Module 3.: Trees and Graphs

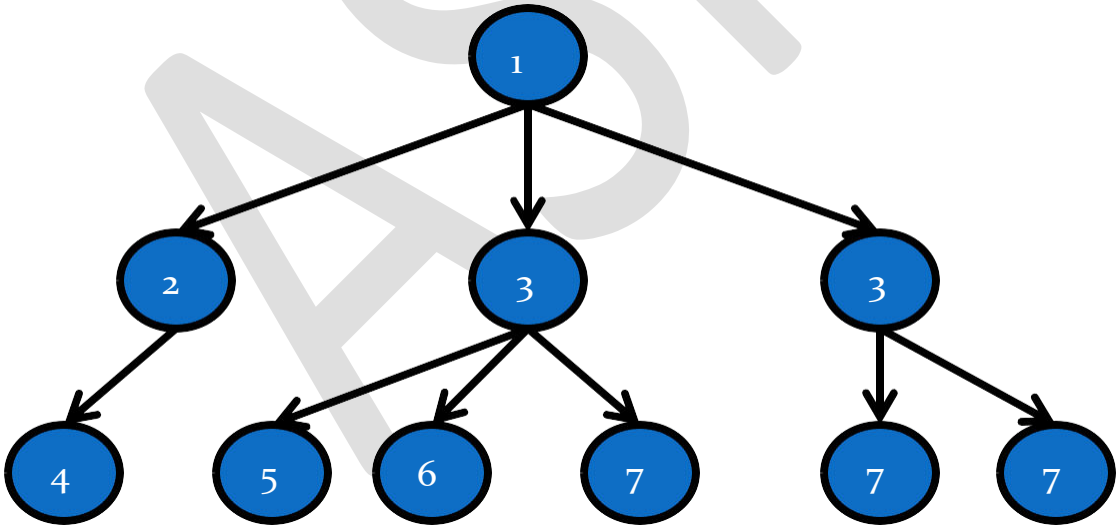
Syllabus

Trees:- Representation Of Trees; Binary Trees - Types and Properties, Binary Tree Representation, Tree Operations, Tree Traversals; Expression Trees; Binary Search Trees - Binary Search Tree Operations; Binary Heaps - Binary Heap Operations, Priority Queue.

Graphs:- Definitions; Representation of Graphs; Depth First Search and Breadth First Search; Applications of Graphs - Single Source All Destination.

Tree

A tree is a finite collection of special data items known as nodes, organized in a hierarchical structure to represent parent-child relationships. Unlike linear data structures such as lists or arrays, which have a linear arrangement of elements, a tree structure branches out in multiple directions, resembling a tree with branches extending from a central trunk. Each node in a tree can have zero or more child nodes, and there are no restrictions on the number of children a node can have. This hierarchical arrangement allows for efficient representation and manipulation of data, making trees suitable for a wide range of applications, including computer science, database systems, and hierarchical data storage.

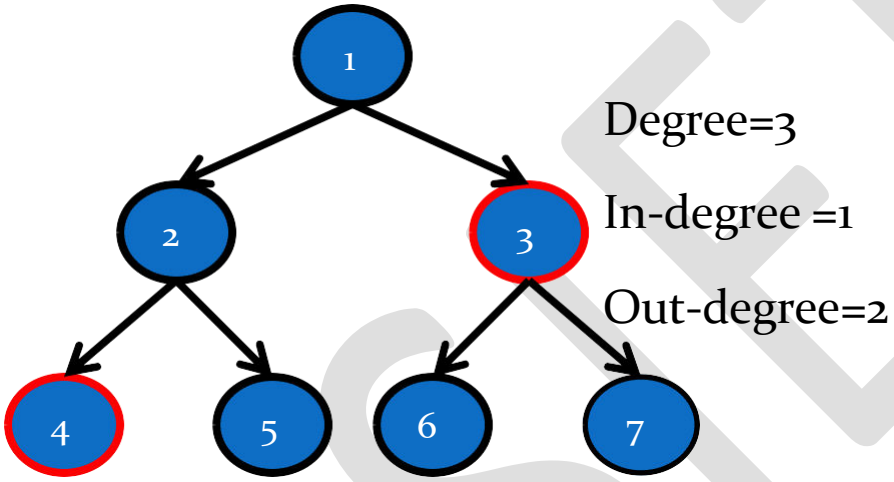


DEGREE of a node: Number of nodes connected to any node

IN-DEGREE of a node: Number of edges that ends at a particular node

OUT-DEGREE: Number of edges that starts from a particular node

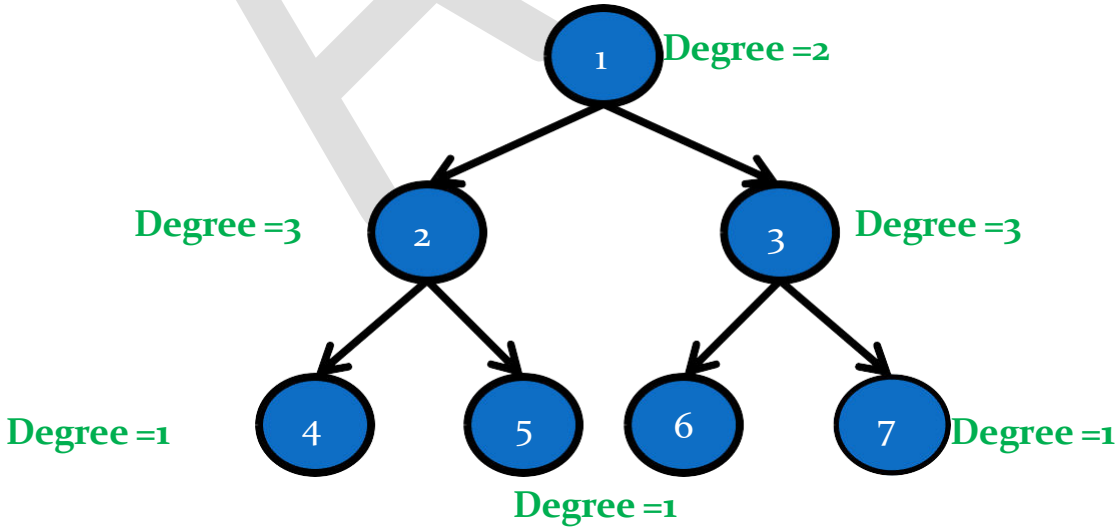
Degree = in-degree+ out-degree



Degree of a tree:

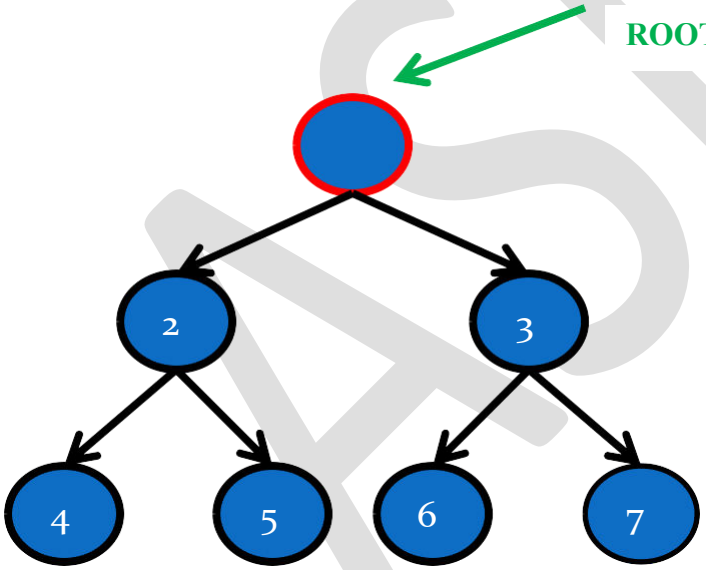
Maximum degree of any node

Degree of tree = 3



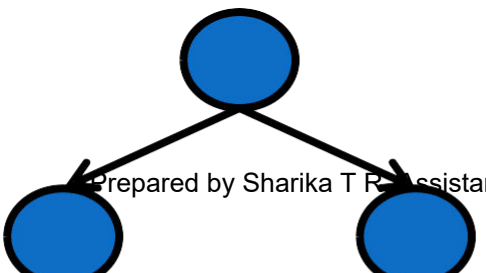
ROOT

First node which does not have any parent Node with In-degree =0



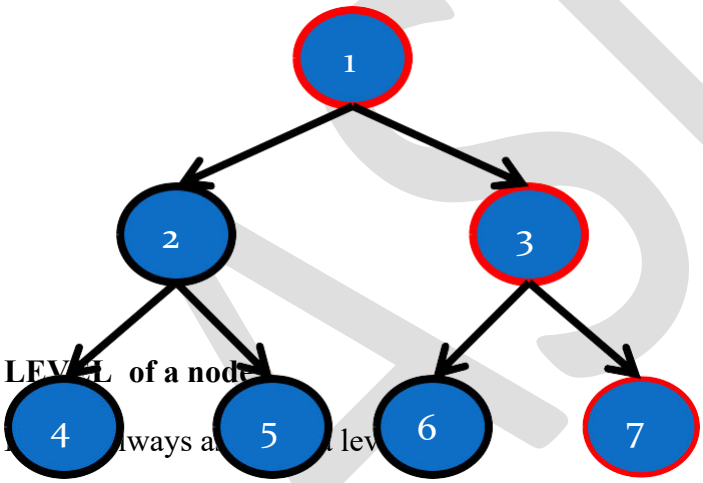
LEAF NODE or TERMINAL NODE

Last node of the tree which does not have any descendants Node with out-degree= o



