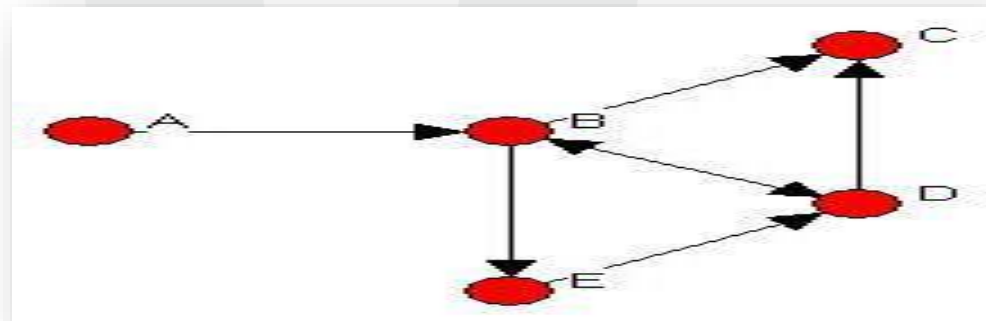
- Directed graphs have edges with direction. The edges indicate a one-way relationship, in that each edge can only be traversed in a single direction.
- For directed graphs we usually use arrows for the arcs between vertices. An arrow from u to v is drawn only if (u,v) is in the Edge set.

The directed graph below Has the following parts.

The Vertices set = $\{A,B,C,D,E\}$

The Edge set = $\{(A,B),(B,C),(D,C),(B,D),(D,B),(E,D),(B,E)\}$

Note that both (B,D) and (D,B) are in the Edge set, so the arc between B and D is an arrow in both directions.



Cyclic Graphs.

- A cyclic graph is a directed graph which contains a path from at least one node back to itself. In simple terms cyclic graphs contain a cycle.
- There exists a path from node B which connects it to itself. The path is $\{(B,C),(C,E),(E,D),(D,B)\}$. Similarly there also exists a path from node D back to itself. The path is $\{(D,B) (B,C) (C,E) (E,D)\}$

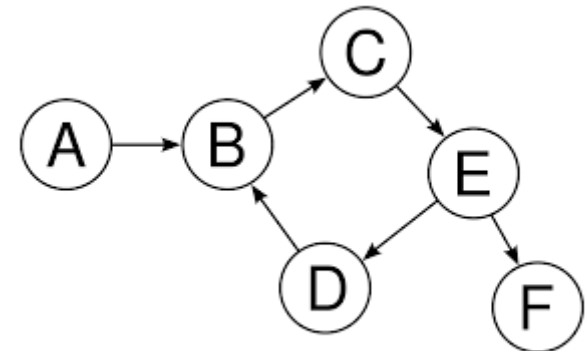
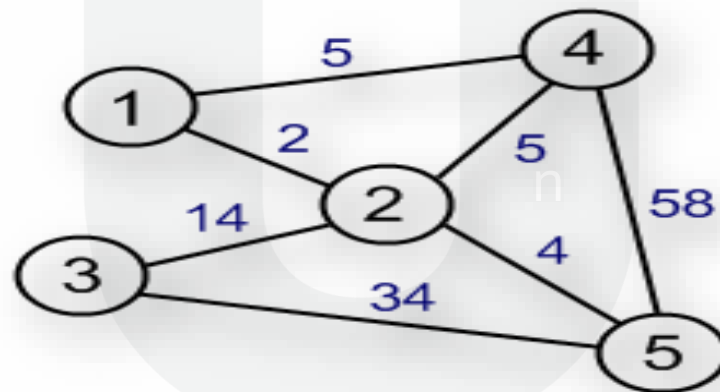


Figure 5 : Cyclic Graph



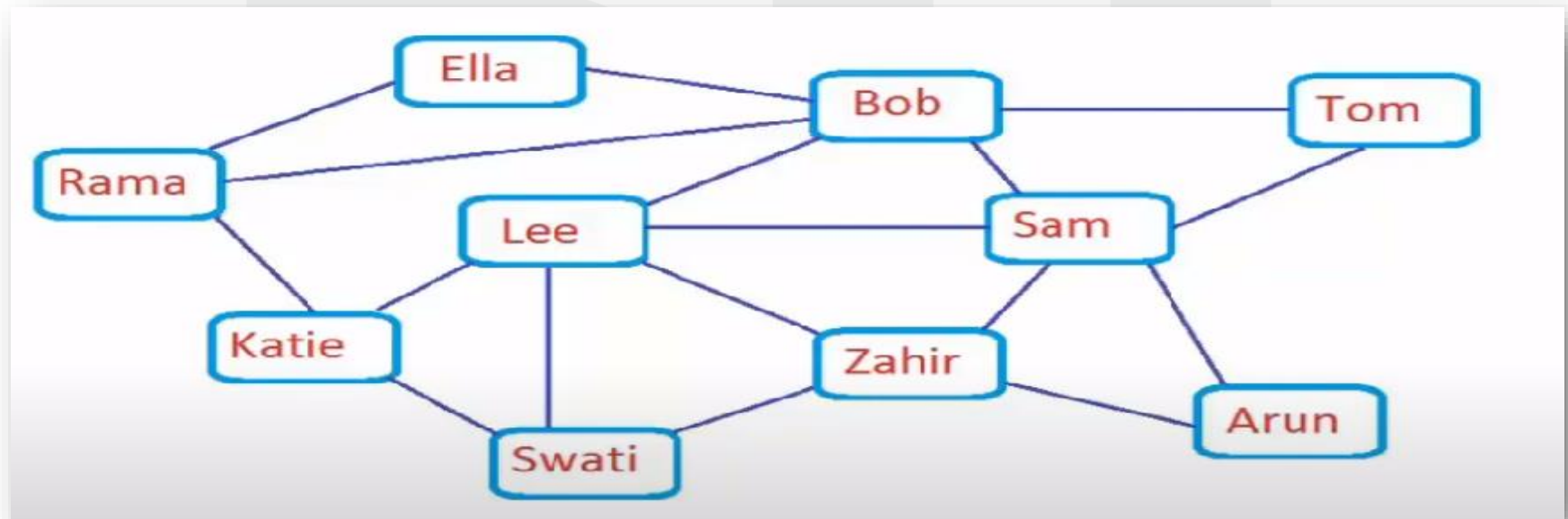
Weighted Graphs.

- A weighted graph is a graph in which each branch is given a numerical weight.
- A weighted graph is therefore a special type of labeled graph in which the labels are numbers (which are usually taken to be positive).



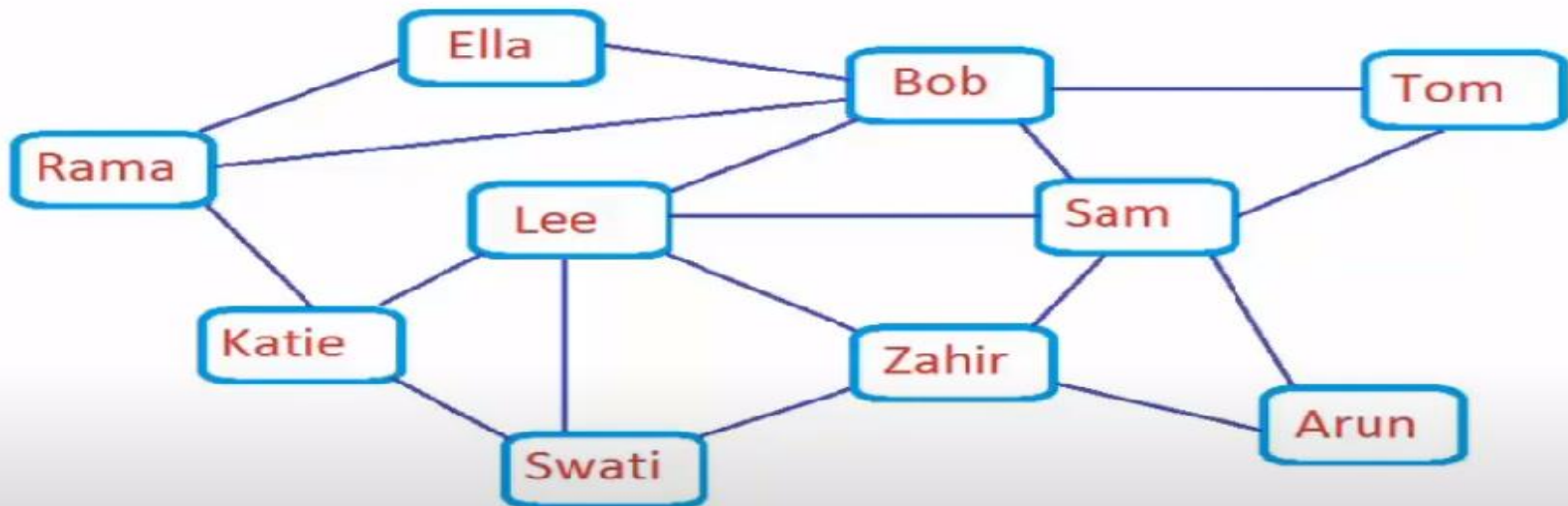
Applications of Graph

- Social Media Network
- Undirected Graph



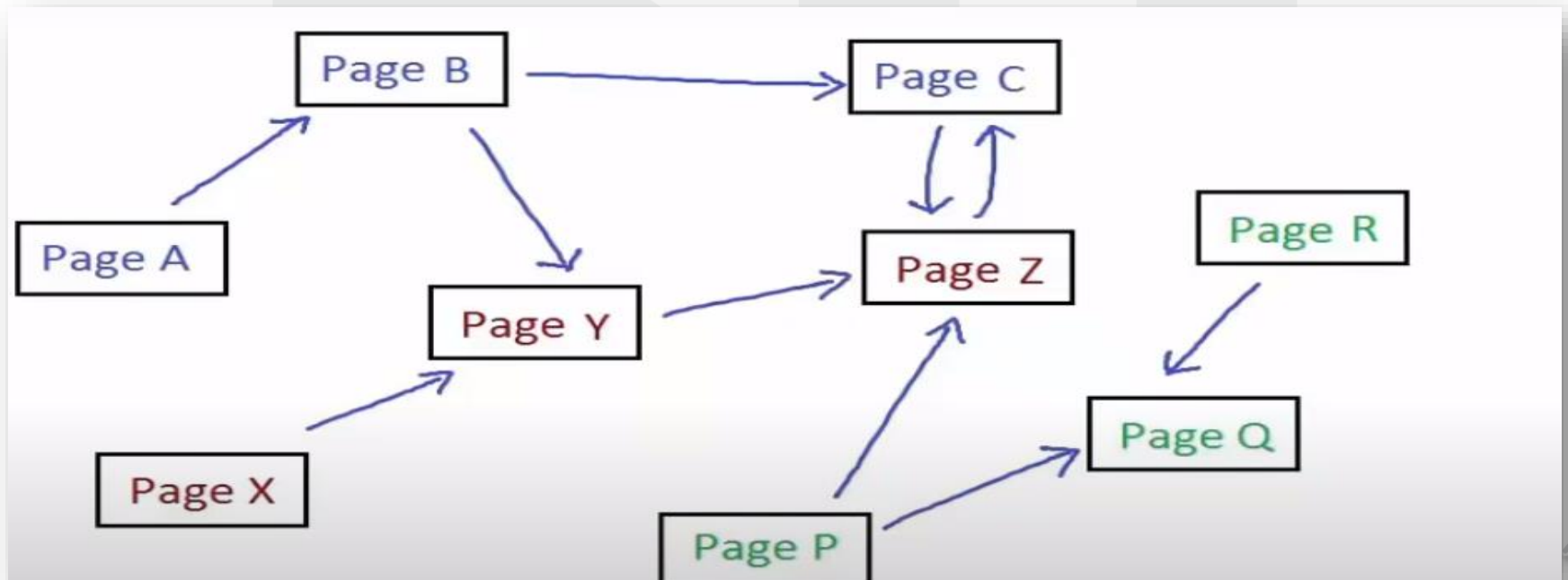
Applications of Graph

- Social Media Network
- Undirected Graph



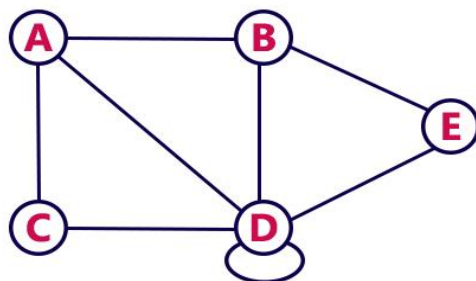
Applications of Graph

- WWW Network
- Directed Graph



Adjacency Matrix


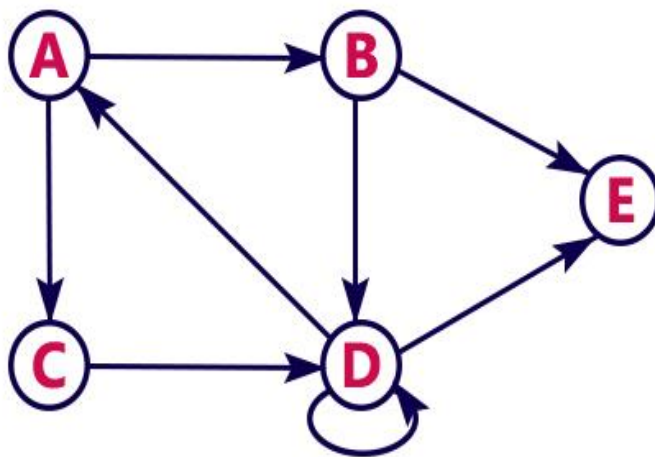
- In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices.
- That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices.
- This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.
- Un-Directed Graph



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

Adjacency Matrix

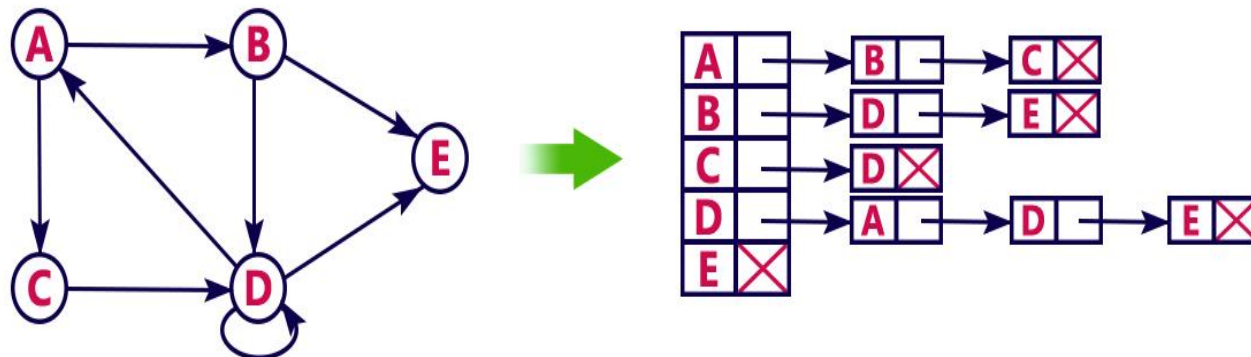
Directed graph representation...



	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

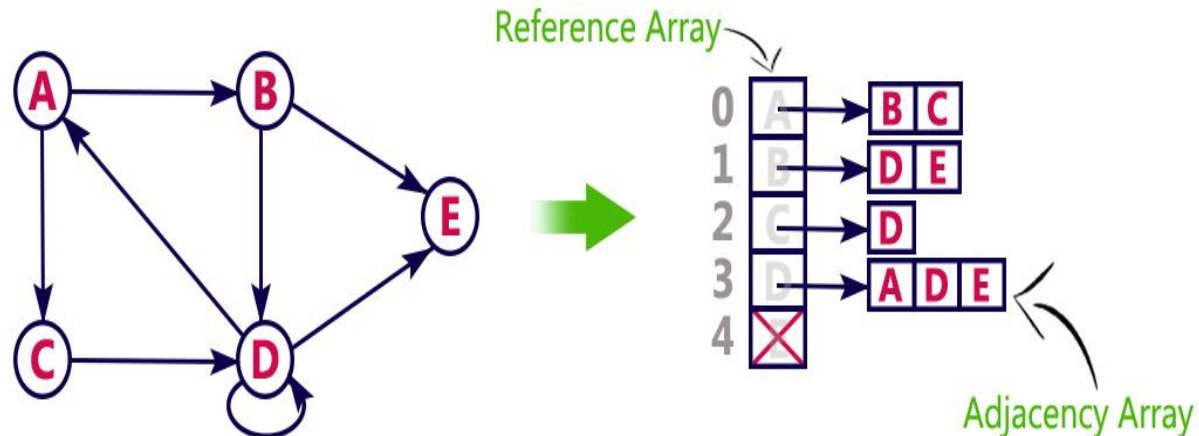
Adjacency List

- In this representation, every vertex of graph contains list of its adjacent vertices. For example, consider the following directed graph representation implemented using linked list...

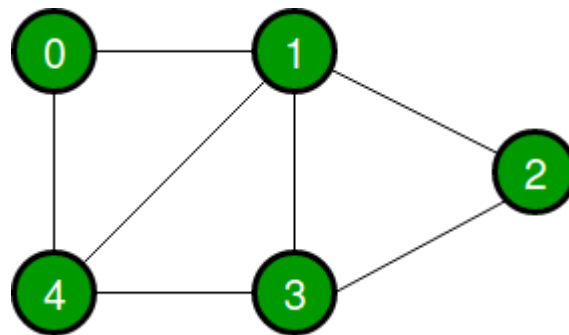


Adjacency List

Representing a graph with adjacency lists combines adjacency matrices with edge lists. For each vertex i , store an array of the vertices adjacent to it. We typically have an array of $|V|$ adjacency lists, one adjacency list per vertex.