PATH

Sequence of edges between any two nodes

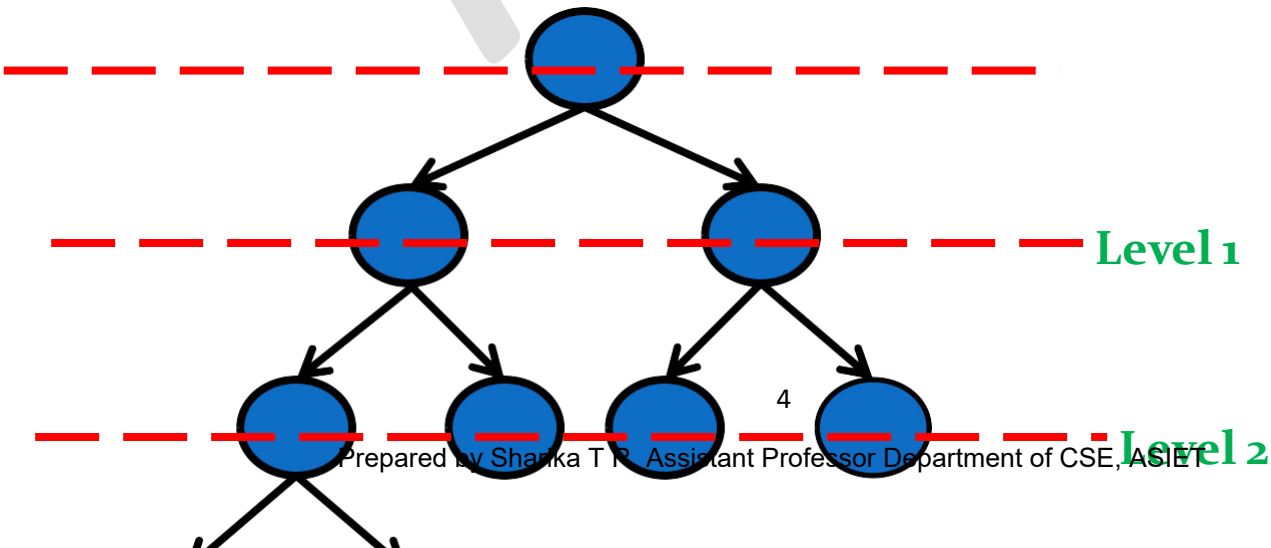


Path : 1-3, 3-7

LEVEL of a node

[illegible] always a[illegible] level

Level of any other node = Level of its parent +1



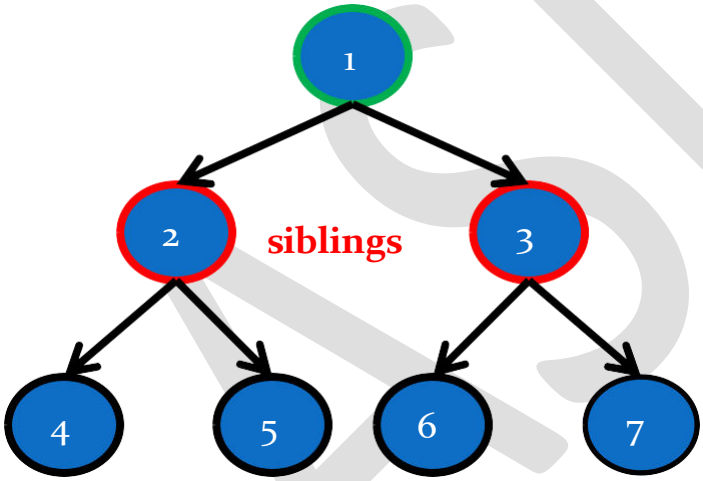
DEPTH of a node

No. of nodes in the longest path from the root to any terminal node Depth of a node = level of a node +1

Depth of a tree = largest level of a tree +1

Siblings

Children of same parent node



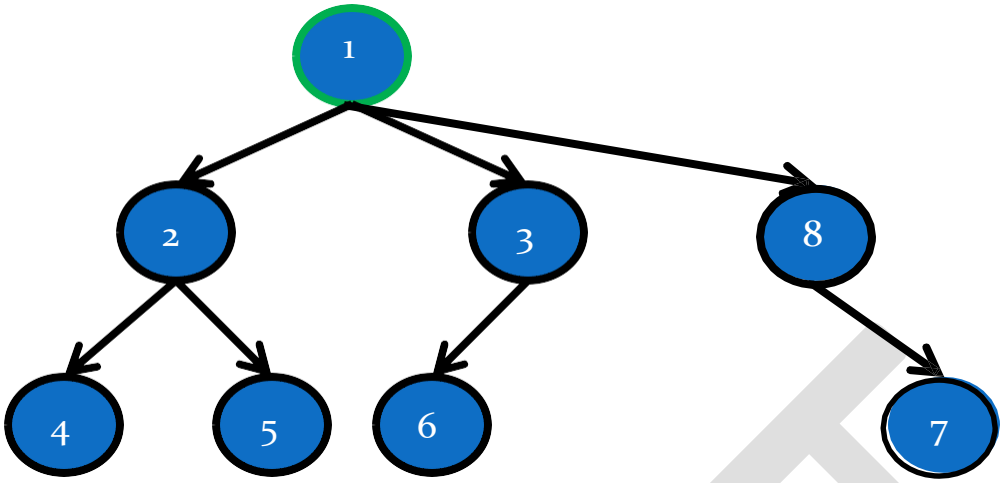
TREE: Definition

A tree is a finite set of one or more nodes such that

1. There is a specially designated node called the root
2. The remaining nodes are partitioned into n>=o disjoint sets T1,

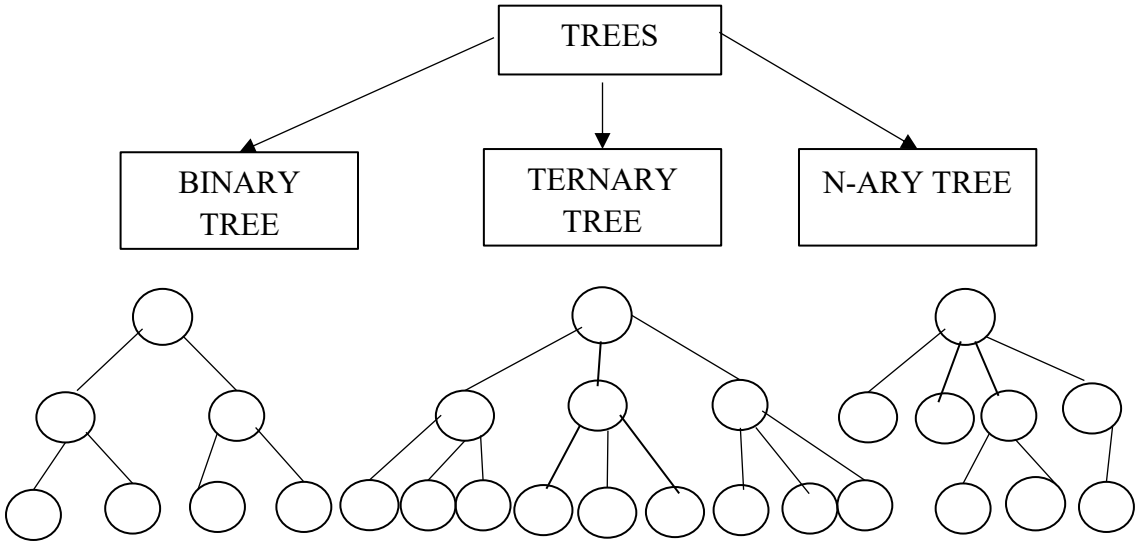
T2, …Tn where each of these sets is a tree

T1, T2, …Tn are called the subtrees of the root



Types of Tree data structures:

- **Binary tree**: In a binary tree, each node can have a maximum of two children linked to it. Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees.
- **Ternary Tree**: A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.
- **N-ary Tree or Generic Tree**: Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.



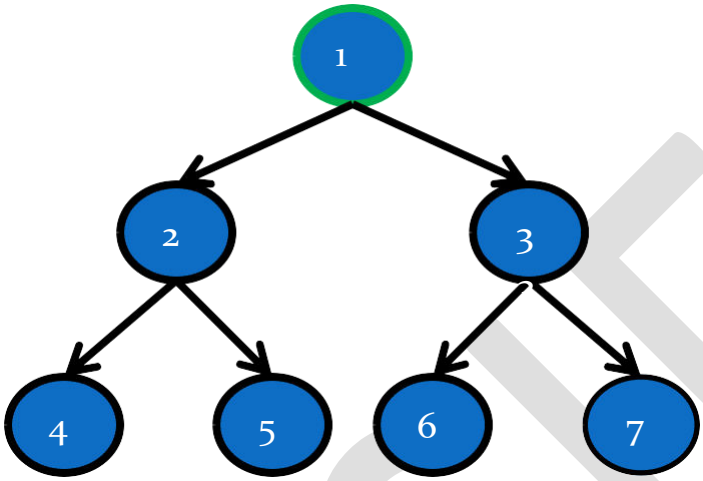
BINARY TREE: Definition

- A binary tree is a finite set of one or more nodes such that

    1. There is a specially designated node called the root
    2. The remaining nodes are partitioned into N=0,1 or 2 disjoint sets

       T1 and T2 where each of these sets is a tree

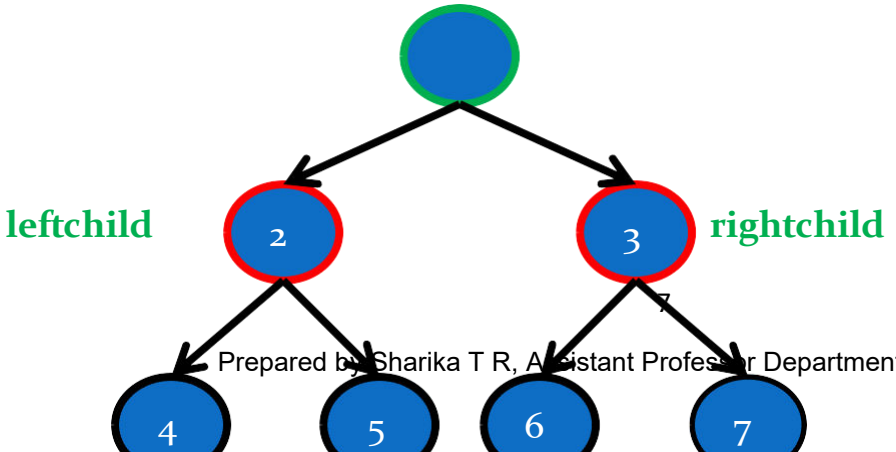
       T1 and T2 are called the subtrees of the root



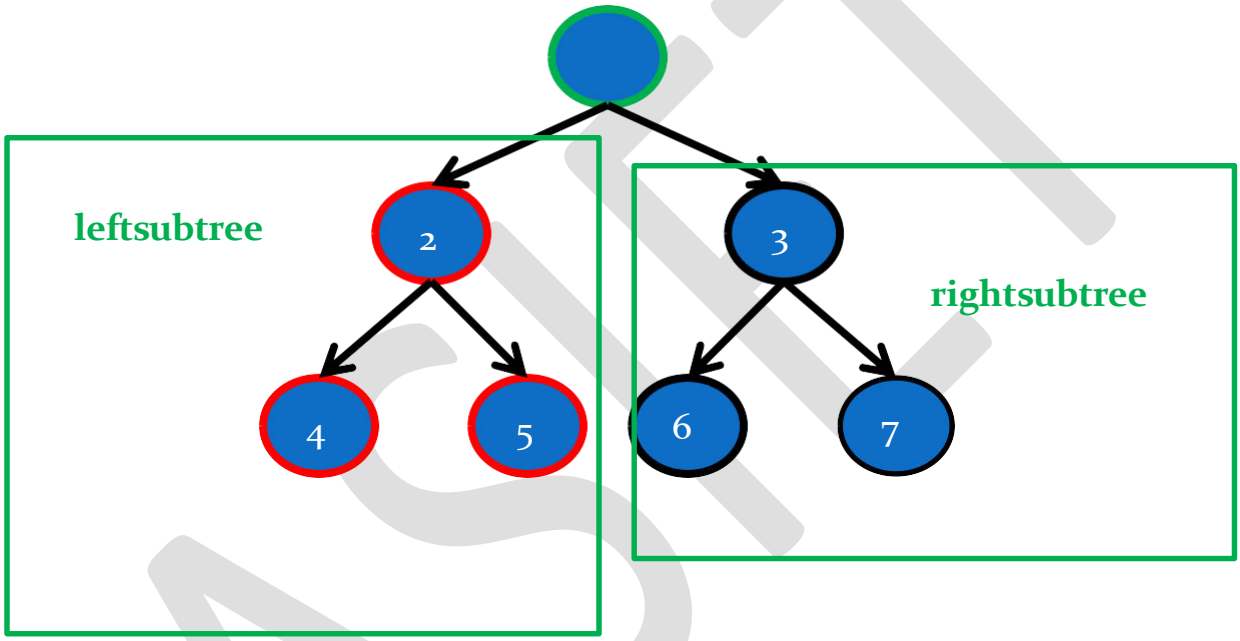
A binary tree is a specific type of tree structure characterized by the following properties:

- Each node in a binary tree can have a maximum of two children, referred to as the left child and the right child.
- Every node in a binary tree can have either zero, one, or two children.
- The tree may be empty, represented by a null or empty root node.
- Each node's left child is located on the left side of the parent node, while the right child is situated on the right side of the parent node.

Binary trees are widely used in computer science and data structures due to their simplicity and versatility. They form the foundation for more complex data structures like binary search trees and heaps, enabling efficient storage and retrieval of data in various applications. The hierarchical nature of binary trees facilitates operations such as traversal, insertion, deletion, and searching, making them essential in algorithm design and implementation.



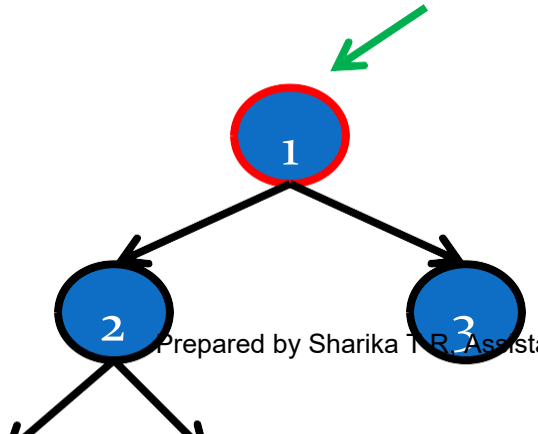
- Left**subtree** - Tree on the left of the parent
- Right**subtree** - Tree on the right of the parent



Classification Of Binary Tree

1. Strictly binary tree

A binary tree in which every nonleaf node has non empty left and right sub tree A strictly binary tree with N leaves always contains 2N-1 nodes



2. **Complete binary tree**

- A binary tree with all levels except the last level contains the maximum number of possible nodes and all nodes in the last level appear as left as possible

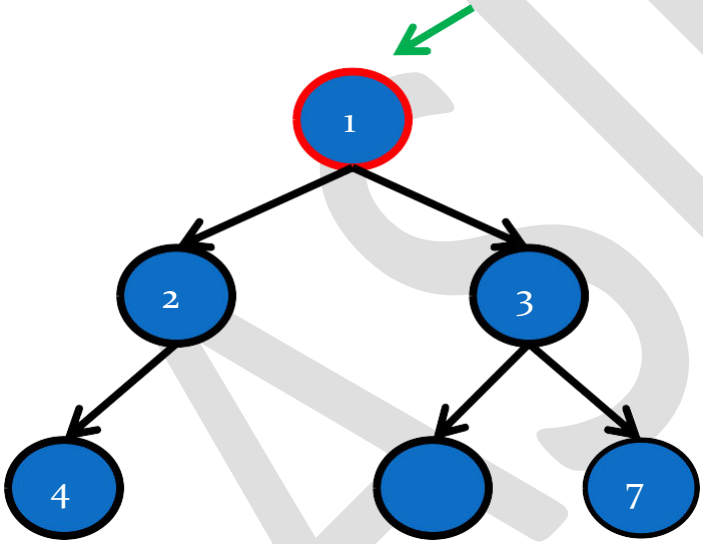
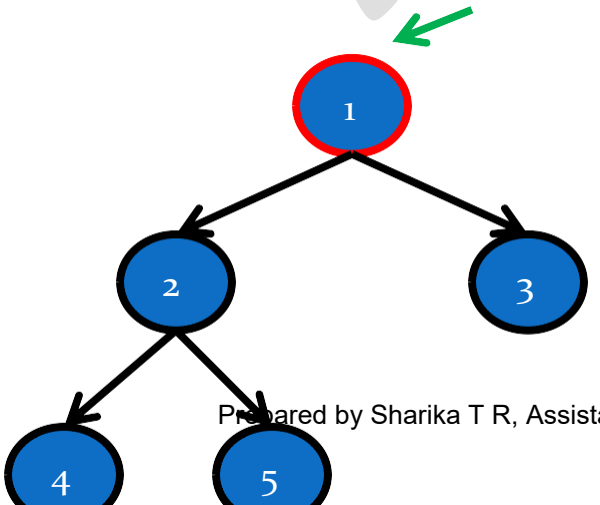