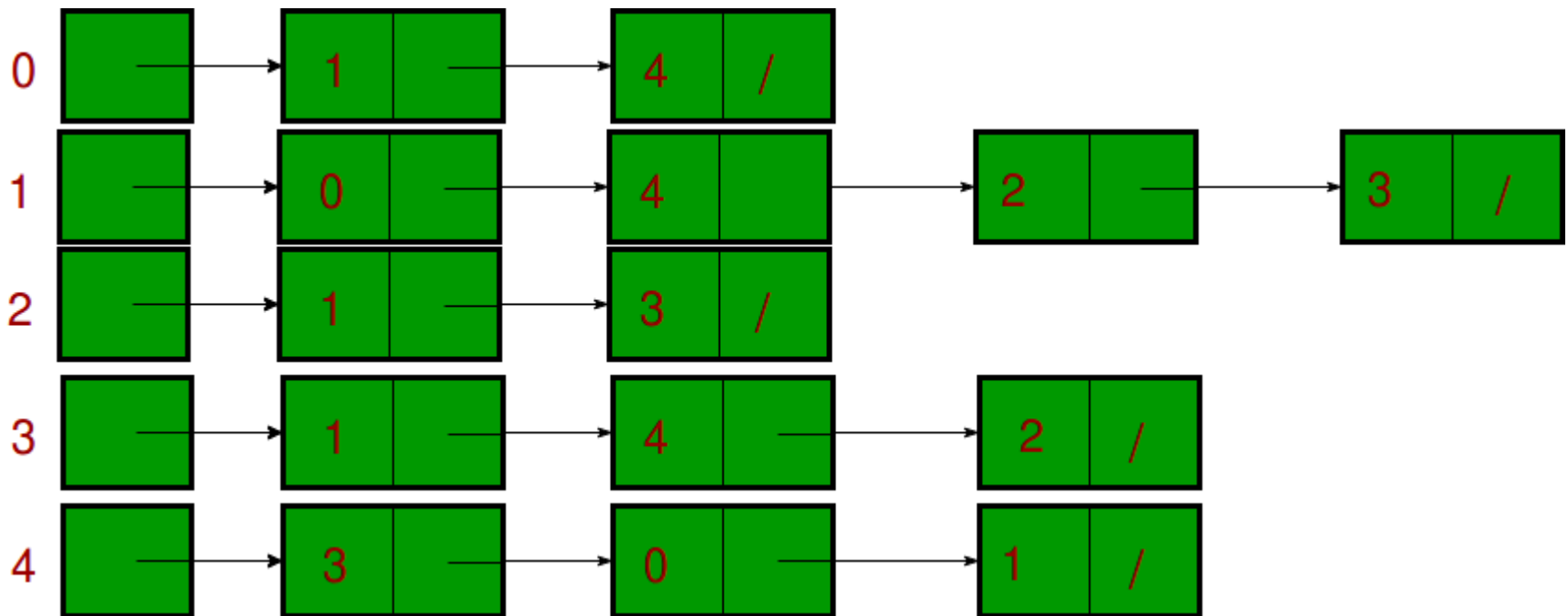
EXAMPLE : 2



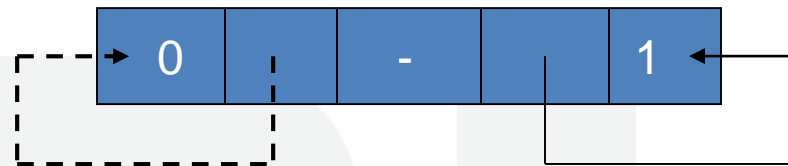
The adjacency matrix for the previous example graph is:

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

Adjacency List:



Threaded Storage Representation



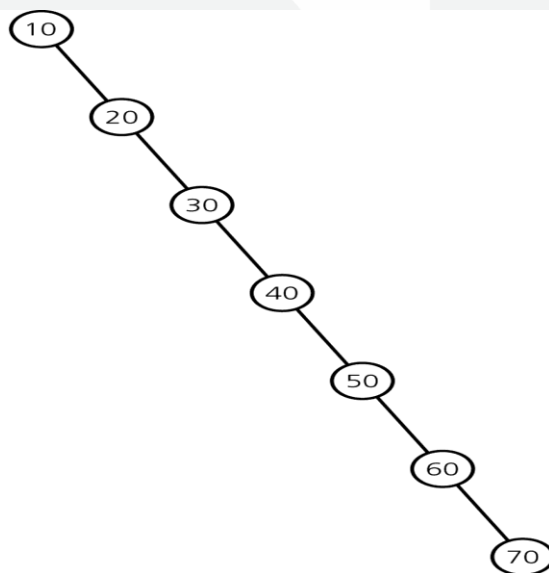
Threaded storage representation of empty Binary tree.



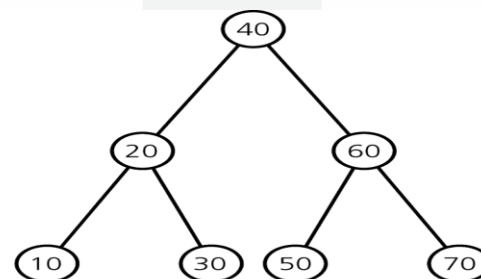
Height Balanced Tree

- To prevent (avoid) skewed (distorted) tree so that Searching becomes Faster
- Height Balanced Trees
 1. AVL Tree
 2. 2-3 Tree

(a)



(b)



Height Balanced Tree - AVL

- Invented by two Russian mathematicians
- G.M. Adelson-Velskii and E/Y.M. Landis (AVL)
- In 1962
- It is a binary tree in which difference of height of two sub-trees with respect to a node never differ by more than one (1).
- Balance factor of a node
- $\text{height}(\text{left sub tree}) - \text{height}(\text{right sub tree})$
- Difference of height must be -1, 0, 1



AVL Tree

In AVL Tree, we use concept of Balance Indicator (BI) for each node

1. Left : A node is called Left Heavy

- If the longest path in its left sub-tree is one longer than the longest path of its right sub-tree.

2. Right : A node is called Right Heavy

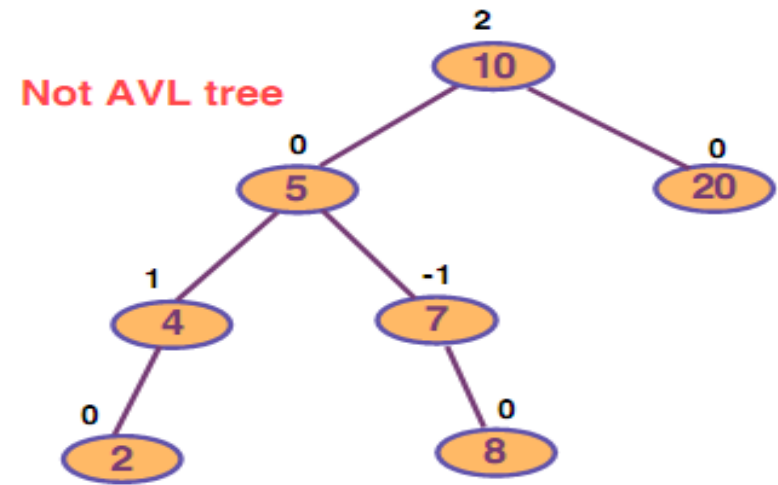
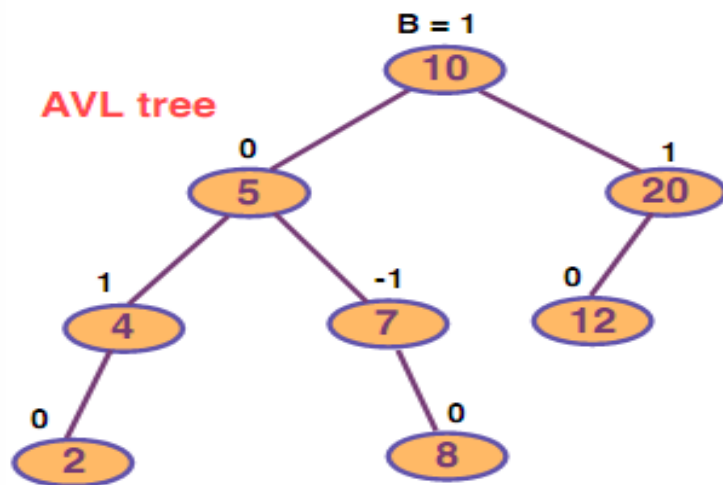
- If the longest path in its right sub-tree is one longer than the longest path of its left sub-tree.

3. Balanced : A node is called Balanced

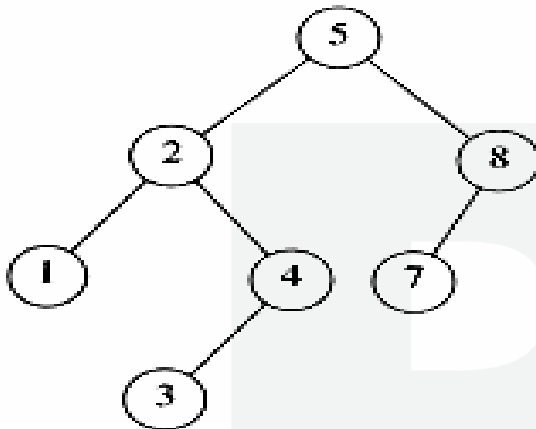
- If the longest path in its left sub-tree is equal to the longest path of its right sub-tree.



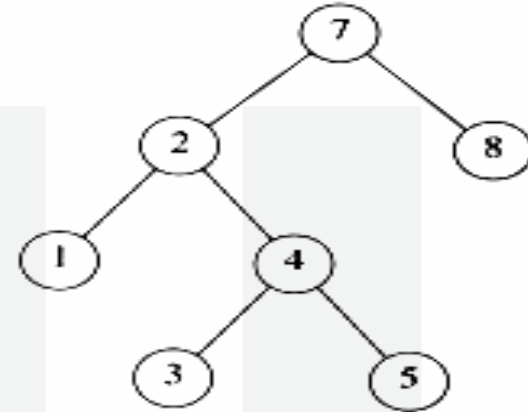
AVL Example



AVL tree?



YES



NO

Left sub-tree has height 3, but right sub-tree has height 1 at node No.7
(Diff = 2)

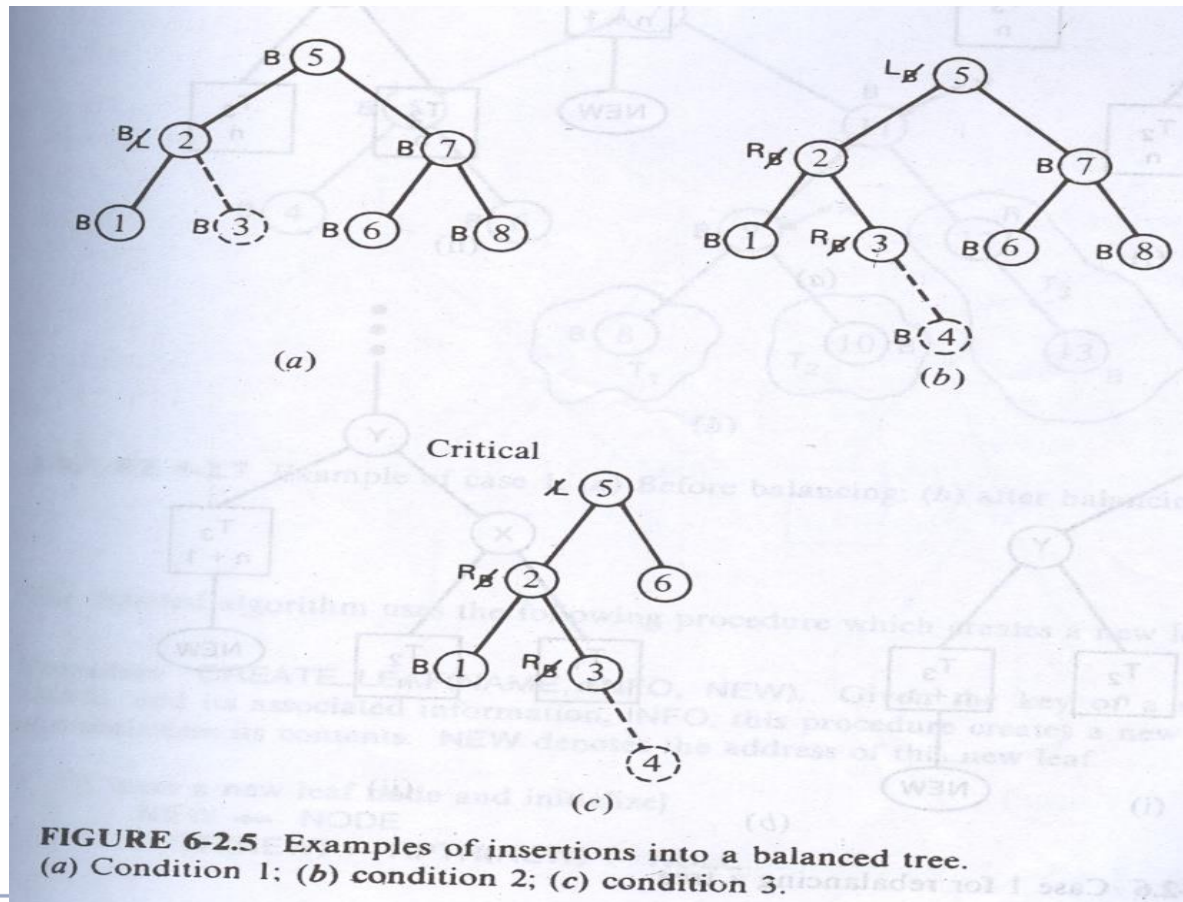


Insert in AVL

- Insertion affects BI of all nodes which are on path from Insertion point to Root Node.
- Possible changes at each node
 1. Left/Right Heavy ? becomes Balanced
 2. Balanced ? becomes Left/Right Heavy
 3. Heavy ? becomes Unbalanced



AVL tree



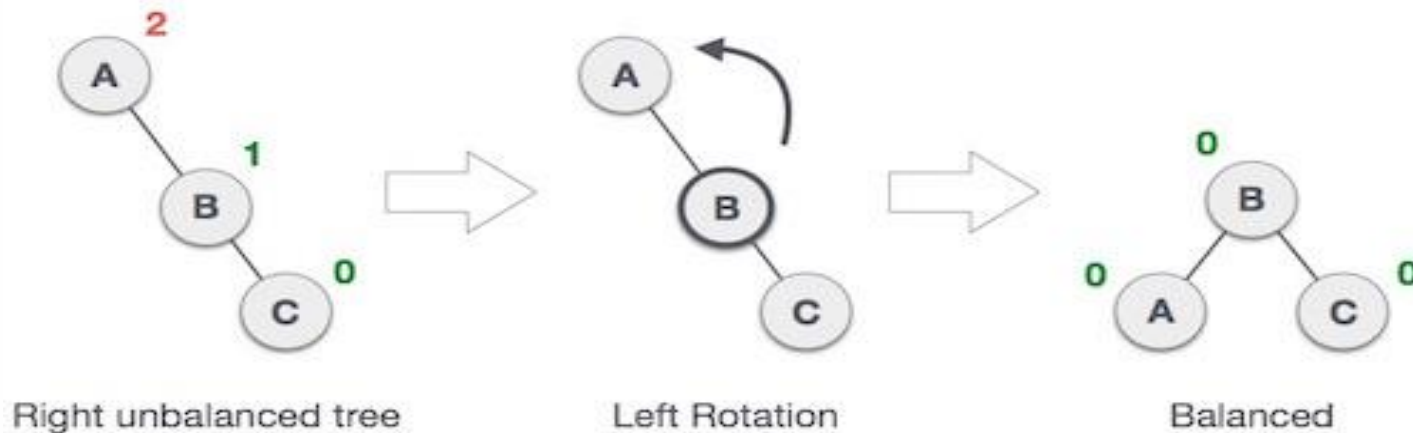
Balancing

- Four types of rotations (corresponding to the four cases of unbalanced trees)
 1. Single Left (L) Rotation: Right of Right
 2. Single Right (R) Rotation: Left of Left
 3. Double Right-Left (RL) Rotation: Left of Right
 4. Double Left-Right (LR) Rotation: Right of Left



Left Rotation

- If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation

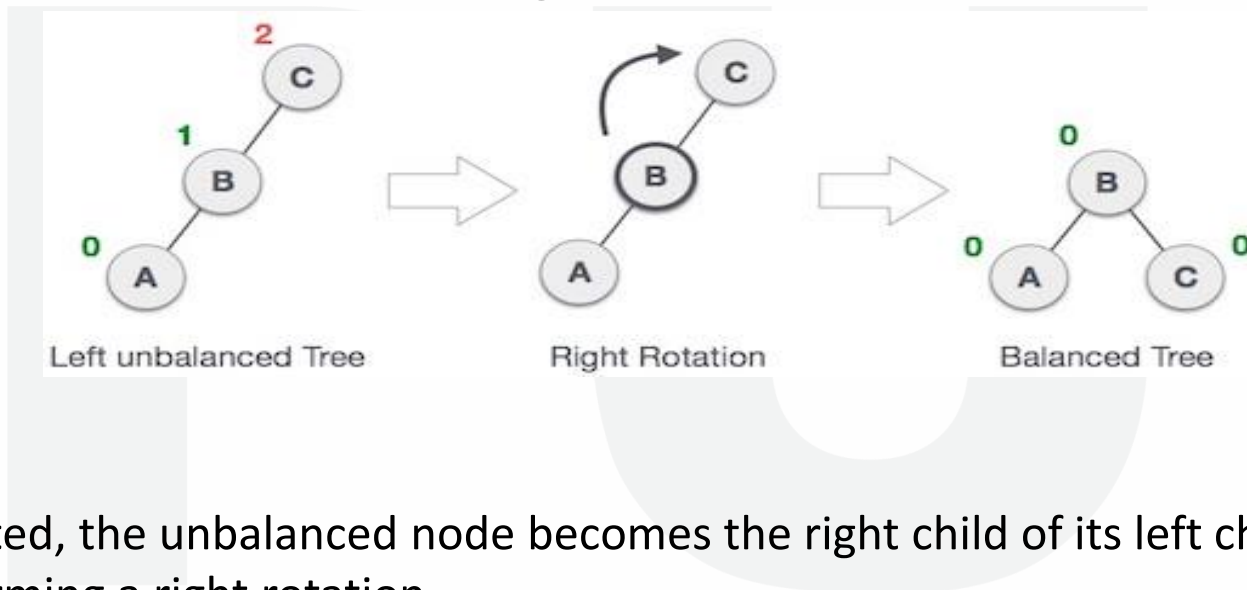


- In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.



Right Rotation

- AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



- As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.