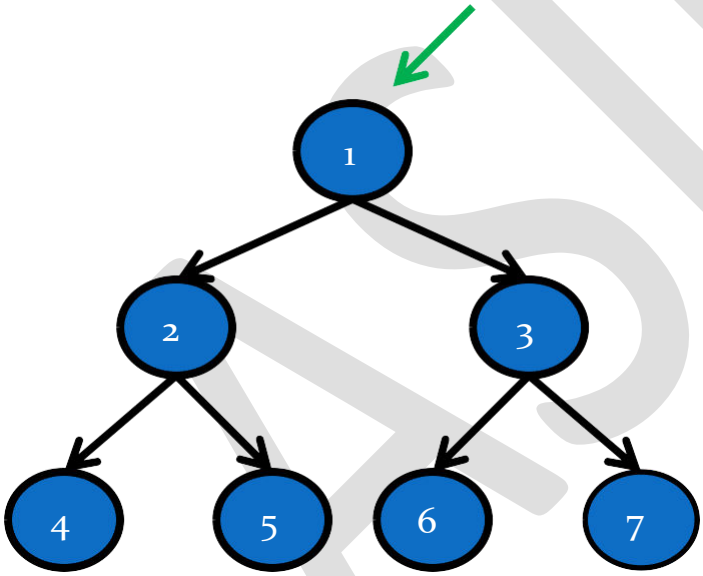
Figure 2: Incomplete Binary Tree



3. **Full binary tree**

A binary tree that contains maximum possible number of nodes at all levels

A tree of depth d will have 2^d-1 nodes



D=3

Nodes = 2^3-1 =7

4. Skewed Binary Trees

A skewed binary tree is a type of binary tree in which each node has only one child, except for the leaf nodes. This results in a tree that is essentially a linear structure, with all nodes aligned either to the left or to the right. Skewed binary trees can be either left-skewed (all nodes have a left child) or right-skewed (all nodes have a right child). They are typically unbalanced and can lead to inefficient operations such as traversal and searching.

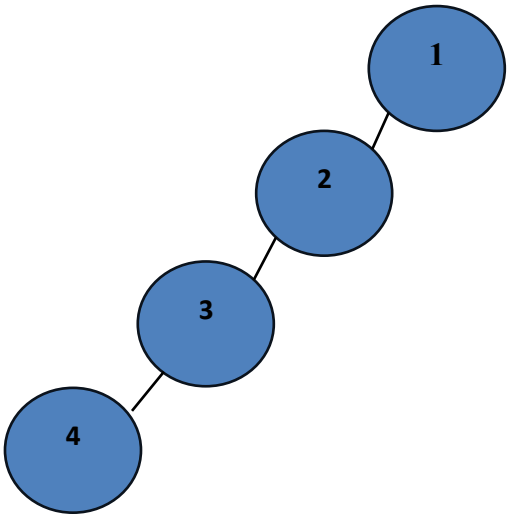


Figure 4: Left Skewed Binary Tree

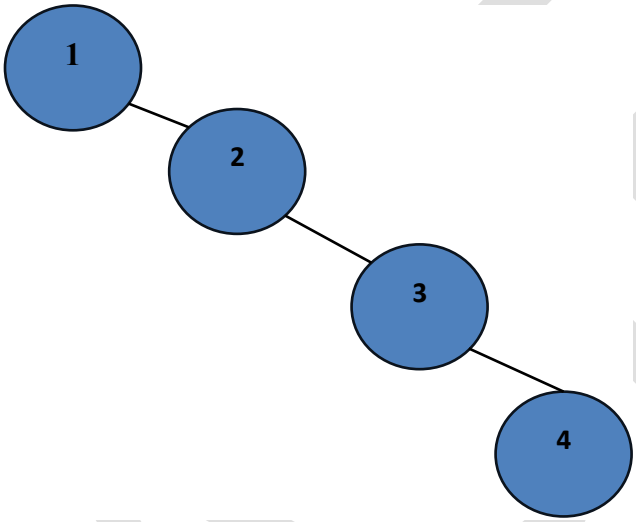
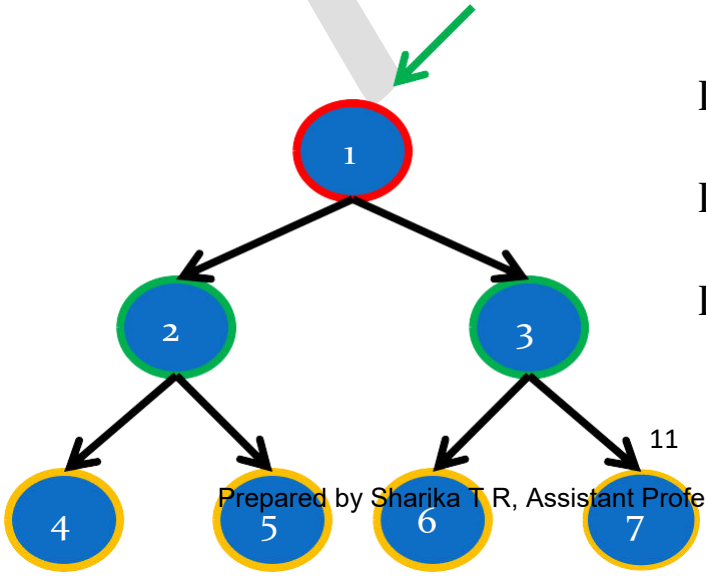


Figure 5:Right Skewed Binary Tree

Properties Of Binary Tree

1. Maximum number of nodes on level l is 2^l, where l>=0

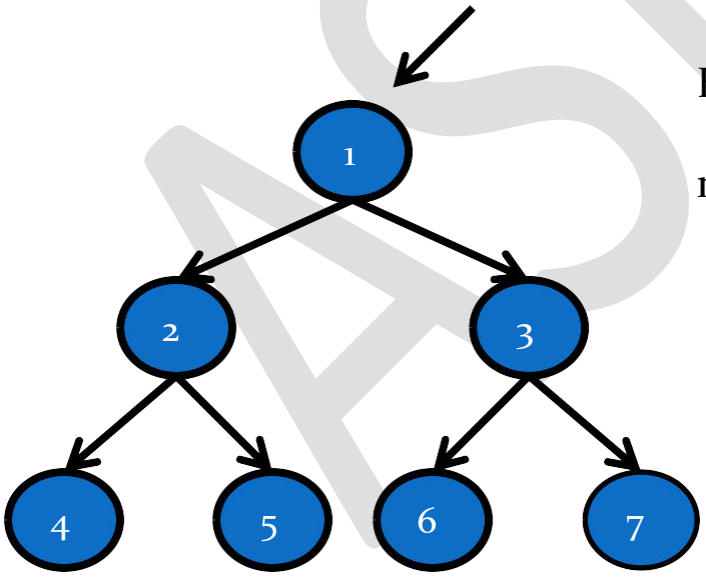


L=0, nodes = $2^0=1$

L=1, nodes = $2^1=2$

L=2, nodes = $2^2=4$

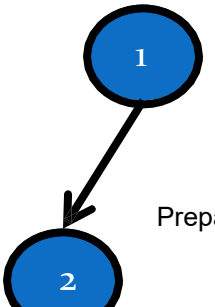
2. Maximum number of nodes possible in a binary tree of height h is 2^h-1.



H=3

nodes= 2^3-1 = 7

3. Minimum number of nodes possible in a binary tree of height h is h.



H=3

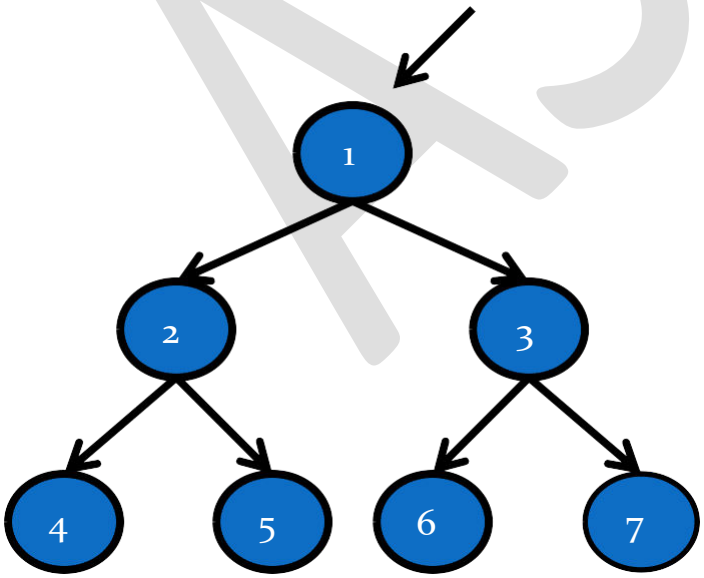
nodes= 3

Representations Of Binary Tree

1. Linear or sequential representation
2. Linked or pointer representation

1. Linear or sequential representation

- A block of memory for an array is allocated before storing the actual tree in it.
- Once memory is allocated, the size of the tree is restricted as permitted by the memory.
- In this representation, the nodes are stored level by level, starting from the zero level where only the root node is present.



A[1]	1
A[2]	2
A[3]	3
A[4]	4
A[5]	5
A[6]	6
A[7]	7

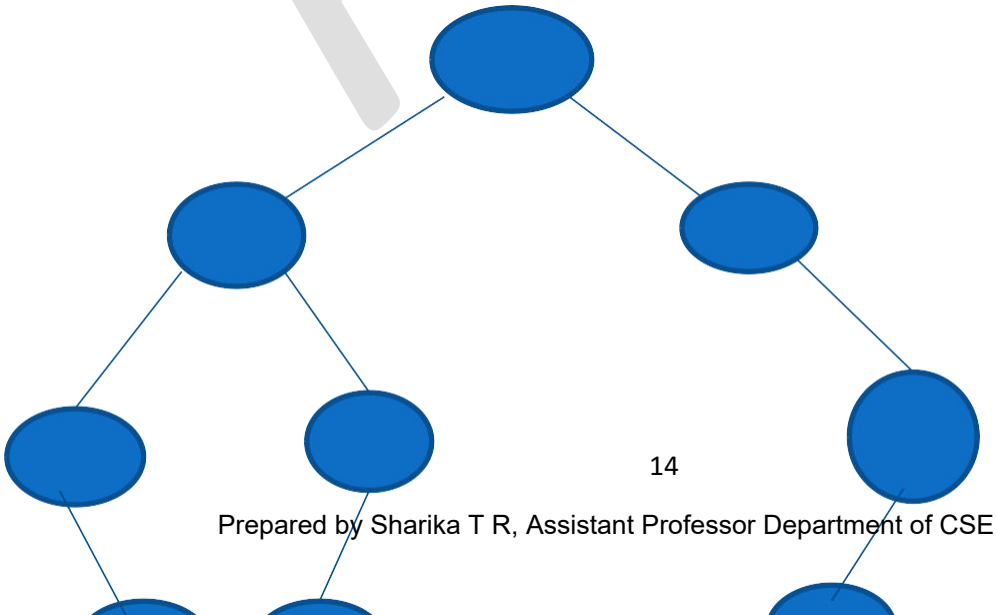
Following rules can be used to decide the location of any node

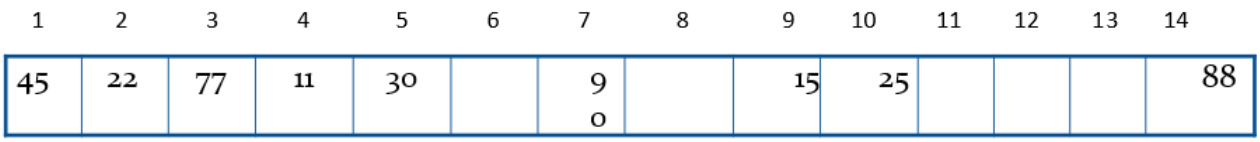
1. The root node is at index 1
2. For any node with index I,
    1. Parent (i) is at i /2
    2. Left child of i is at 2i

Right child of i is at 2i + 1

Left child of i is at 2i :

- If 2i > n , then the node does not have a left child.
- Eg: i=4 , then 2i = 8, but 8>n
- no left child Right child of i is at 2i +1:
- If 2i + 1 > n , then the node does have a right child.
- Eg: i=4 , then 2i+1 = 9, but 9>n
- no right child





Advantages of linear representation

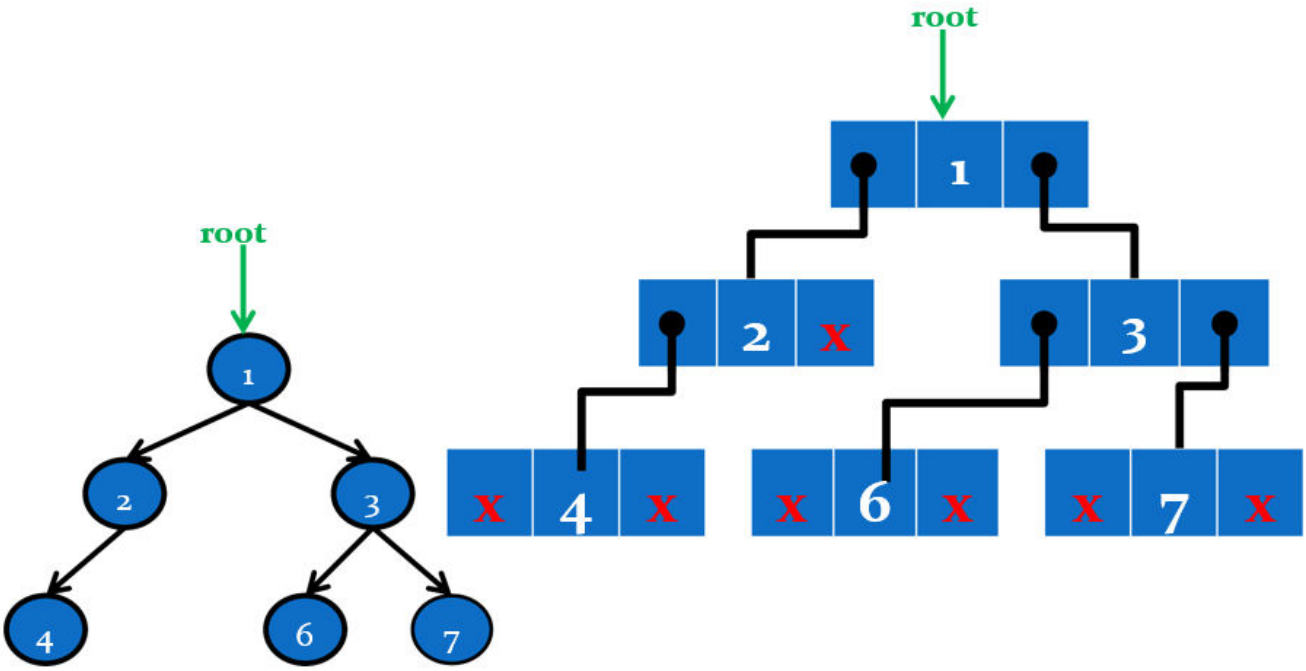
- Any node can be accessed by calculating the index
- For each node, only data is stored – no need to store any pointers

Disadvantages

- Other than for full binary tree, majority of the array entries may be empty
- No way to increase the size of the tree during execution
- Inserting or deleting a node is inefficient

2. **Linked or pointer representation**

- Tree can also be defined as a finite collection of nodes where each node is divided into 3 parts containing
  - Left child address
  - Information (data)
  - Right child address



Operations on binary tree

- Insertion as a leaf node
- Deletion of a leaf node
- Traversal

Insertion as a leaf node

Algorithm: InsertLeaf(x, k)

Input:

- x: Value of the node to be inserted.

- k: Key value used to search for the insertion position.

Steps

1. l <- Search(k)
   - Search(k) function:
     - Returns the index of the node with key k if found, otherwise returns -1.

2. If l == -1:
   - Print "Search is unsuccessful"
  Else:
   a. Read option to insert as left child (LC) or right child (RC).
   b. If option == RC:
      - If a[(2 * l) + 1] == NULL:
         - a[(2 * l) + 1] <- x
      - Else:
         - Print "Insertion not possible"
      - EndIf
   c. Else if option == LC:
      - If a[(2 * l)] == NULL:
         - a[(2 * l)] <- x
      - Else:
         - Print "Insertion not possible"
      - EndIf
   d. EndIf
  EndIf

3. Search a key

Algorithm: BinaryTreeSearch(k)

Input:

- k: Key value to be searched in the binary tree.

Output:

- Index of the node with key k if found, otherwise -1.

Steps

1. For i <- 1 to size:
   a. If a[i] == k:
      - Return i
   b. EndIf
2. EndFor
3. Return -1

4. Deletion of a leaf node

Algorithm: DeleteLeaf(x)

Input:

- x: Value of the node to be deleted.

Steps

1. l <- Search(x)

- Search(x) function:
  - Returns the index of the node with value x if found, otherwise returns -1.