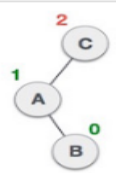
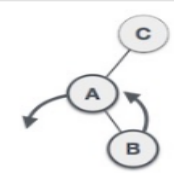
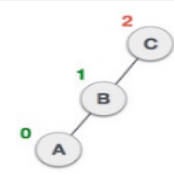
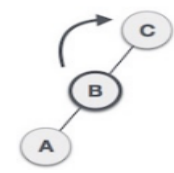
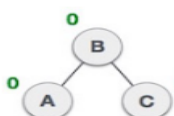


Left-Right Rotation

- Double rotations are slightly complex version of already explained versions of rotations.
- To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.



Left-Right Rotation

State	Action
 <p>A diagram of an AVL tree with root node C (balance factor 2). Node C has a left child A (balance factor 1) and a right child B (balance factor 0). Node A has a left child (balance factor 0).</p>	<p>A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.</p>
 <p>A diagram showing a left rotation around node C. Node A is being moved to become the left child of node B. Arrows indicate the rotation path.</p>	<p>We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.</p>
 <p>A diagram of the AVL tree after the first rotation. Node C is still the root (balance factor 2). Node C has a left child B (balance factor 1) and a right child (balance factor 0). Node B has a left child A (balance factor 0).</p>	<p>Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.</p>
 <p>A diagram showing a right rotation around node C. Node B is being moved to become the new root of the subtree. Arrows indicate the rotation path.</p>	<p>We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.</p>
 <p>A diagram of the final balanced AVL tree. Node B is the root (balance factor 0). Node B has a left child A (balance factor 0) and a right child C (balance factor 0).</p>	<p>The tree is now balanced.</p>





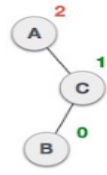
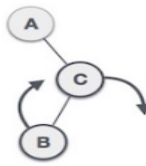
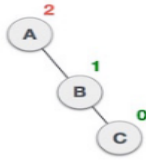
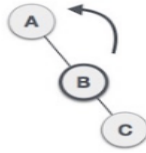
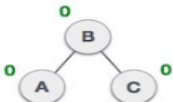
Right-Left Rotation

- The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

PU



Right-Left Rotation

 <p>A tree with root A (balance factor 2). A has a left child B (balance factor 0) and a right child C (balance factor 1).</p>	<p>A node has been inserted into the left subtree of the right subtree. This makes A, an unbalanced node with balance factor 2.</p>
 <p>Diagram showing a right rotation around node C. Node C moves down to become the right child of node B. Node A remains the root.</p>	<p>First, we perform the right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.</p>
 <p>A tree with root A (balance factor 2). A has a left child B (balance factor 1) and a right child C (balance factor 0).</p>	<p>Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.</p>
 <p>Diagram showing a left rotation around node A. Node A moves down to become the left child of node B. Node C remains the right child of node B.</p>	<p>A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.</p>
 <p>A balanced tree with root B (balance factor 0). B has a left child A (balance factor 0) and a right child C (balance factor 0).</p>	<p>The tree is now balanced.</p>



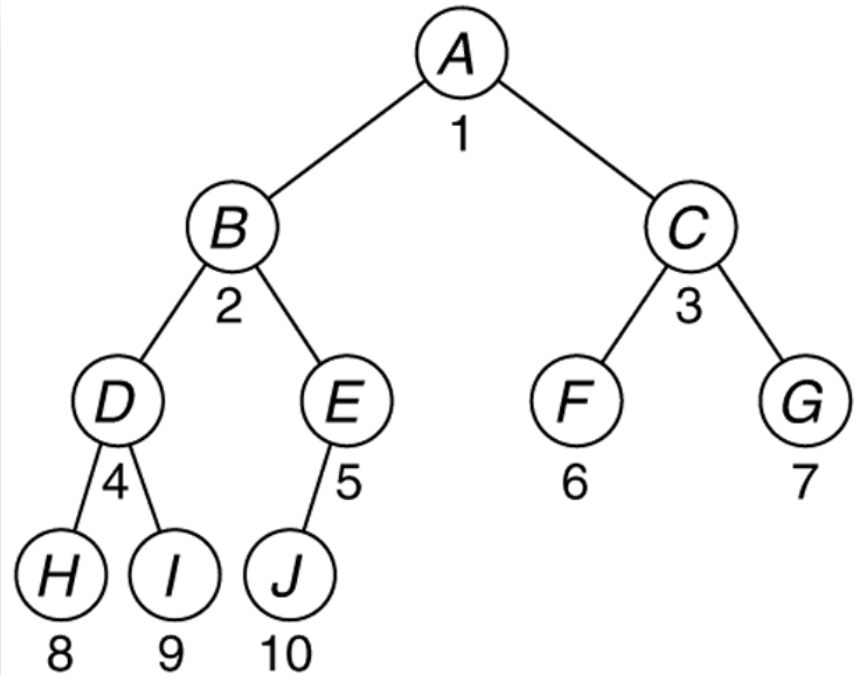
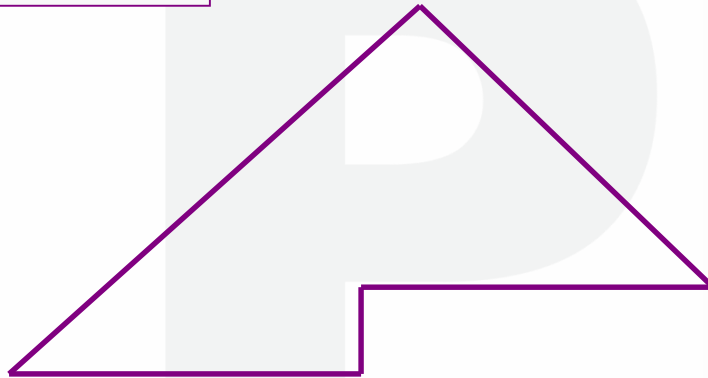
Heap

- Heaps are binary trees that are ordered from bottom to top, so that a traversal along any leaf-to-root path will visit the keys
 - in ascending order: max heaps : $\text{key}(\text{parent}) \geq \text{key}(\text{child})$
 - in descending order: min heaps : $\text{key}(\text{parent}) \leq \text{key}(\text{child})$
- Heaps are
 - (1) to implement priority queues,
 - (2) to implement the Heap sort algorithm.
- The parent of node number i is numbered $(i/2)$ and its two children are numbered $2i$ and $(2i + 1)$.
- A heap is a complete binary tree.



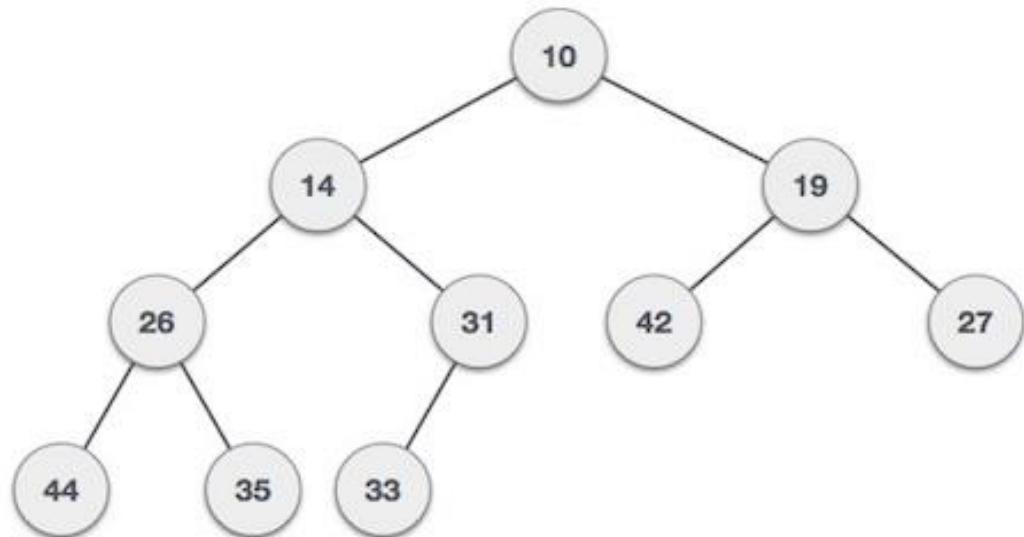
Heap

Heap shape:



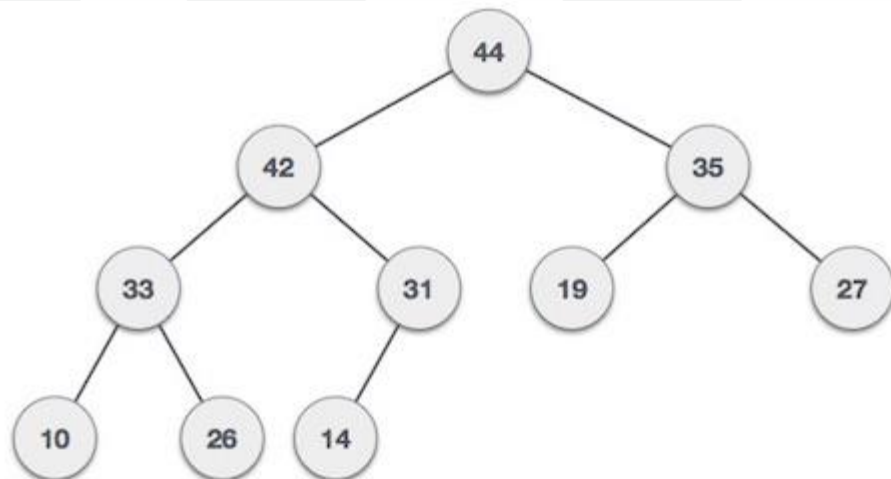
Min Heap

- A minimal heap (descending heap) is an almost complete binary tree in which the value at each parent node is less than or equal to the values in its child nodes.
- Obviously, the minimum value is in the root node.
- Note, too, that any path from a leaf to the root passes through the data in descending order.
- Here is an example
- of a minimal heap:



Max Heap

- A Max heap (Ascending heap) is an almost complete binary tree in which the value at each parent node is greater than or equal to the values in its child nodes.
- Obviously, the maximum value is in the root node.
- Note, too, that any path from a leaf to the root passes through the data in Ascending order. Here is an example :



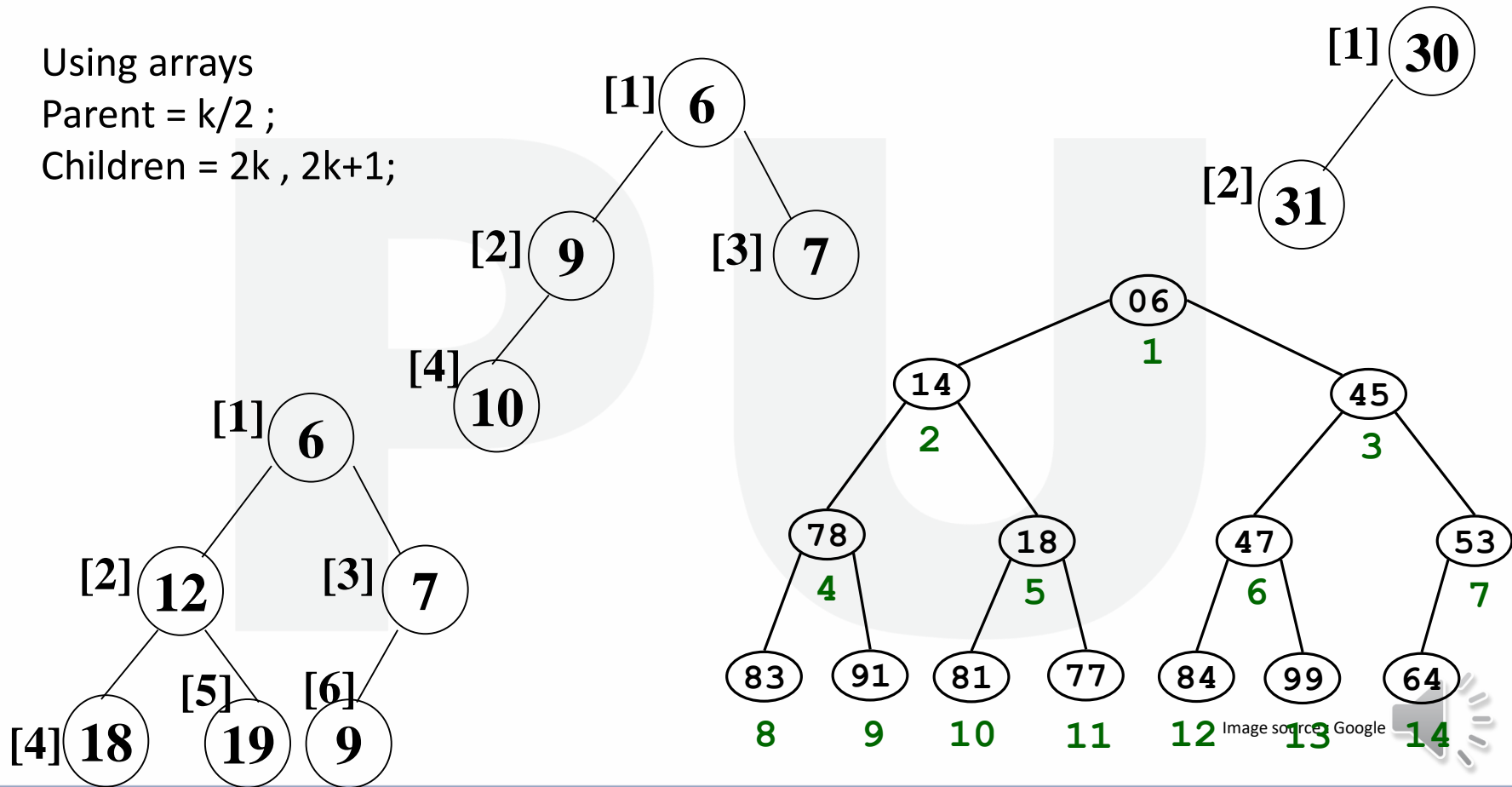
Heap Implementation

- The typical storage method for a heap, or any almost complete binary tree, works as follows.
- Begin by numbering the nodes level by level from the top down, left to right.
- when implementing a complete binary tree, we actually can "cheat" and just use an array
- index of root = 1 (leave 0 empty for simplicity)
- for any node n at index i,
- index of n.left = $2i$
- index of n.right = $2i + 1$
- For example, consider the following heap. The numbering has been added below the nodes.



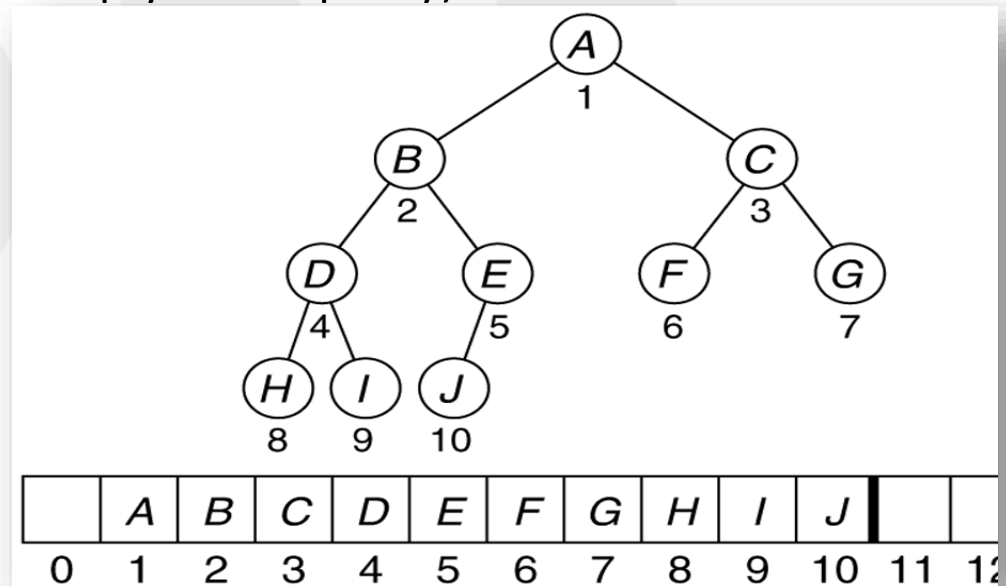
Heap Implementation

- Using arrays
- Parent = $k/2$;
- Children = $2k, 2k+1$;



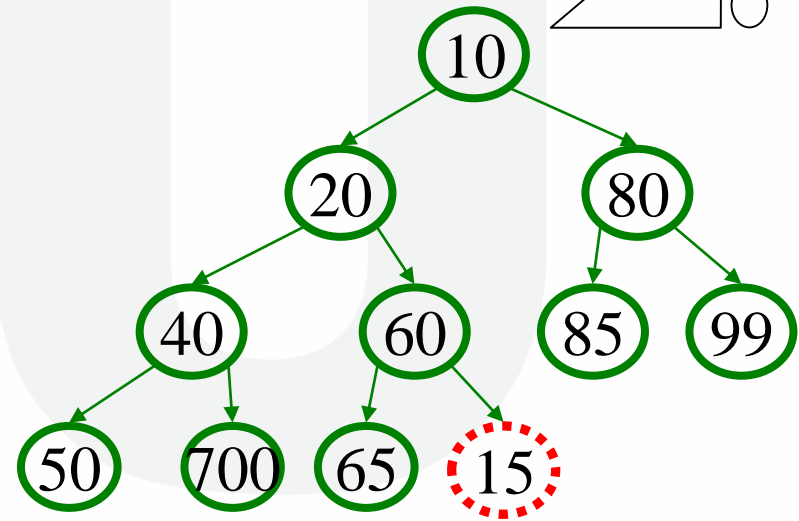
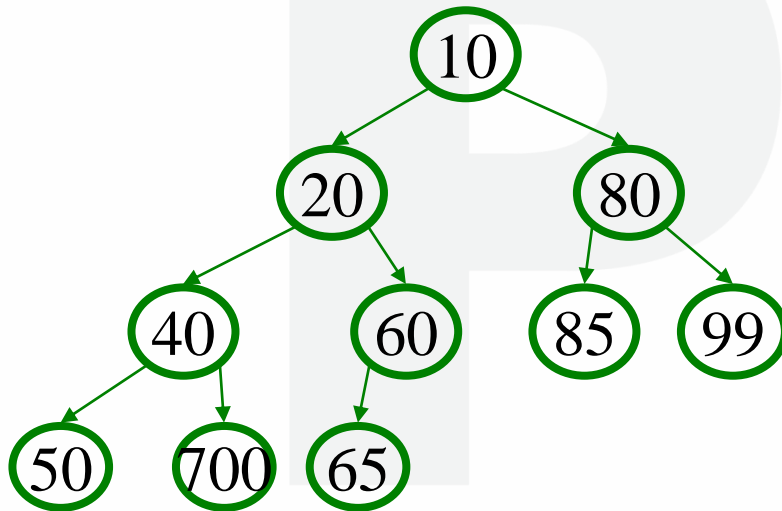
Heap Implementation