2. If l != -1:
   a. If a[(2 * l)] == NULL && a[(2 * l) + 1] == NULL:
      - Delete a[l]
      - Set a[l] <- NULL
   b. Else:
      - Print "Item is not a leaf node"
   c. EndIf
  Else:
   - Print "Item does not exist"
  EndIf

Insertion of leaf node in a linked list representation

Here's the algorithm for inserting a leaf node in a binary tree represented using a linked list:

Algorithm: InsertLeaf(root, value)

Input:

- root: Pointer to the root node of the binary tree.
- value: Value of the node to be inserted as a leaf node.

Output: Binary tree with the new node inserted as a leaf node.

Steps

1. Create a new node with the given value:
   a. newNode <- CreateNode(value)
      - CreateNode(value) function:
        - Allocate memory for a new node.
        - Set newNode.data <- value.
        - Set newNode.left <- NULL.
        - Set newNode.right <- NULL.
        - Return newNode.
2. If root is NULL:
   - Set root <- newNode
   - Exit
3. Create a queue for level-order traversal:
  - Initialize an empty queue.
4. Enqueue the root node into the queue.
5. While the queue is not empty:
   - Dequeue a node from the front of the queue.
   - If the dequeued node has no left child:
     - Set newNode as its left child and break.
   - Else if the dequeued node has no right child:
     - Set newNode as its right child and break.
   - Else:

- Enqueue both the left and right children of the dequeued node (if they exist).
6. Return the modified binary tree.

This algorithm performs a level-order traversal of the binary tree using a queue. At each level, it checks if the dequeued node has any empty child slots. If so, it inserts the new node as a leaf node in the first available empty slot. If both slots are filled, it continues the traversal until an appropriate position for insertion is found. Finally, it returns the modified binary tree with the new node inserted as a leaf node.

Searching in a linked list representation

Algorithm: Search(root, value)

Input:

- root: Pointer to the root node of the binary tree.

- value: Value to be searched in the binary tree.

Output:

- True if the value is found in the binary tree, False otherwise.

Steps

1. If root is NULL:
   - Return False
2. Create a queue for level-order traversal:
   - Initialize an empty queue.
3. Enqueue the root node into the queue.
4. While the queue is not empty:
   - Dequeue a node from the front of the queue.
   - If the dequeued node's data is equal to the value:
     - Return True
   - Enqueue the left and right children of the dequeued node (if they exist).

5. If the value is not found after traversing the entire binary tree:
   - Return False

Deleting a leaf node from a linked list representation

Algorithm: DeleteLeaf(root, value)

Input:

- root: Pointer to the root node of the binary tree.
- value: Value of the leaf node to be deleted.

Output:

- Binary tree with the specified leaf node deleted.

Steps

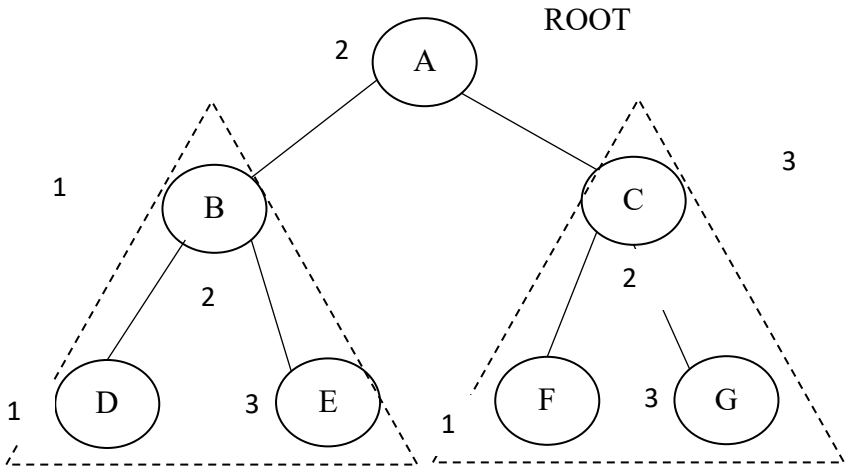
1. If root is NULL:
   - Return root
2. If root is a leaf node and its data matches the value:
   - Deallocate memory for root
   - Set root to NULL
   - Return NULL
3. Create a queue for level-order traversal:
   - Initialize an empty queue.
4. Enqueue the root node into the queue.
5. While the queue is not empty:
   - Dequeue a node from the front of the queue.
   - If the dequeued node's left child is a leaf node and its data matches the value:
     - Deallocate memory for the left child
     - Set the left child of the dequeued node to NULL
     - Return root
   - If the dequeued node's right child is a leaf node and its data matches the value:
     - Deallocate memory for the right child
     - Set the right child of the dequeued node to NULL
     - Return root
   - Enqueue the left and right children of the dequeued node (if they exist).
6. If the value is not found after traversing the entire binary tree:
   - Return root

Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

In-order Traversal : In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.



D → B → E → A → F → C → G

Algorithm

Until all nodes are traversed –

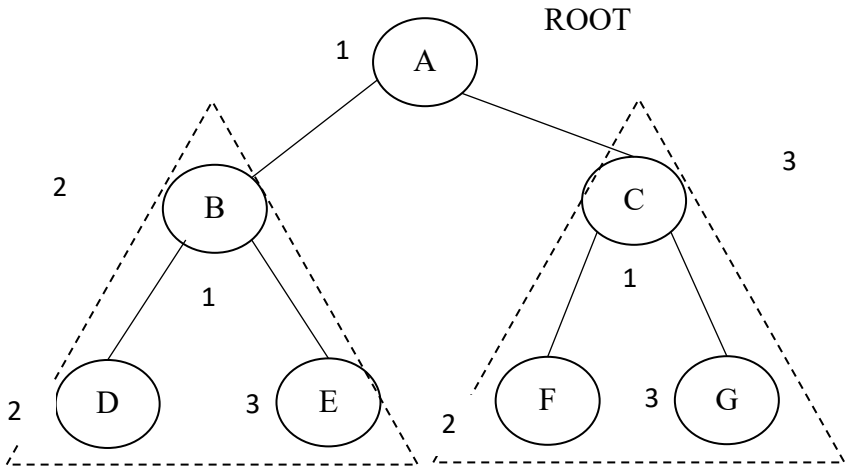
1. Current=root
2. If(current!=NULL)
    1. INORDER(current.llink)
    2. VISIT(current)
    3. INORDER(current.rlink)
3. EndIf
4. Stop

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited.

The output of pre-order traversal of this tree will be



– A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed –

Steps:

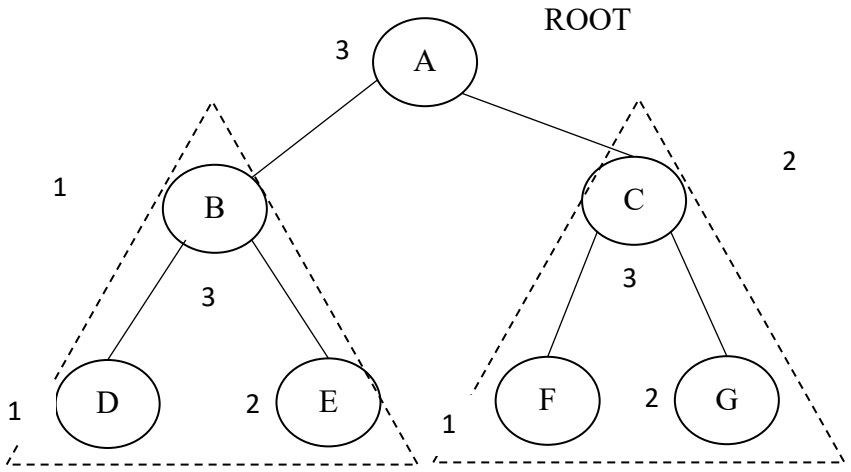
1. Current=root
2. If(current!=NULL)
    1. VISIT(current)
    2. PREORDER(current.llink)
    3. PREORDER(current.rlink)
3. EndIf

StopPost-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited.

The output of post-order traversal of this tree will be



– D → E → B → F → G → C → A

Algorithm

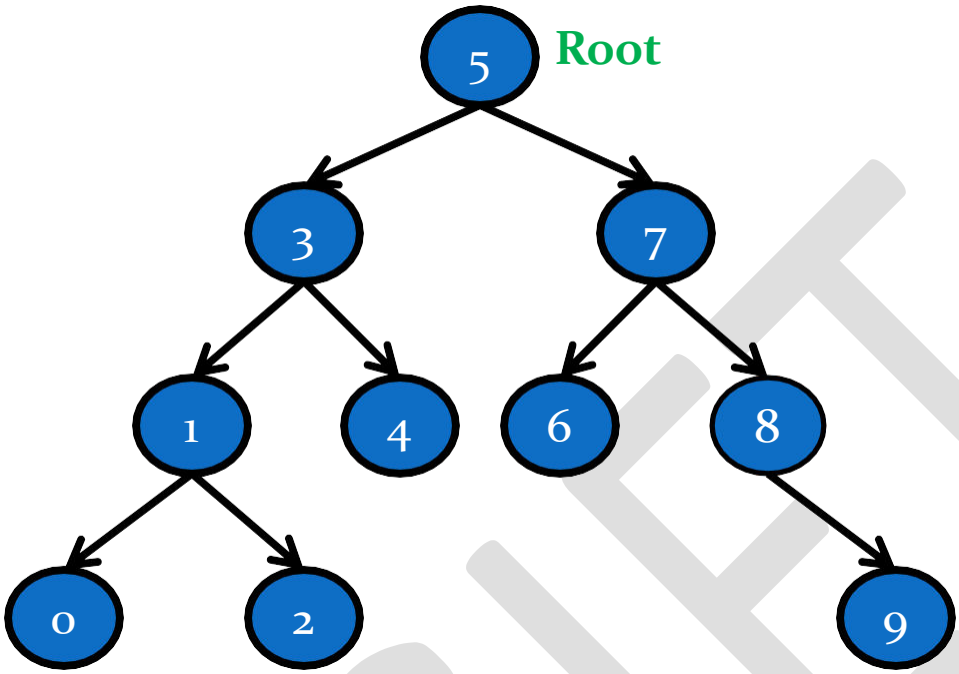
Until all nodes are traversed –

Algorithm

1. Current=root
2. If(current!=NULL)
    1. POSTORDER(current.llink)
    2. POSTORDER(current.rlink)
    3. VISIT(current)
3. EndIf
4. Stop

Question

Write the preorder, inorder and postorder traversal of



Answer.

PREORDER = 5 3 1 0 2 4 7 6 8 9

INORDER= 0 1 2 3 4 5 6 7 8 9 (sorted order)

POSTORDER= 0 2 1 4 3 6 9 8 7 5

Construct tree from the inorder and preorder traversals

To construct a binary tree from its inorder and preorder traversals, we can follow these steps recursively:

- Pick the first element from the preorder traversal array. This will be the root of the current subtree.
- Search for the index of this root element in the inorder traversal array. This index divides the inorder array into left and right subtrees.
- Recursively build the left subtree using the left portion of the inorder array and the corresponding portion of the preorder array.

- Recursively build the right subtree using the right portion of the inorder array and the corresponding portion of the preorder array.
- Return the root node of the subtree.

Algorithm: ConstructBinaryTree(inorder, preorder)

Input:

- inorder: Array representing the inorder traversal of the binary tree.

- preorder: Array representing the preorder traversal of the binary tree.

Output:

- Root node of the constructed binary tree.

Steps

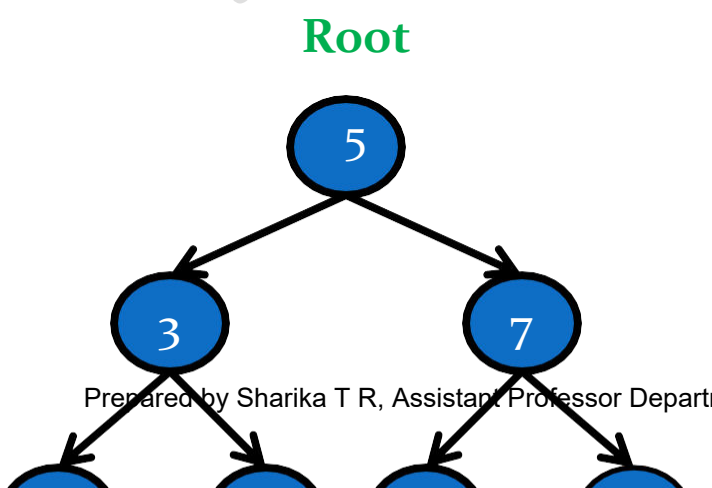
1. If inorder is empty or preorder is empty:
   - Return NULL
2. EndIf
3. Set rootValue <- First element of preorder array
4. Create a new TreeNode with value rootValue
5. Find the index of rootValue in the inorder array and store it in index
6. Set root.left <- ConstructBinaryTree(left portion of inorder, left portion of preorder)
   - This recursively constructs the left subtree
7. Set root.right <- ConstructBinaryTree(right portion of inorder, right portion of preorder)
   - This recursively constructs the right subtree
8. Return root

Question

Construct the tree from the following inorder and postorder traversals

POSTORDER = 0 2 1 4 3 6 9 8 7 5

INORDER= 0 1 2 3 4 5 6 7 8 9

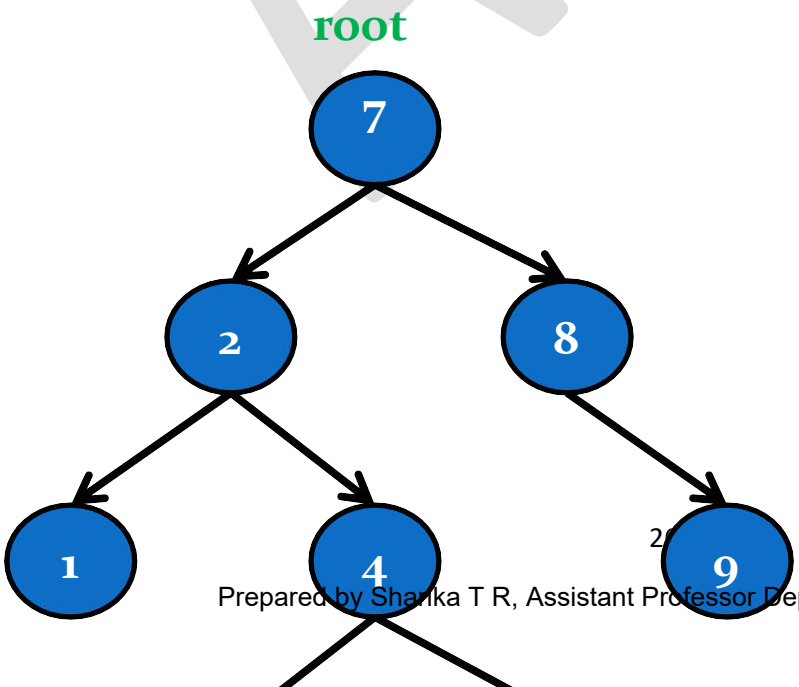


Binary Search Tree

Indeed, a binary search tree (BST) is a type of binary tree with specific properties:

1. It may be empty, meaning it contains no elements.
2. If it's not empty, each element has a key, and no two elements in the tree have the same key (no duplicates).
3. For any given node in the tree, all keys in its left subtree are smaller than the key of the node itself.
4. Similarly, for any given node in the tree, all keys in its right subtree are larger than the key of the node itself.
5. Both the left and right subtrees of any node in the tree are also binary search trees themselves, satisfying the same properties.

These properties ensure that a BST maintains an ordered structure, making it efficient for searching, inserting, and deleting elements based on their keys. The binary search tree's recursive nature allows for efficient operations by exploiting the ordered arrangement of its elements.



Binary Search Tree Insertion

Here's a structured algorithm for inserting a node with value item into a binary search tree (BST):

Algorithm: InsertBST(root, item)

Input:

- root: Pointer to the root node of the binary search tree.

- item: Value of the node to be inserted.

Output:

- Binary search tree with the new node inserted.

Steps

1. Create a new node:
   a. newNode <- CreateNode(item)
      - CreateNode(item) function:
        - Allocate memory for a new node.
        - Set newNode.data <- item.
        - Set newNode.left <- NULL.
        - Set newNode.right <- NULL.
        - Return newNode.
2. If root is NULL:
   - Set root <- newNode
   - Return root
3. Set current <- root
4. Set parent <- NULL
5. While current is not NULL:
   a. Set parent <- current
   b. If item > current.data:
      - Set current <- current.right
   c. Else:
      - Set current <- current.left

d. EndIf
6. EndWhile
7. If item > parent.data:
   - Set parent.right <- newNode
   Else:
   - Set parent.left <- newNode
8. EndIf
9. Return root

This algorithm inserts a new node with value item into a binary search tree by traversing the tree from the root until finding the appropriate position for insertion. It then attaches the new node to the appropriate position based on the value of item. Finally, it returns the root of the modified binary search tree.

Binary Search Tree Insertion

Here's the provided algorithm BST_INSERT(item) rewritten for clarity and readability:

Algorithm: BST_INSERT(item)

Input: Item is the data component of a node that has to be inserted.

Output: Item inserted into the proper place of the tree.