- when implementing a complete binary tree, we actually can "cheat" and just use an array
- index of root = 1 (leave 0 empty for simplicity)
- for any node n at index i,
- index of n.left = $2i$
- index of n.right = $2i + 1$



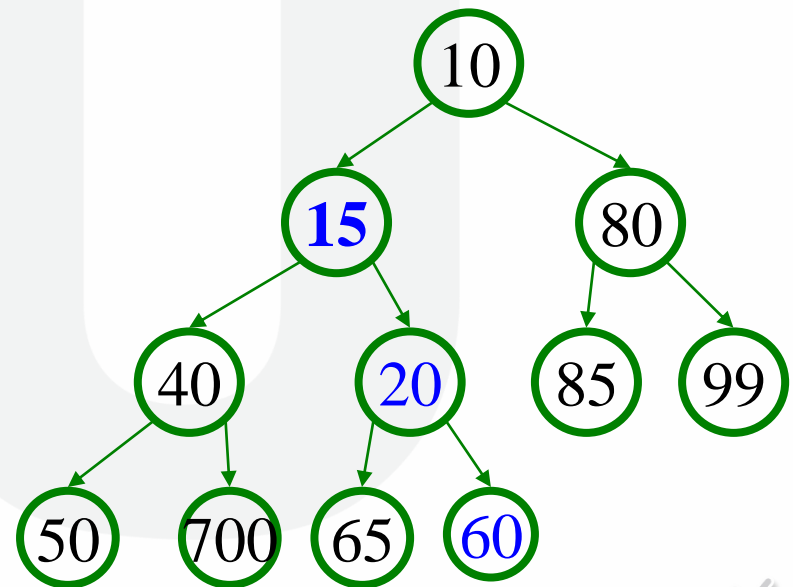
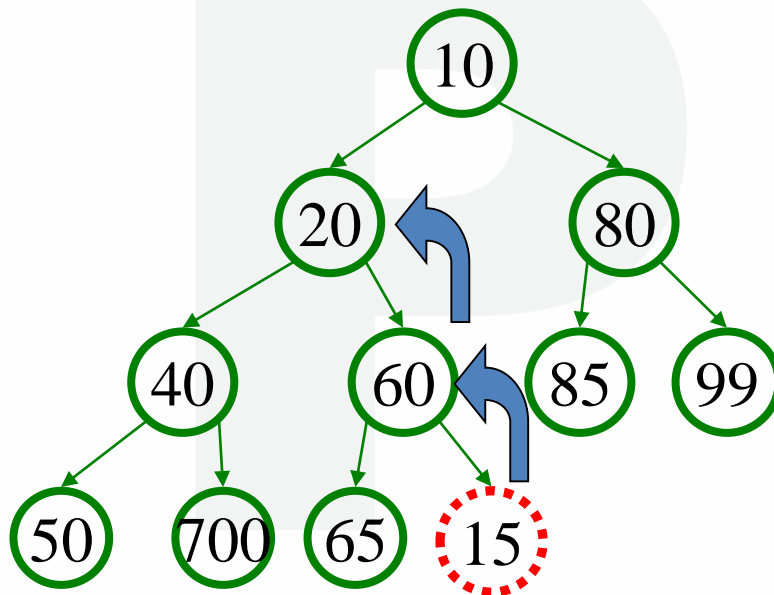
Adding to a heap

- when an element is added to a heap, it should be initially placed as the rightmost leaf (to maintain the completeness property)
- heap ordering property becomes broken!



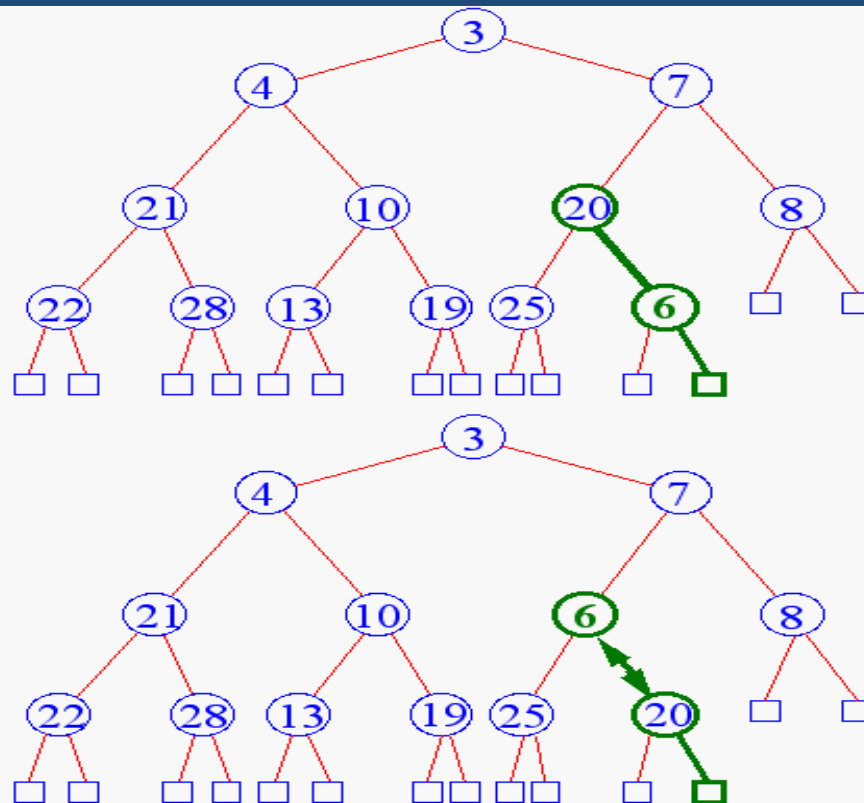
Adding to a min heap

- To restore heap ordering property, the newly added element must be shifted upward ("bubbled up") until it reaches its proper place
- bubble up by swapping with parent

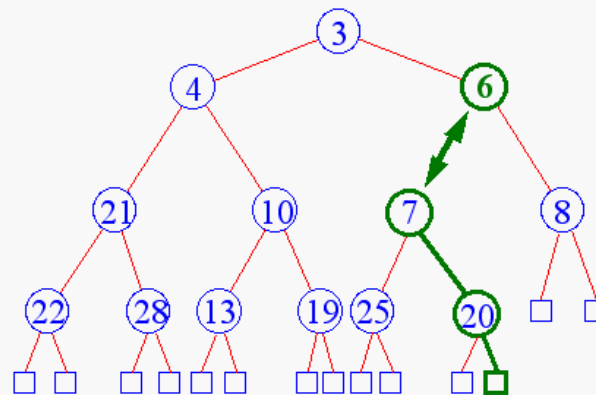
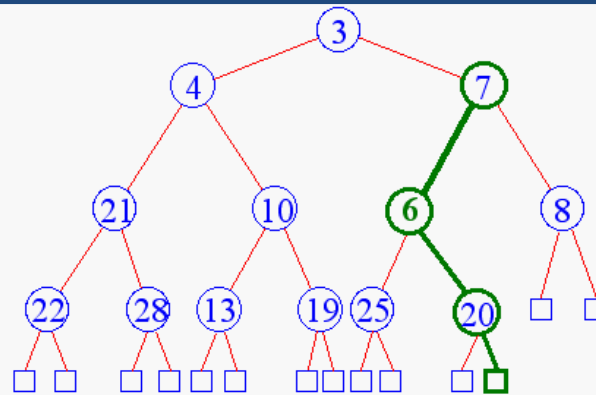


Adding to a min heap

- Begin Unheap

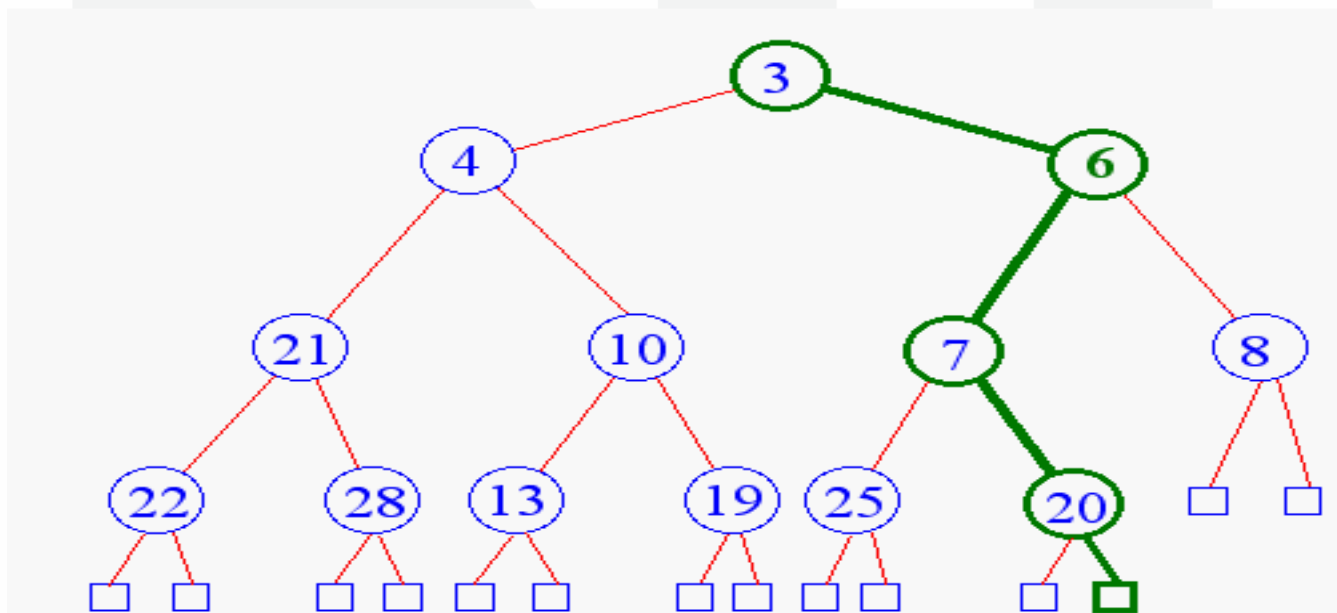


Adding to a min heap



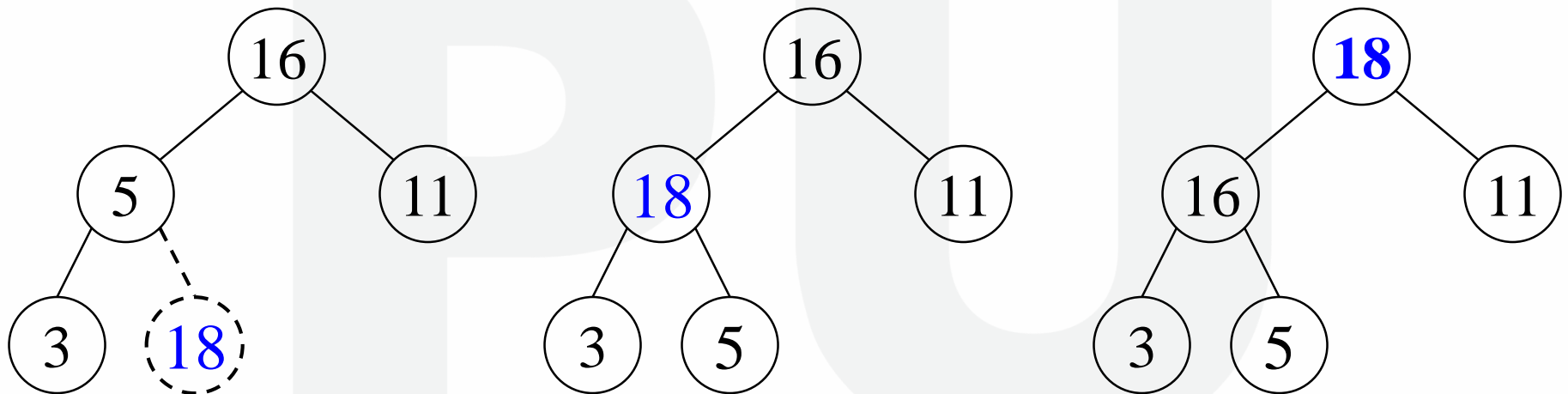
Adding to a min heap

- Terminate unheap when
- reach root
- key child is greater than key parent



Adding to a max-heap

- same operations, but must bubble up larger values to top



Breadth-First Search

- Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes.
- Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.
- BFS - > Queue



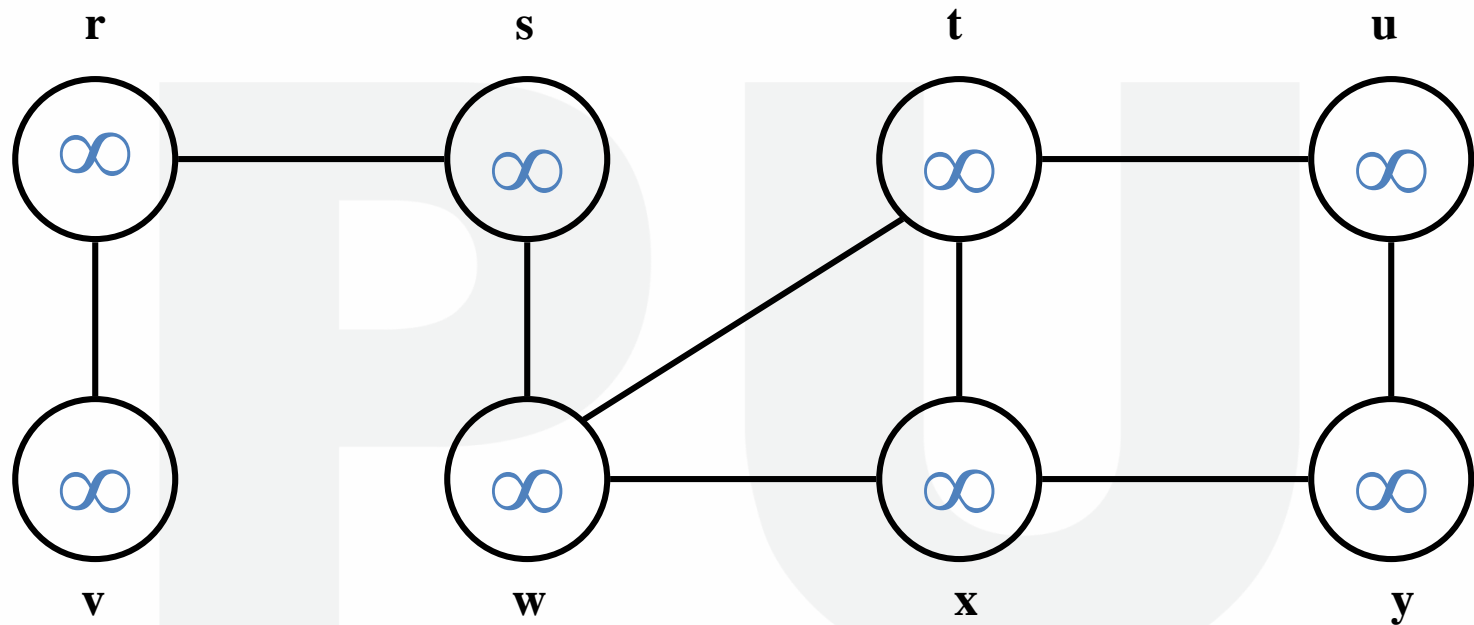


Breadth-First Search

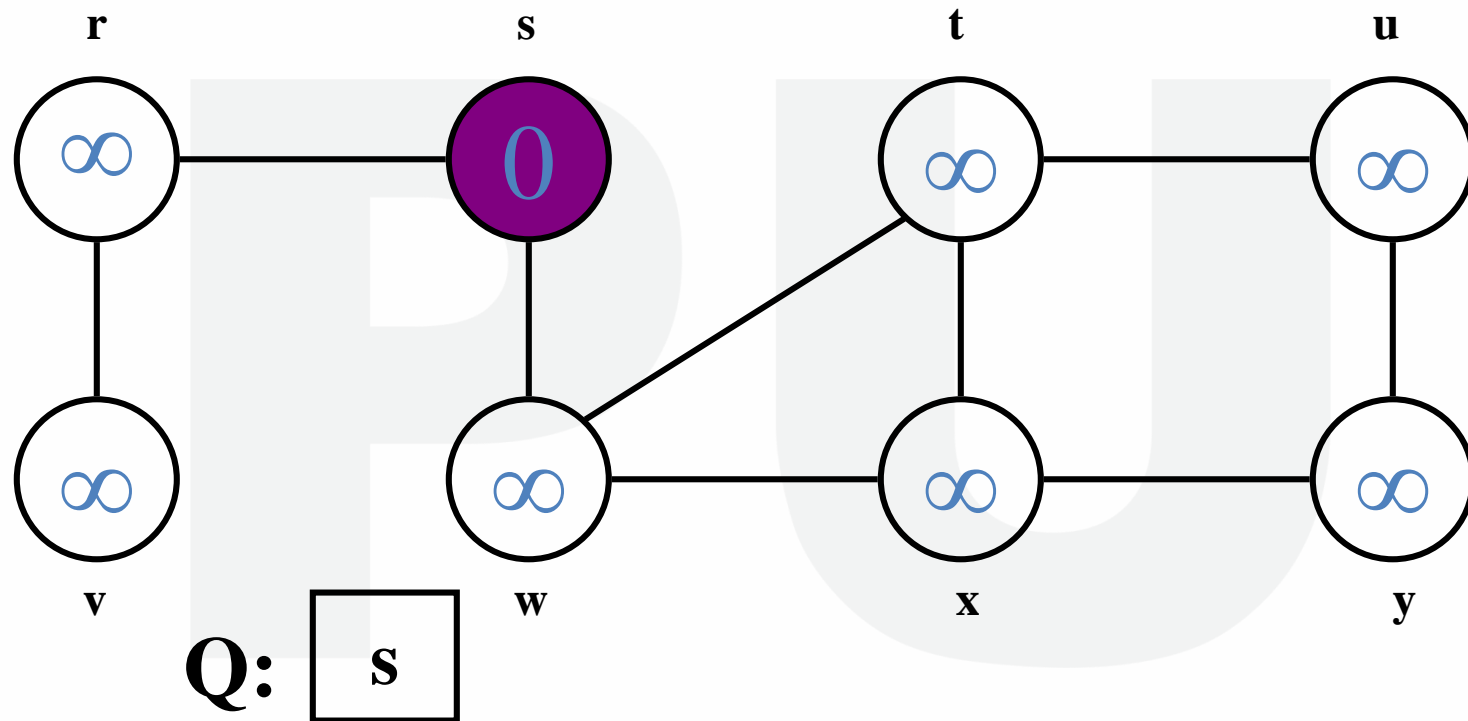
- Again will associate vertex “colors” to guide the algorithm
 - White vertices have not been discovered
 - All vertices start out white
 - Grey vertices are discovered but not fully explored
 - They may be adjacent to white vertices
 - Black vertices are discovered and fully explored
 - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices



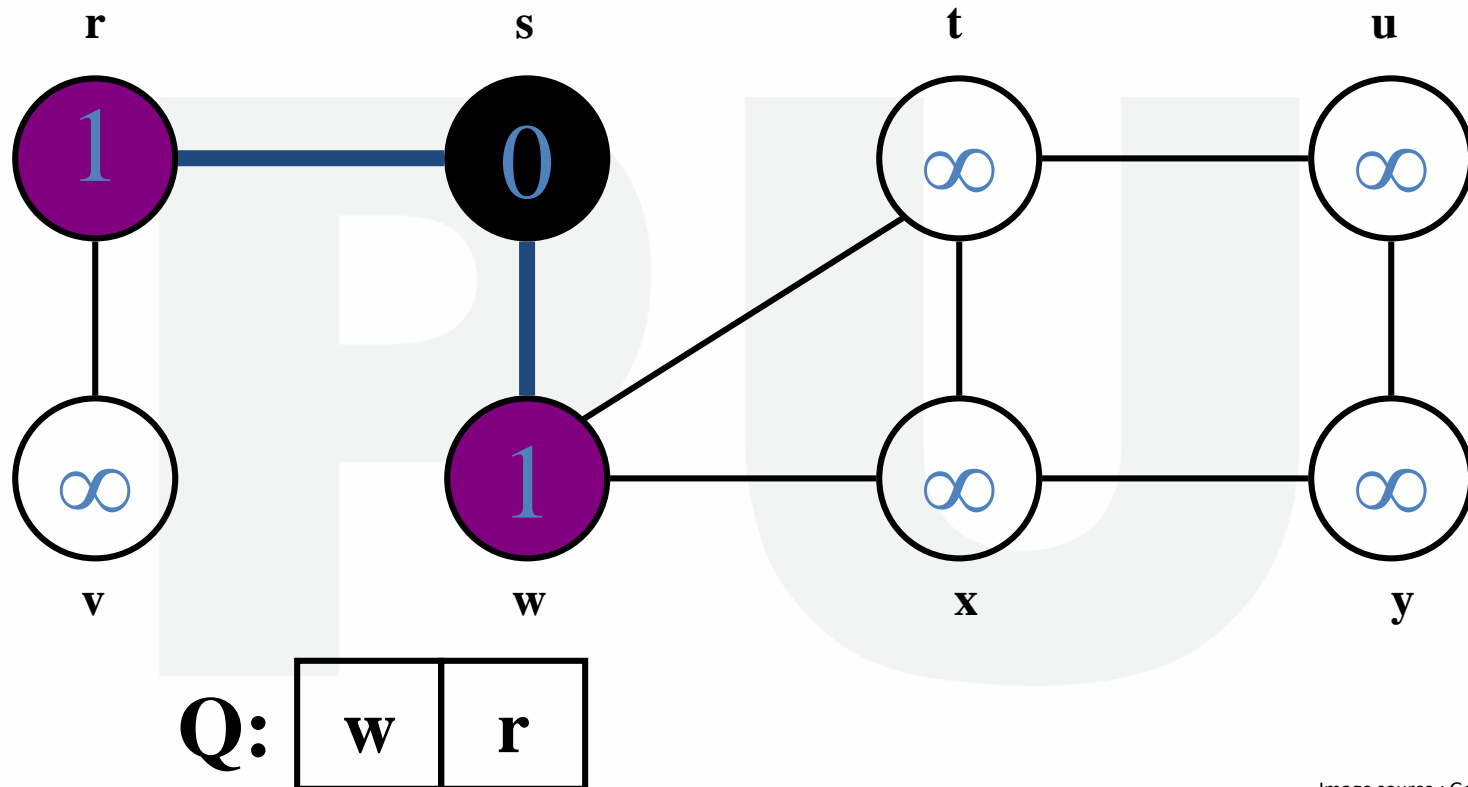
Breadth-First Search: Example



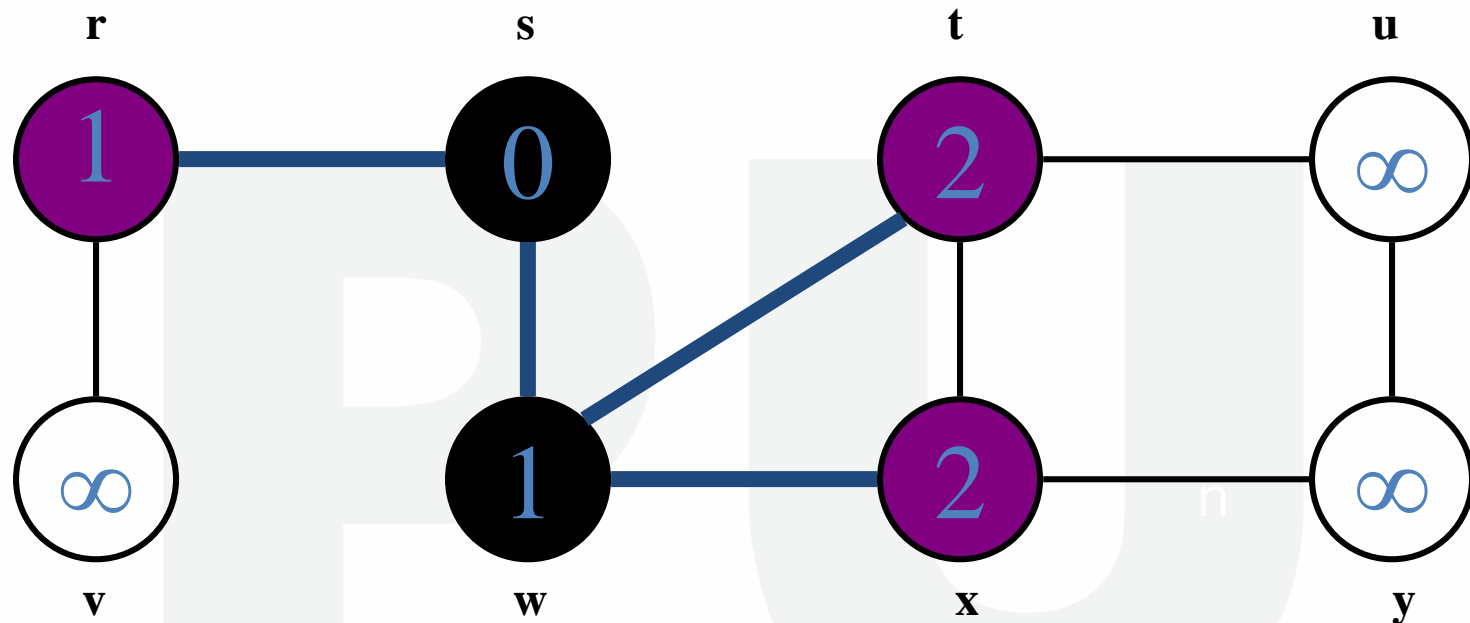
Breadth-First Search: Example



Breadth-First Search: Example



Breadth-First Search: Example

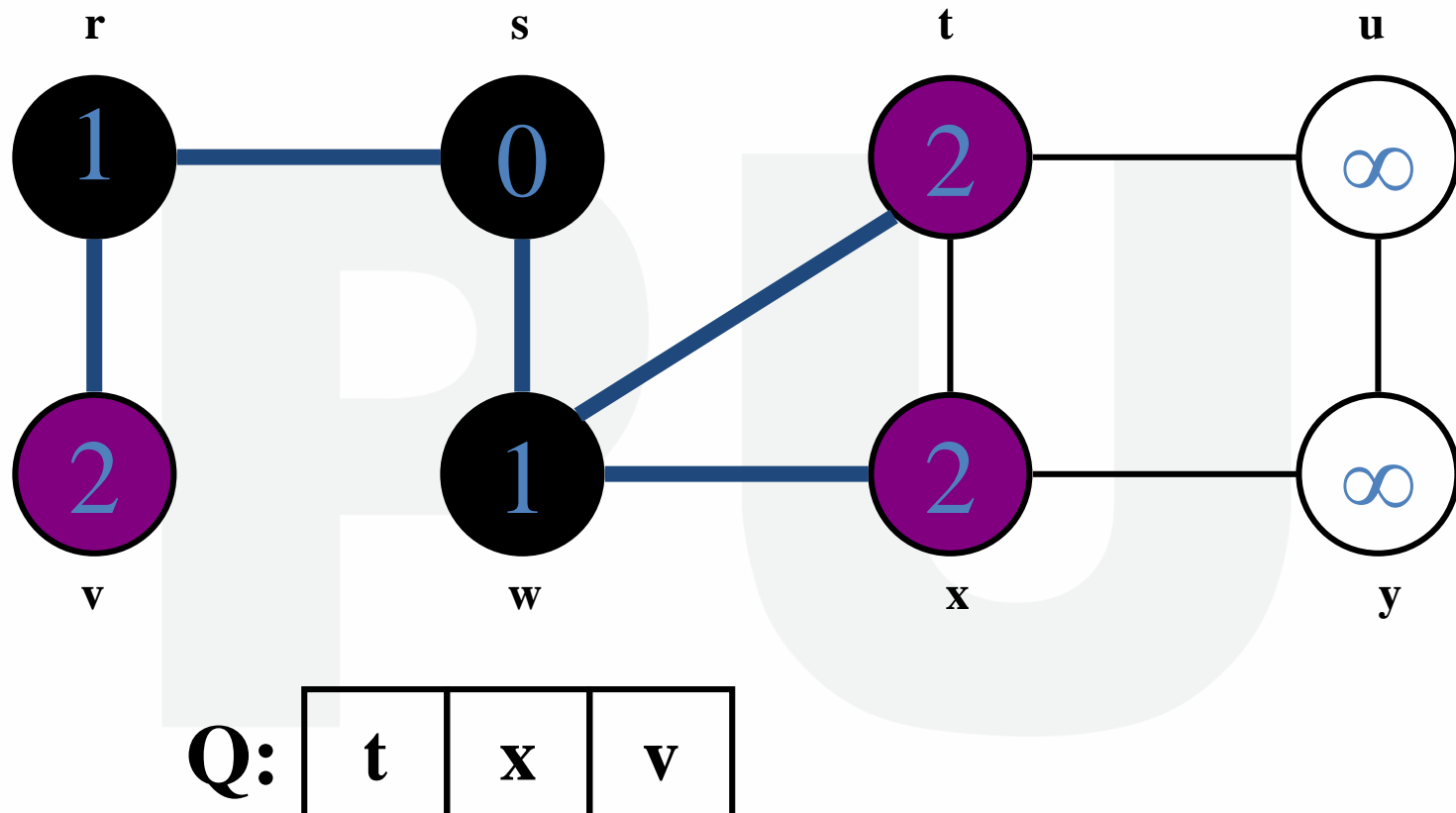


Q:

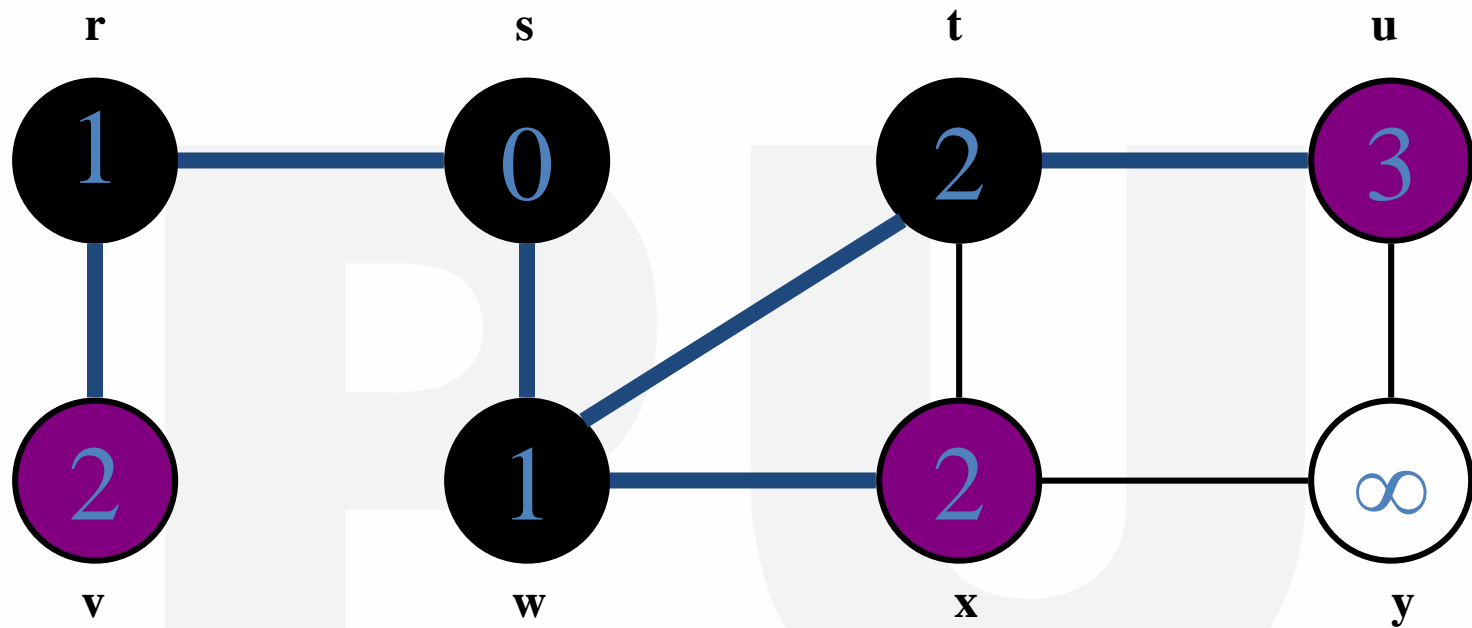
r	t	x
----------	----------	----------



Breadth-First Search: Example



Breadth-First Search: Example

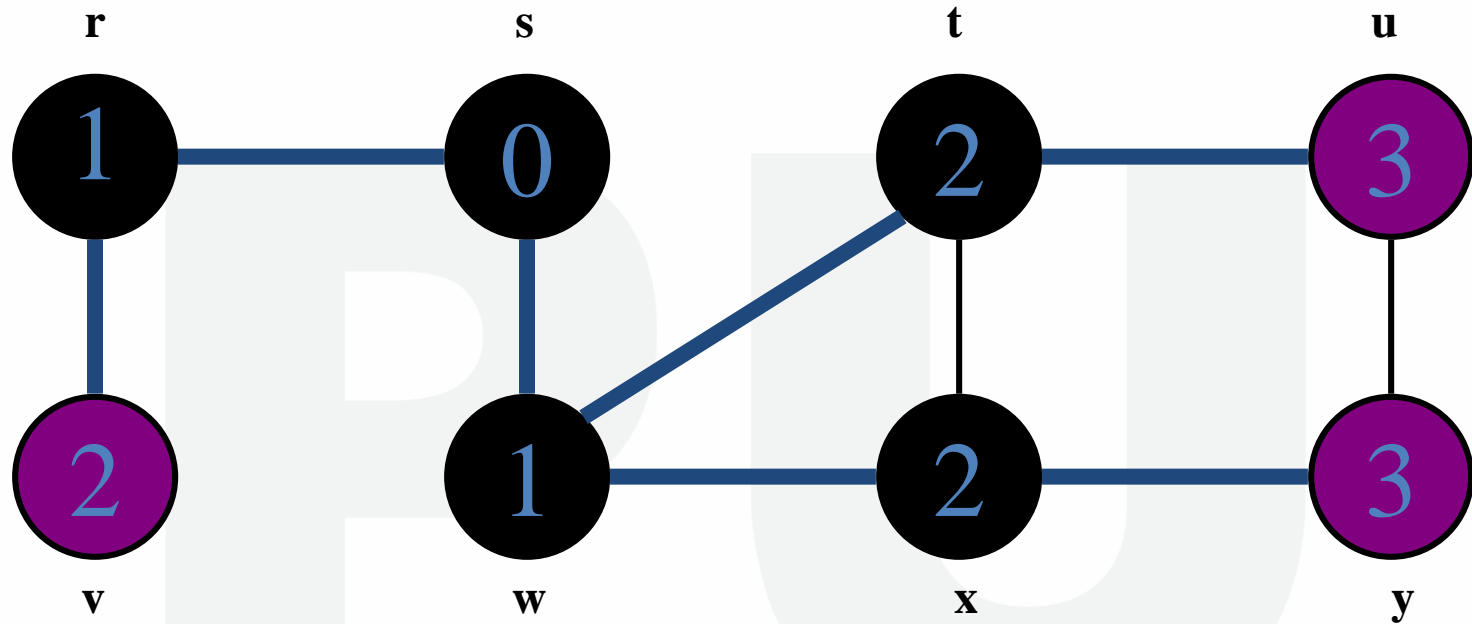


Q:

x	v	u
---	---	---



Breadth-First Search: Example

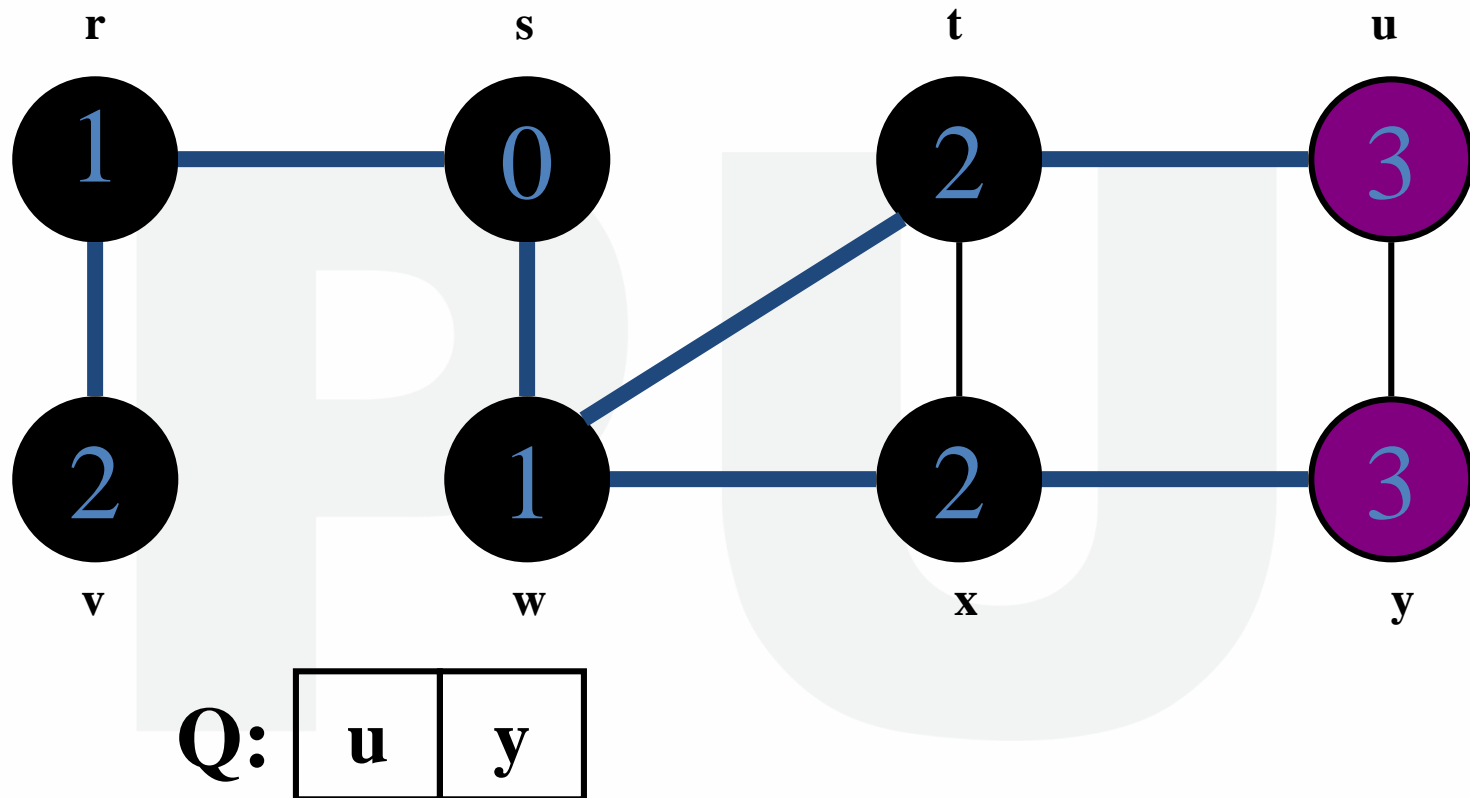


Q:

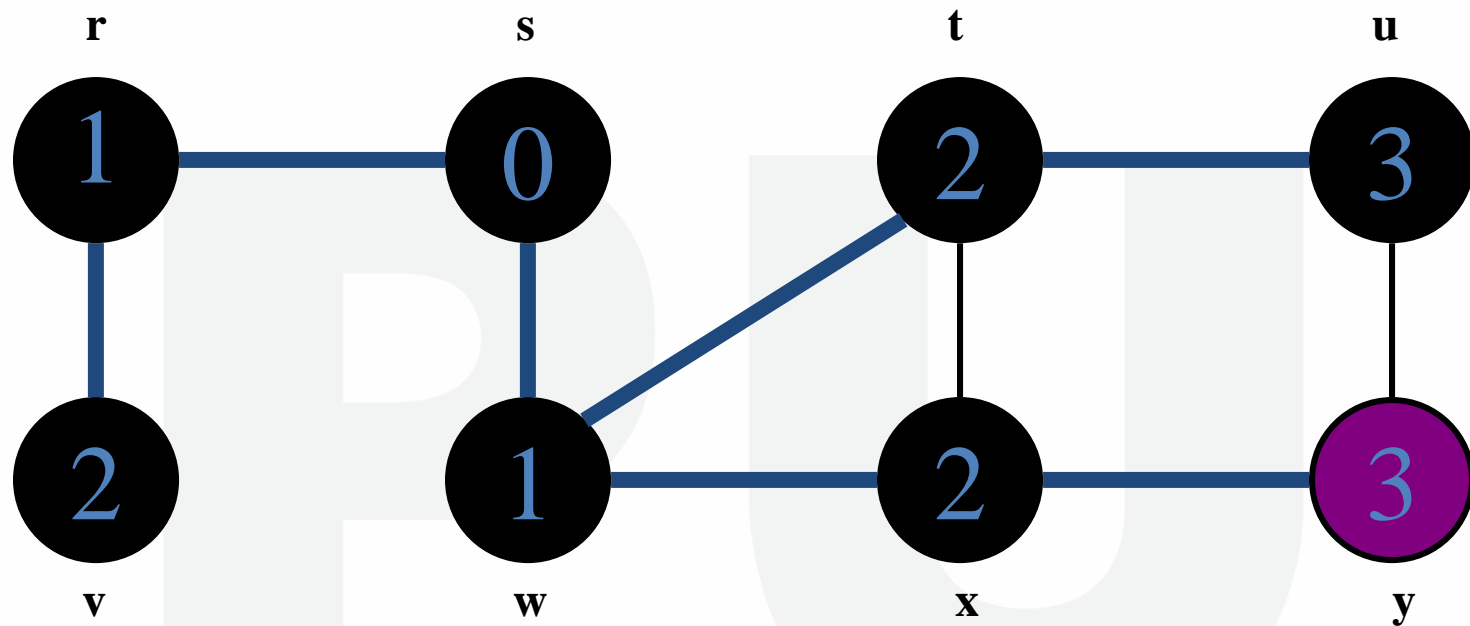
v	u	y
---	---	---



Breadth-First Search: Example



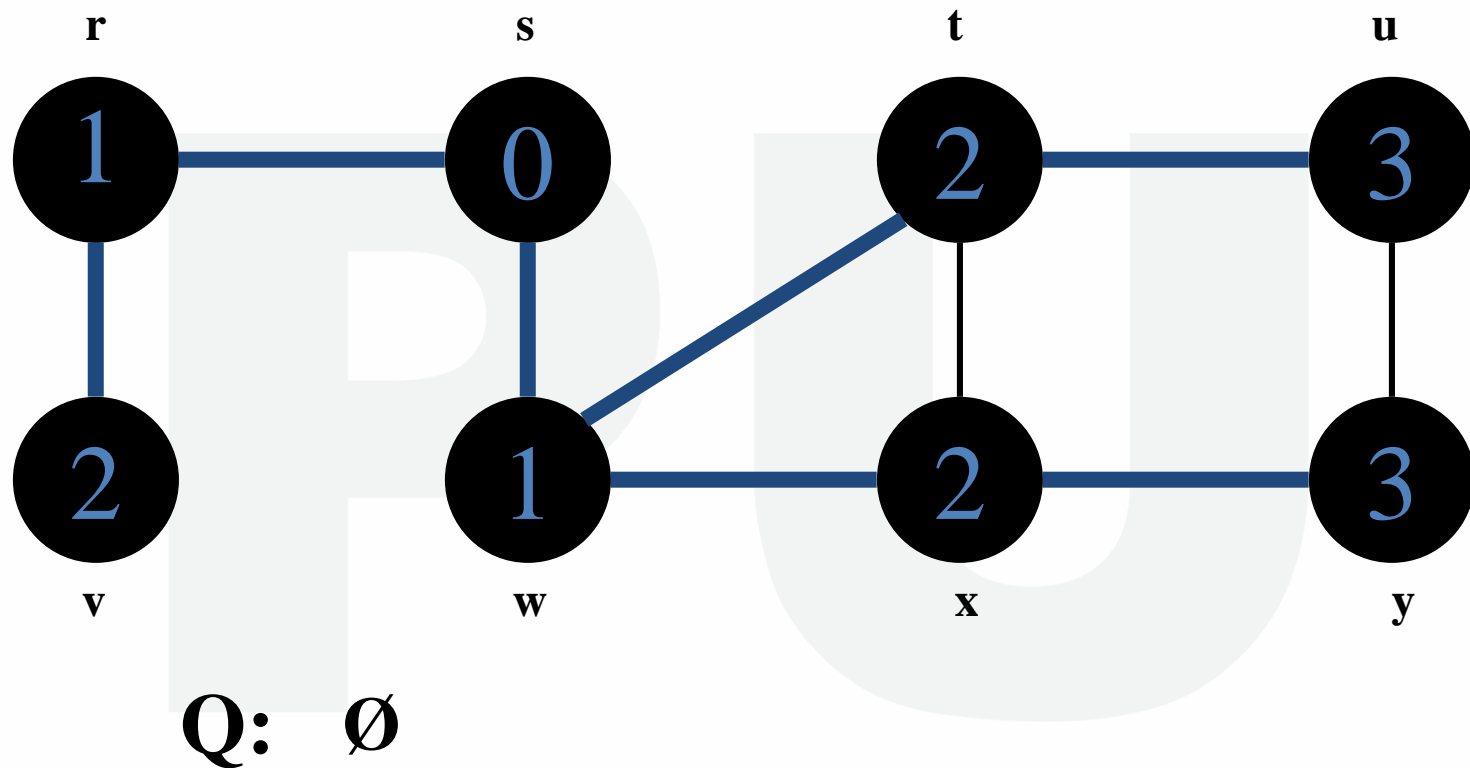
Breadth-First Search: Example



Q: y



Breadth-First Search: Example



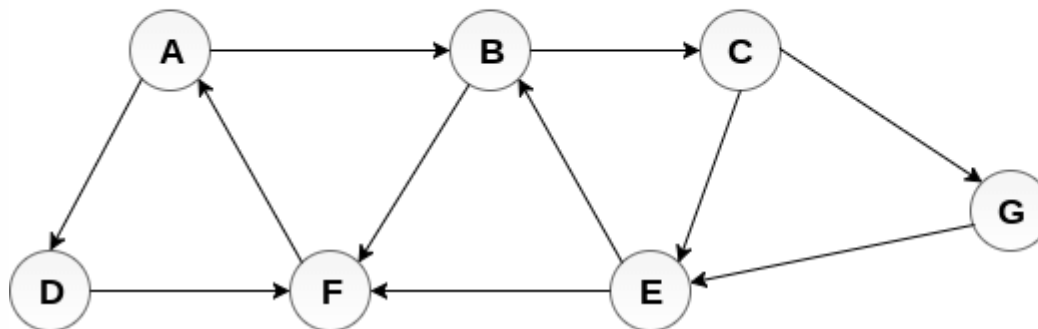
Breadth-First Search: Properties

- BFS calculates the shortest-path distance to the source node
 - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v , or ∞ if v not reachable from s
 - Proof given in the book
- BFS builds breadth-first tree, in which paths to root represent shortest paths in G
 - Thus can use BFS to calculate shortest path from one vertex to another in $O(V+E)$ time



Example

- Consider the graph G shown in the following image, calculate the minimum path p from node A to node E. Given that each edge has a length of 1.
- The minimum path will be $A \rightarrow B \rightarrow C \rightarrow E$.



Adjacency Lists

A : B, D

B : C, F

C : E, G

G : E

E : B, F

F : A

D : F



Requirements

- Can be used to attempt to visit all nodes of a graph in a systematic manner
- Works with directed and undirected graphs
- Works with weighted and unweighted graphs

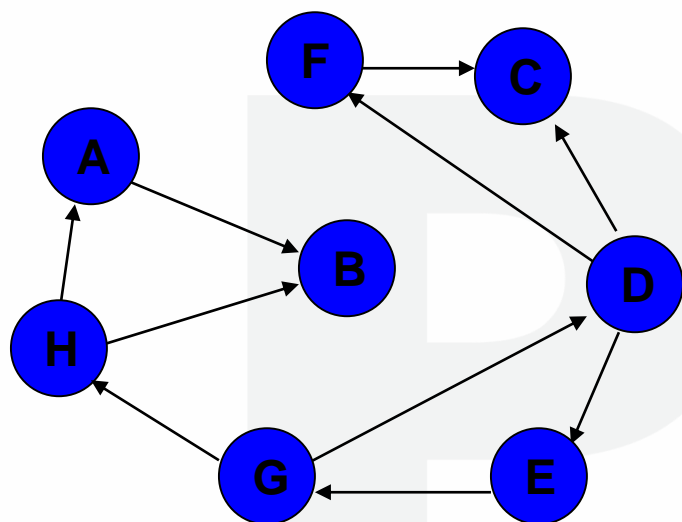


Depth First Search - DFS

- Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures.
- The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.
- Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



Walk-Through



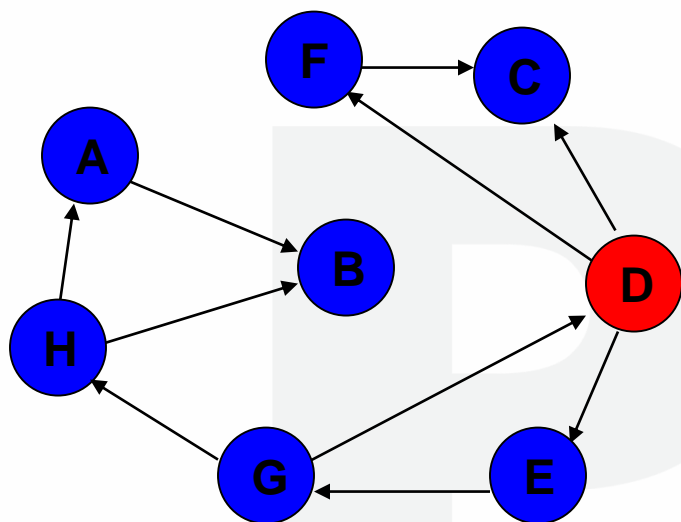
Visited Array



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



Walk-Through



The order nodes are visited:

D

Visited Array

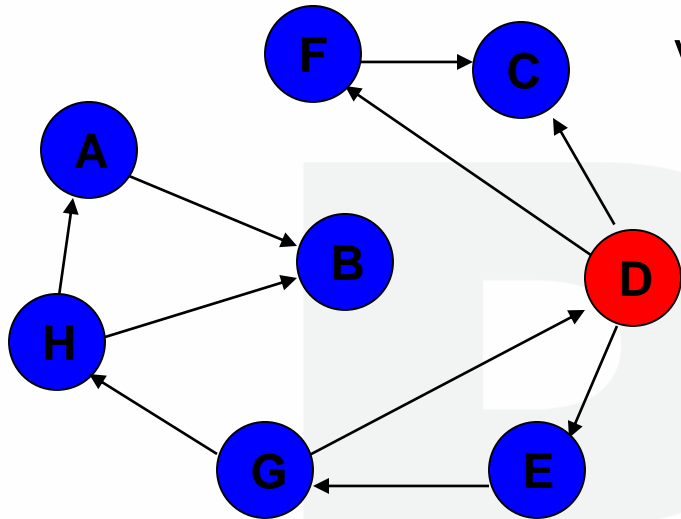
A	
B	
C	
D	✓
E	
F	
G	
H	

Visit D

D



Walk-Through



Visited Array

A	
B	
C	
D	✓
E	
F	
G	
H	



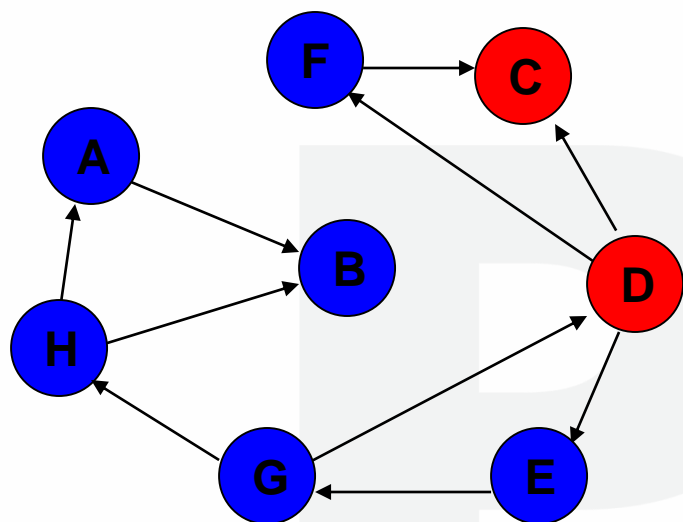
The order nodes are visited:

D

Consider nodes adjacent to D, decide to visit C first (Rule: visit adjacent nodes in alphabetical order)



Walk-Through



The order nodes are visited:

D, C

Visited Array

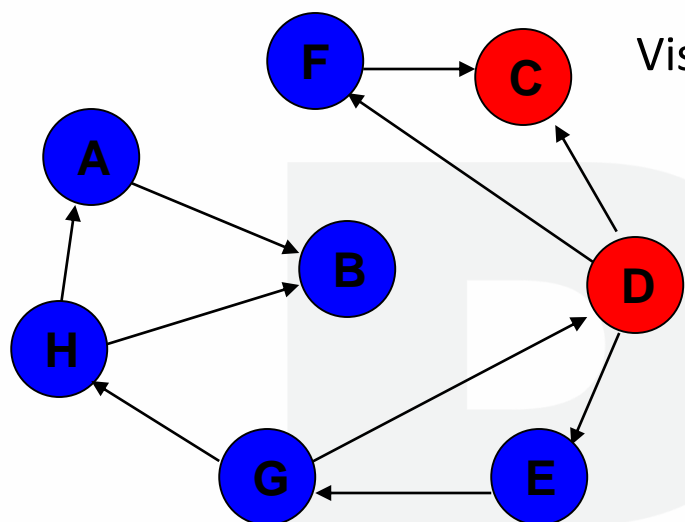
A	
B	
C	✓
D	✓
E	
F	
G	
H	

C
D

Visit C



Walk-Through



Visited Array

A	
B	
C	✓
D	✓
E	
F	
G	
H	



The order nodes are visited:

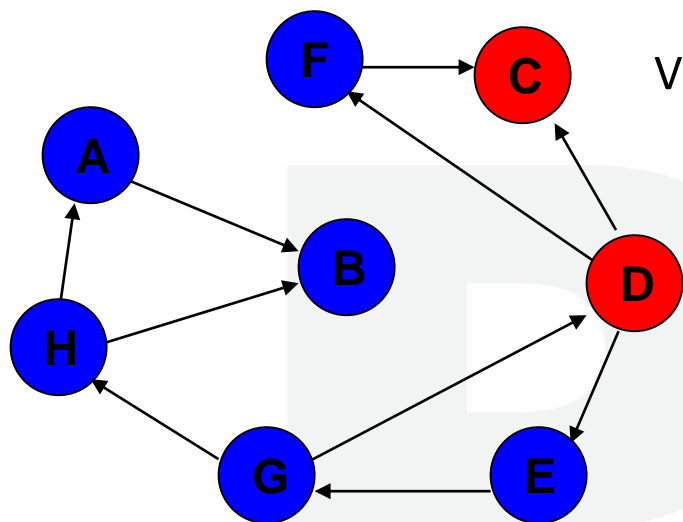
D, C \

No nodes adjacent to C; cannot continue

→ *backtrack*, i.e., pop stack and restore previous state



Walk-Through



Visited Array

A	
B	
C	✓
D	✓
E	
F	
G	
H	

D

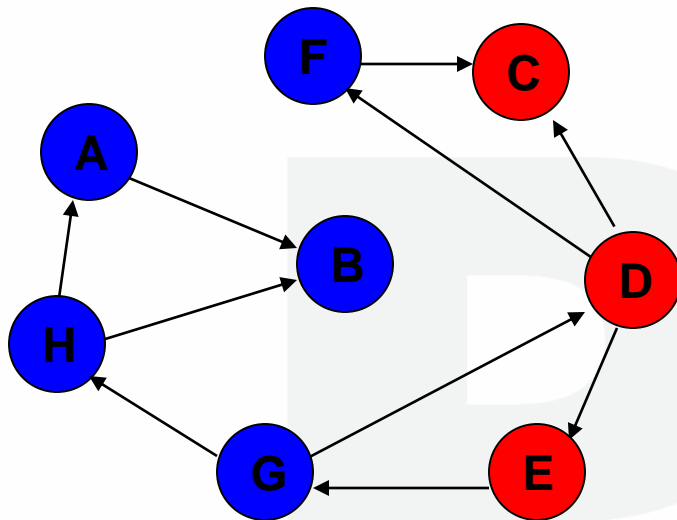
The order nodes are visited:

D, C

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



Walk-Through



Visited Array

A	
B	
C	✓
D	✓
E	✓
F	
G	
H	



The order nodes are visited:

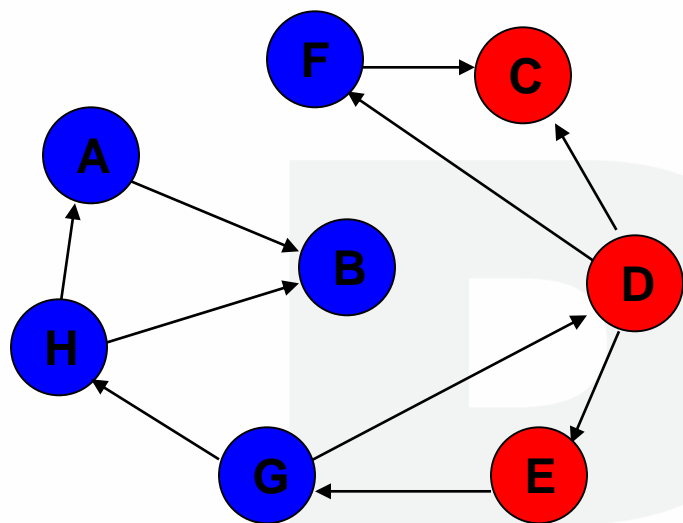
D, C, E \

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





Walk-Through



Visited Array

A	
B	
C	✓
D	✓
E	✓
F	
G	
H	

E
D

The order nodes are visited:

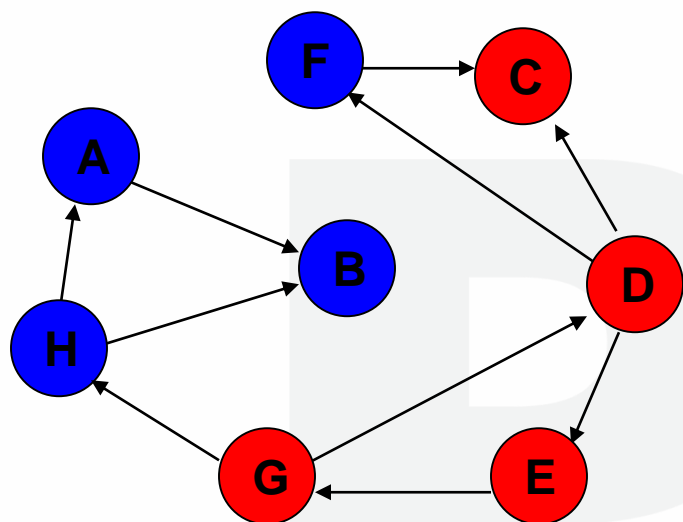
D, C, E \

Only G is adjacent to E





Walk-Through



Visited Array

A	
B	
C	✓
D	✓
E	✓
F	
G	✓
H	

G
E
D

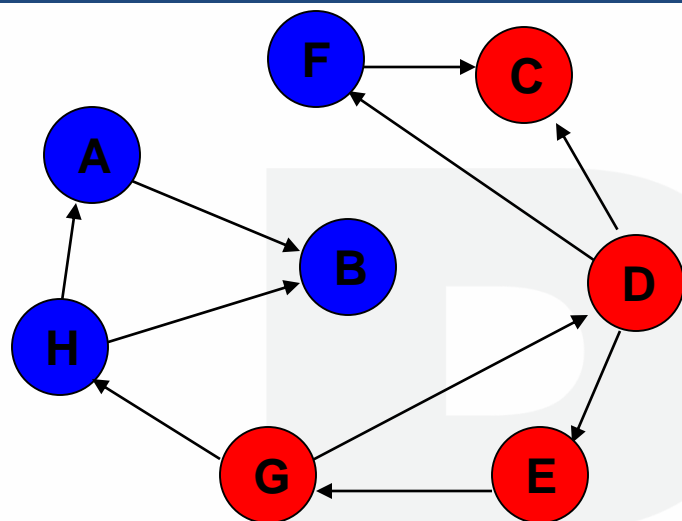
The order nodes are visited:

D, C, E, G\

Visit G



Walk-Through



Visited Array

A	
B	
C	✓
D	✓
E	✓
F	
G	✓
H	

G
E
D

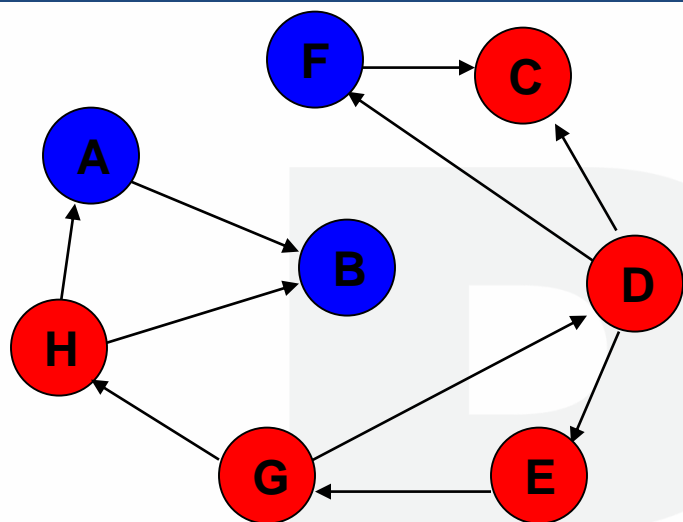
The order nodes are visited:

D, C, E, G

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



Walk-Through



Visited Array

A	
B	
C	✓
D	✓
E	✓
F	
G	✓
H	✓

H
G
E
D

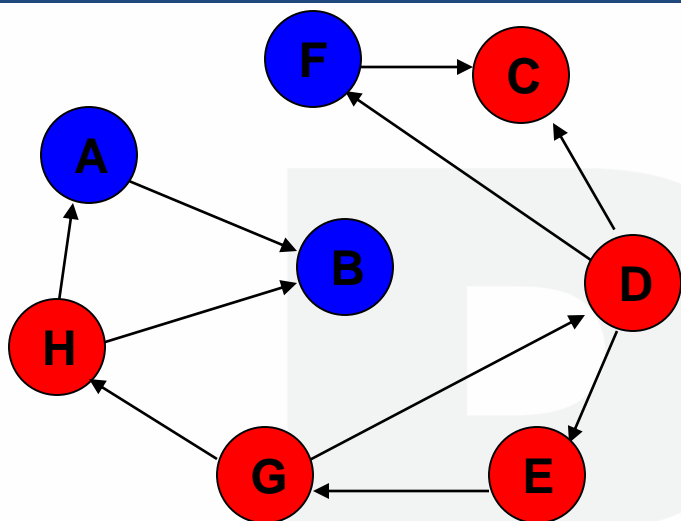
The order nodes are visited:

D, C, E, G, \H

Visit H



Walk-Through



Visited Array

A	
B	
C	✓
D	✓
E	✓
F	
G	✓
H	✓

H
G
E
D

The order nodes are visited:

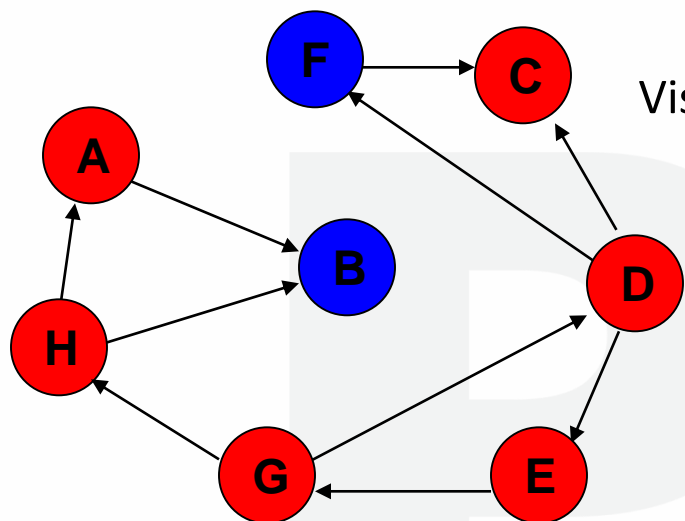
D, C, E, G, H

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





Walk-Through



Visited Array

A	✓
B	
C	✓
D	✓
E	✓
F	
G	✓
H	✓

A
H
G
E
D

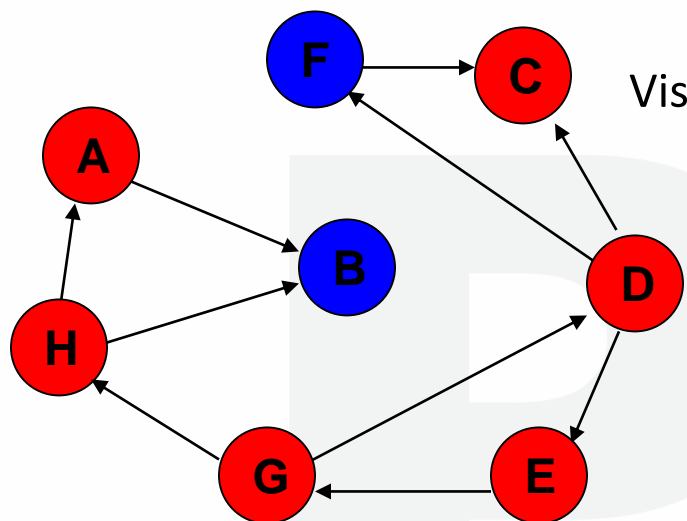
The order nodes are visited:

D, C, E, G, \H, A

Visit A



Walk-Through



Visited Array

A	✓
B	
C	✓
D	✓
E	✓
F	
G	✓
H	✓

A
H
G
E
D

The order nodes are visited:

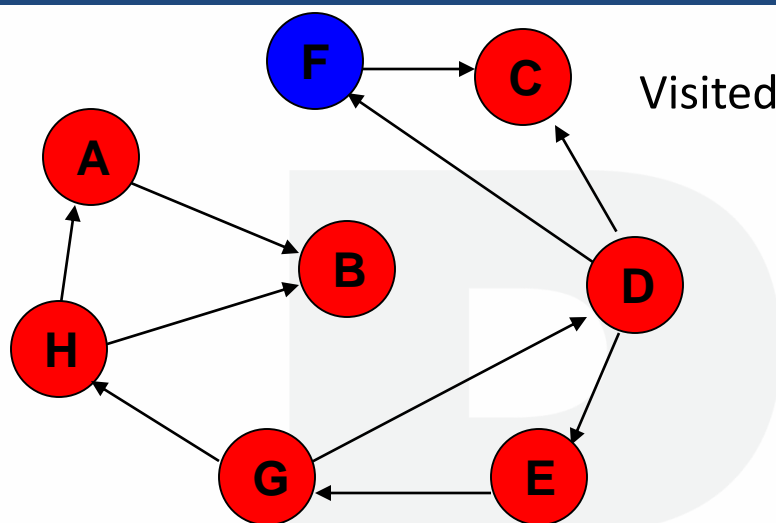
D, C, E, G, \H, A

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





Walk-Through



Visited Array

A	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓

B
A
H
G
E
D

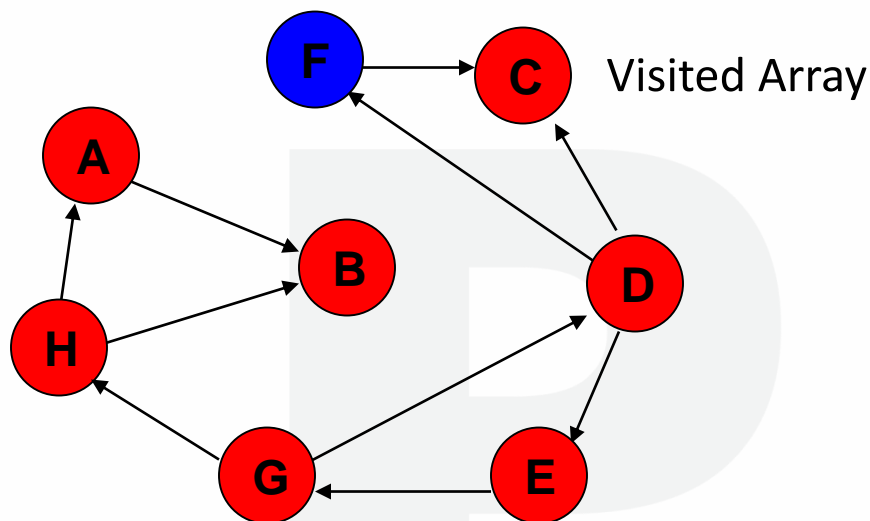
The order nodes are visited:

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

Visit B



Walk-Through



The order nodes are visited:

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

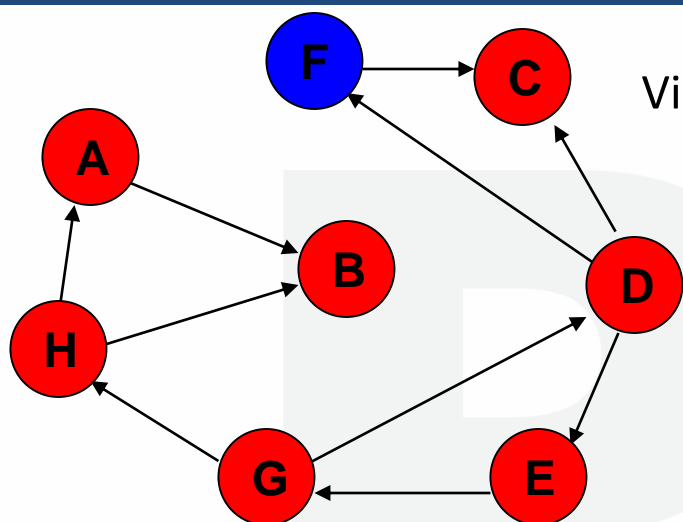
A	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓

A
H
G
E
D
\
/

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



Walk-Through



Visited Array

A	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓

H
G
E
D

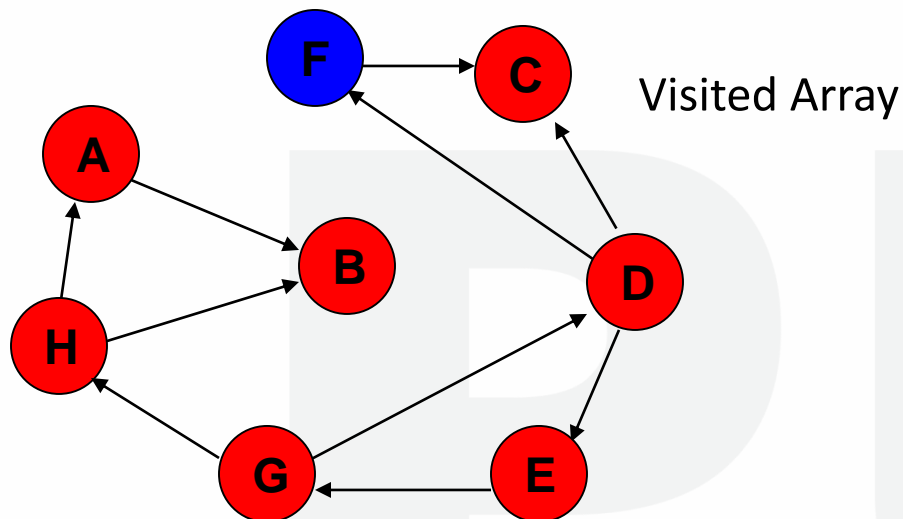
The order nodes are visited:

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

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



Walk-Through



The order nodes are visited:

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

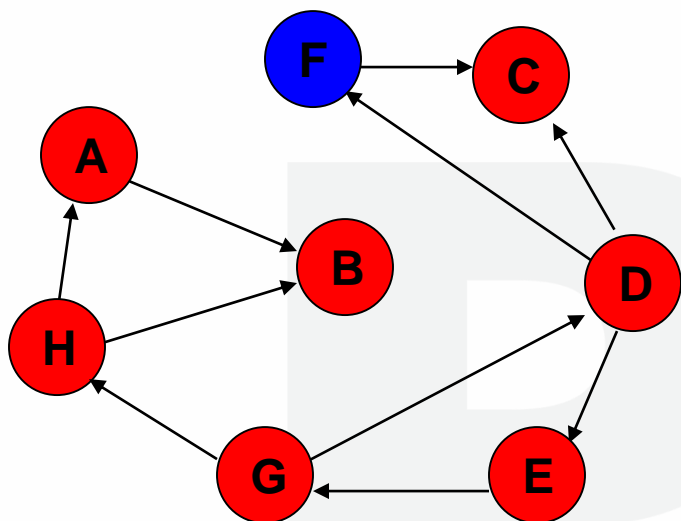
A	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓

G
E
D

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



Walk-Through



Visited Array

A	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓

E
D

The order nodes are visited:

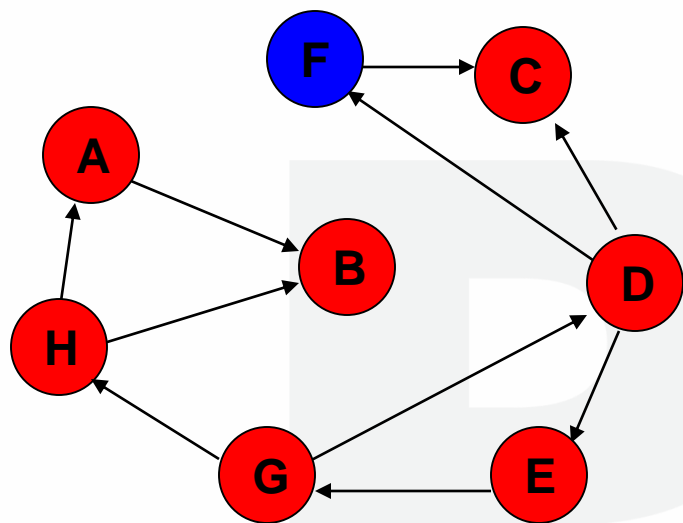
D, C, E, G, H, A, B

No unvisited nodes adjacent to G.

Backtrack (pop the stack).



Walk-Through



Visited Array

A	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓



The order nodes are visited:

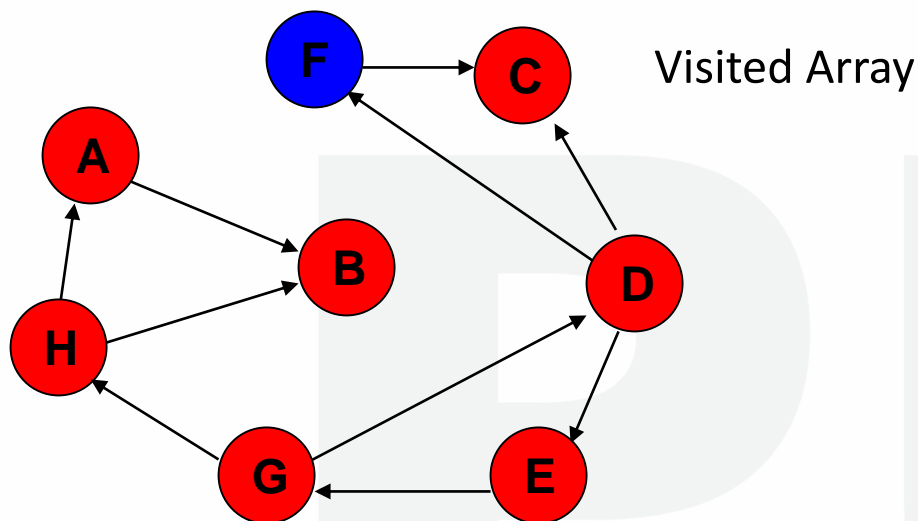
D, C, E, G, \H, A, B

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





Walk-Through



A	✓
B	✓
C	✓
D	✓
E	✓
F	
G	✓
H	✓



The order nodes are visited:

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

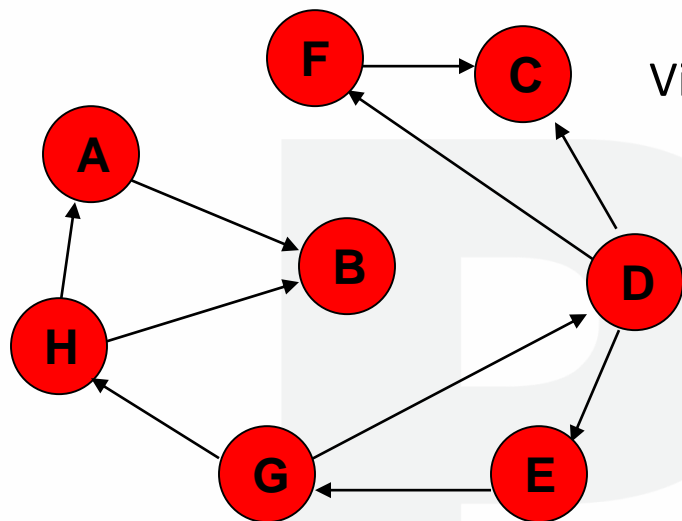
F is unvisited and is adjacent to D.

Decide to visit F next.





Walk-Through



Visited Array

A	✓
B	✓
C	✓
D	✓
E	✓
F	✓
G	✓
H	✓

F
D

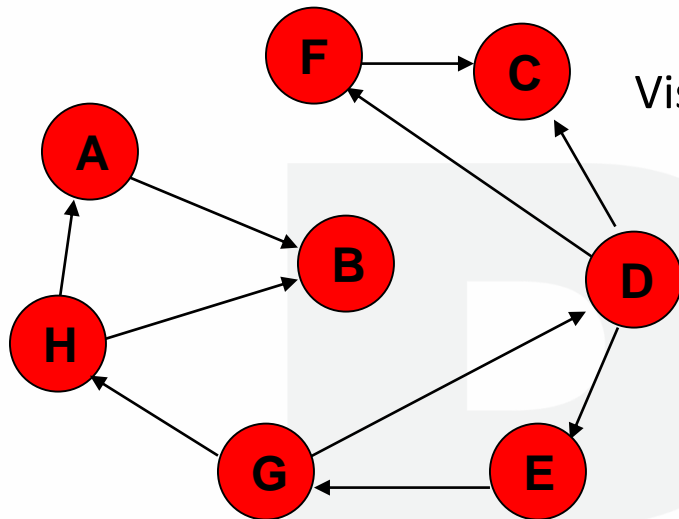
Visit F

The order nodes are visited:

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



Walk-Through



Visited Array

A	✓
B	✓
C	✓
D	✓
E	✓
F	✓
G	✓
H	✓



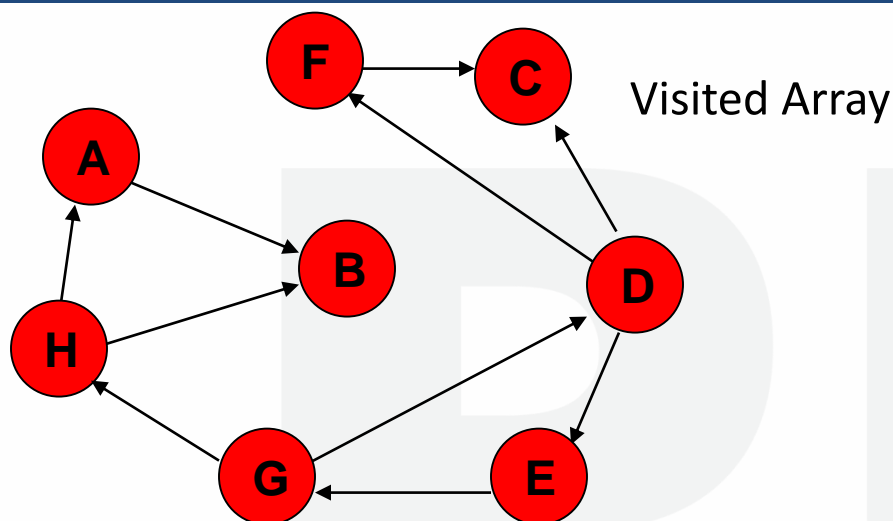
The order nodes are visited:

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

No unvisited nodes adjacent to F.
Backtrack.



Walk-Through



The order nodes are visited:

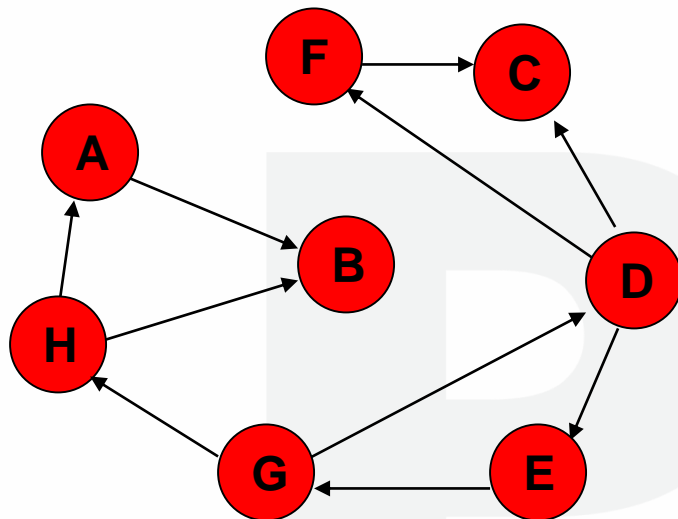
D, C, E, G, H, A, B, F

A	✓
B	✓
C	✓
D	✓
E	✓
F	✓
G	✓
H	✓

No unvisited nodes adjacent to D. Backtrack.



Walk-Through



Visited Array

A	✓
B	✓
C	✓
D	✓
E	✓
F	✓
G	✓
H	✓



The order nodes are visited:

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

Stack is empty. Depth-first traversal is done.



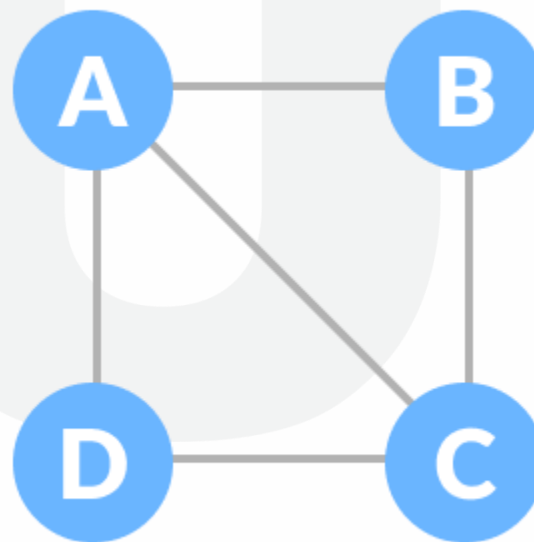
Consider Trees

1. What depth-first traversals do you know?
2. How do the traversals differ?
3. In the walk-through, we visited a node just as we pushed the node onto the stack. Is there another time at which you can visit the node?
4. Conduct a depth-first search of the same graph using the strategy you came up with in #3.



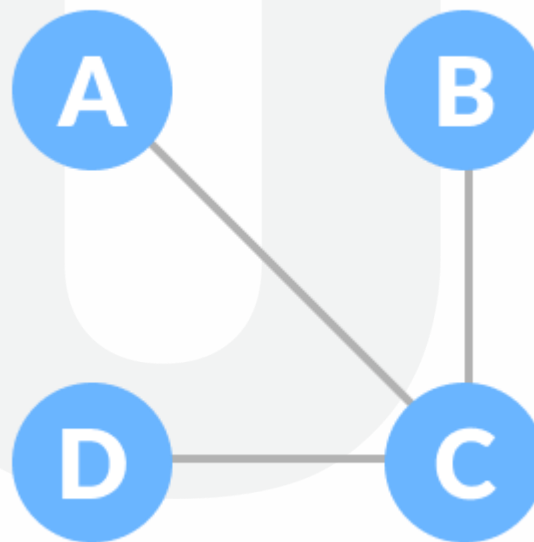
Spanning Tree

- Before we learn about spanning trees, we need to understand two graphs: undirected graphs and connected graphs.
- An undirected graph is a graph in which the edges do not point in any direction (ie. the edges are bidirectional).



Spanning Tree

- A connected graph is a graph in which there is always a path from a vertex to any other vertex.



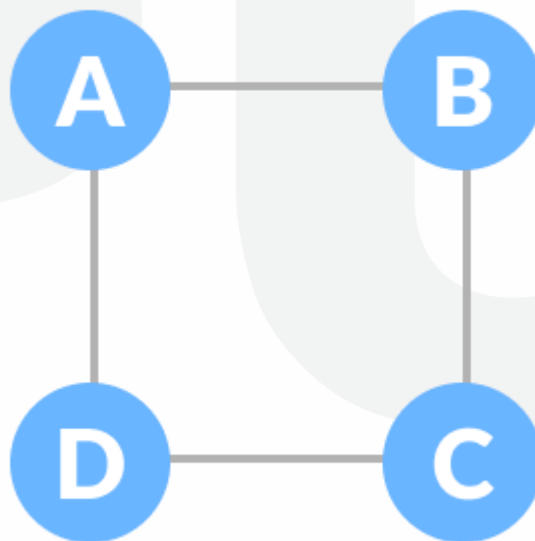
Spanning Tree

- A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.
- Some properties of a spanning tree can be deduced from this definition:
- Since “a spanning tree covers all of the vertices”, it cannot be disconnected.
- A spanning tree cannot have any cycles and consist of $(n-1)$ edges (where n is the number of vertices of the graph) because “it uses the minimum number of edges”.
- The total number of spanning trees with n vertices that can be created from a complete graph is equal to $n(n-2)$.



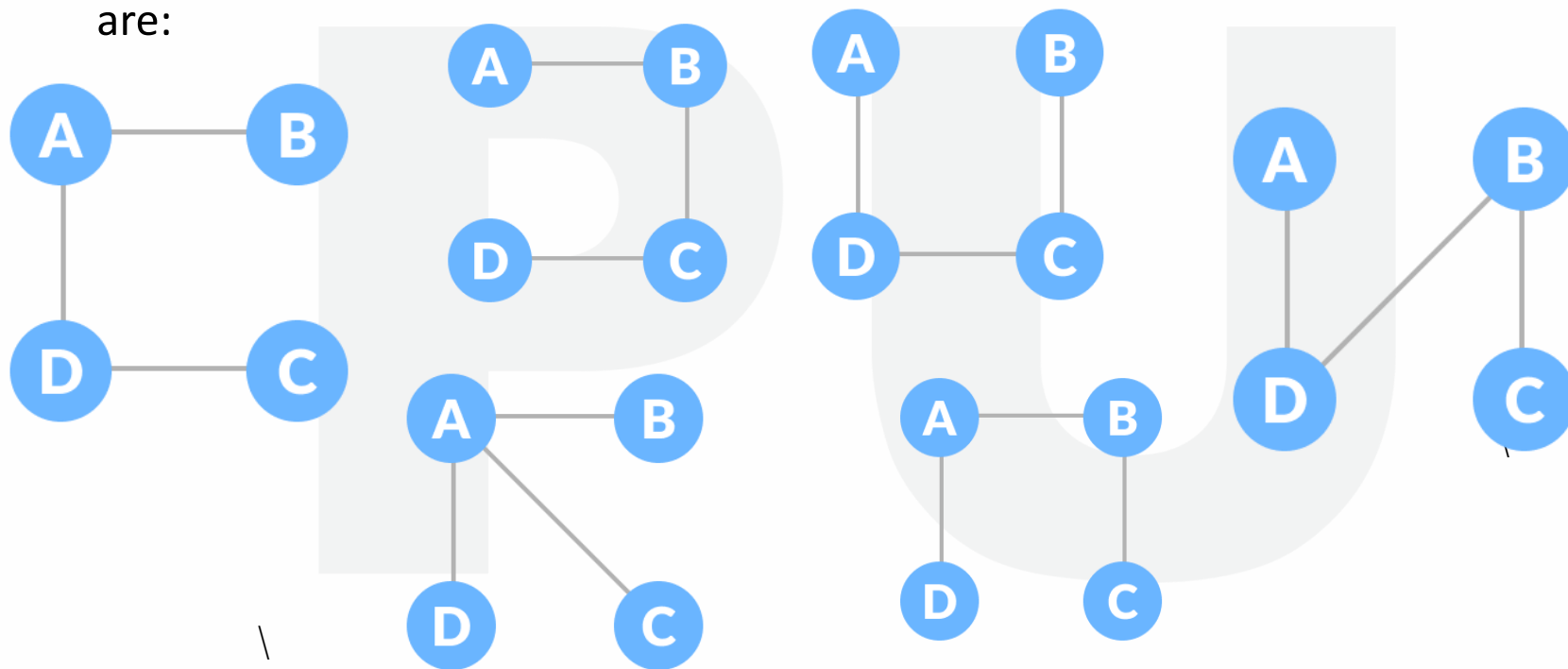
Example of a Spanning Tree

- Let's understand the spanning tree with examples below:
- Let the original graph be:



Example of a Spanning Tree

- Some of the possible spanning trees that can be created from the above graph are:



× ○ DIGITAL LEARNING CONTENT



Parul[®] University



www.paruluniversity.ac.in