Data structure: Linked structure

Steps

1. Create a new node:
   a. Newnode <- malloc(newnode)
   b. Newnode.data <- item
   c. Newnode.left <- NULL
   d. Newnode.right <- NULL

2. If root is NULL:
   a. Set root <- Newnode
   b. Exit
3. Set current <- root
4. Set parent <- NULL
5. While current is not NULL:
   a. Set parent <- current
   b. If item > current.data:
      - Set current <- current.right
   c. Else If item < current.data:
      - Set current <- current.left
   d. Else:
      - Print "Duplicate node"
      - Exit

e. EndIf
6. EndWhile
7. If item > parent.data:
- Set parent.right <- Newnode
Else:
- Set parent.left <- Newnode
8. EndIf
9. Stop

This rewritten algorithm maintains the functionality of inserting a new node with data item into a binary search tree (BST). It creates a new node, traverses the tree to find the appropriate insertion point, and attaches the new node accordingly. Additionally, it handles cases of duplicate nodes by printing a message and exiting the algorithm.

Searching in a binary search tree

Searching for a data in a binary search tree is much faster than in arrays or linked lists. So the applications where frequent searching operations are to be performed, this data structure is used. To search for an item

1. start from the root node,
2. if item < root node data
    a. proceed to its left child.
3. If item > root node data
    a. proceed to its right child.
4. Continue step 2 and 3 till the item is found or we reach to a dead end (leaf node)

Here's the provided algorithm BST_SEARCH(item):

Algorithm: BST_SEARCH(item)

Input: item is the data to be searched.

Output: Pointer to the node containing the data if found, NULL otherwise.

Data Structure: Linked Structure.

Steps

1. Set current <- root
2. While current is not NULL and current.data != item:
a. If item > current.data:
- Set current <- current.right
b. Else:
- Set current <- current.left
c. EndIf
3. EndWhile
4. If current is NULL:

- Print "Item not found"
- Exit

5. Else:
   - Print "Item Found"
6. EndIf
7. Return current

Deleting a node from a binary search tree

Suppose

T→binary search tree

ITEM → data to be deleted from T if it exists in the tree.

N → node which contains the information ITEM.

PARENT(N) →the parent node of N

SUCC(N) →the inorder successor of node N.

Then the deletion of the node N depends on the number of children for node N. Hence, three cases may arise:

Case 1: N is a leaf node

Case 2: N has exactly one child

Case 3: N has two Childs.

Algorithm: DeleteBST(root, item)

Input:

- root: Pointer to the root node of the binary search tree.
- item: Data of the node to be deleted.

Output:

- Binary search tree with the specified node deleted.

Steps

1. Search for the node with data item in the BST:
   a. Set current <- root
   b. While current is not NULL and current.data != item:
      - If item > current.data:
        - Set current <- current.right
      - Else:
        - Set current <- current.left
      - EndIf
   c. EndWhile
2. If current is NULL:
   - Print "Item not found"
   - Exit
3. Else:
   a. Case 1: N is a leaf node:
      - Delete current node (N)
      - Set parent's appropriate child pointer to NULL
   b. Case 2: N has exactly one child and it's a left child:
      - Adjust parent's left child pointer to N's child
      - Delete current node (N)
   c. Case 3: N has exactly one child and it's a right child:
      - Adjust parent's right child pointer to N's child
      - Delete current node (N)
   d. Case 4: N has two children:
      - Find the inorder successor or predecessor of N (e.g., inorder successor)
      - Copy the data of the successor/predecessor to N
      - Recursively delete the successor/predecessor node
   e. EndIf
4. Stop

Deleting a node from a binary search tree

1. Case 1: N is a leaf node

    1. N is the root

    2. N is the left child of its parent

    3. N is the right child of its parent

2. Case 2: N has exactly one child and it’s a left child

    1. N is the root

    2. N is the left child of its parent

3. N is the right child of its parent

3. Case 3: N has exactly one child and it's a right child

1. N is the root

2. N is the left child of its parent

3. N is the right child of its parent

4. Case 4: N has two Childs.

Algorithm: BST_DELETE(item)

Input:

- item: Data to be removed from the binary search tree (BST).

Output:

- If the node with data as item exists, it is deleted; otherwise, a message is printed.

Data Structure:

- Linked structure of binary tree, where root points to the root node.

Steps:

1. Set Flag <- 0
2. Set Parent <- NULL
3. If root is NULL:
   a. Print "Tree is Empty"
   b. Exit
   EndIf
4. Set Current <- root
5. While current is not NULL:
   a. If current.data equals item:
      - Set Flag <- 1
      - ExitWhile
      Else:
      - Set Parent <- current
      - If item > current.data:

```
            - Set current <- current.right
          Else:
            - Set current <- current.left
        - EndIf
    b. EndIf
  EndWhile
6. If Flag is 0:
    a. Print "No item found in the tree"
    b. Exit
  EndIf

7. If current.left is NULL and current.right is NULL:
    a. If parent is NULL:
        - Free current
        - Set root <- NULL
      ElseIf parent.left equals current:
        - Set parent.left <- NULL
        - Free current
      ElseIf parent.right equals current:
        - Set parent.right <- NULL
        - Free current
      EndIf
8. ElseIf current.left is NULL and current.right is not NULL:
    a. If parent is NULL:
        - Set root <- root.right
        - Free current
      ElseIf parent.left equals current:
        - Set parent.left <- current.right
        - Free current
      ElseIf parent.right equals current:
        - Set parent.right <- current.right
        - Free current
      EndIf
9. EndIf
```

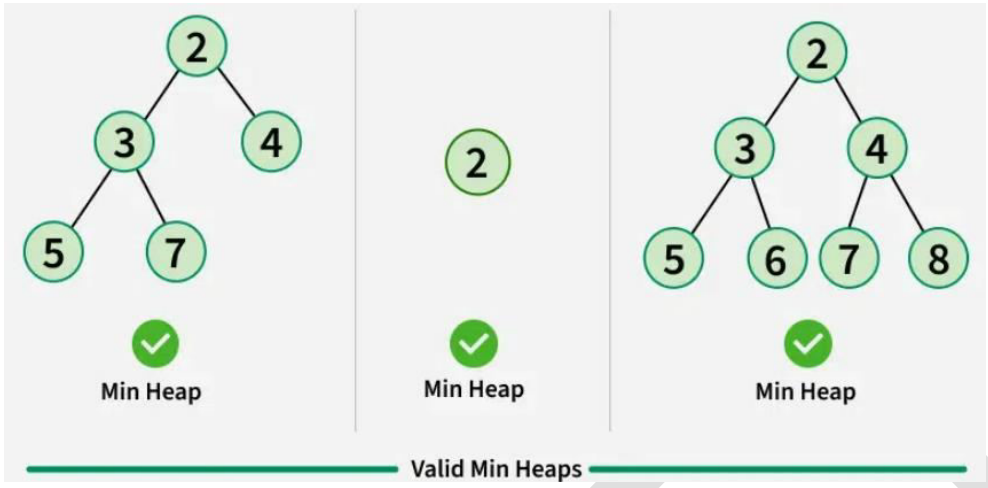
Binary Heap

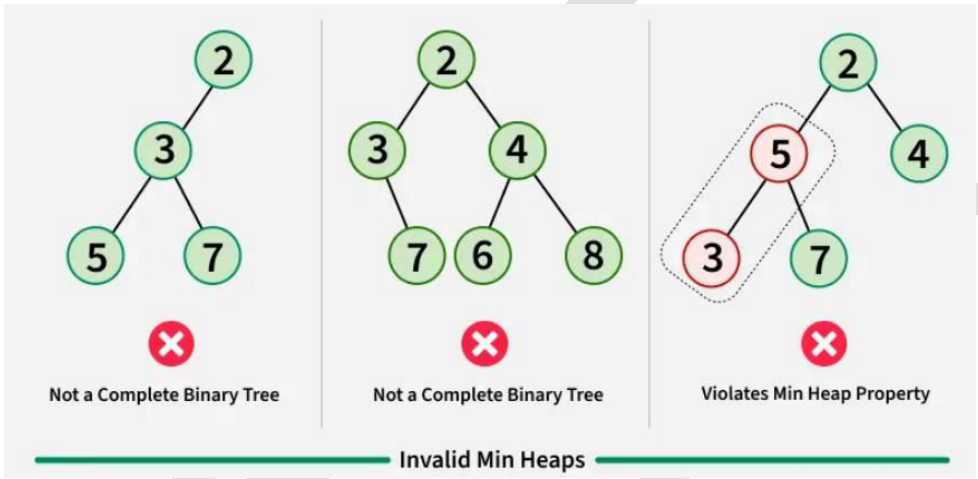
A Binary Heap is a special type of complete binary tree, meaning all levels are filled except possibly the last, which is filled from left to right.

It allows fast access to the minimum or maximum element. There are two types of binary heaps: Min Heap and Max Heap.

- Min Heap: The value of the root node is the smallest, and this property is true for all subtrees.
- Max Heap: The value of the root node is the largest, and this rule also applies to all subtrees.

- Binary heaps are commonly used in priority queues and heap sort algorithms because of their efficient insertion and deletion operations.



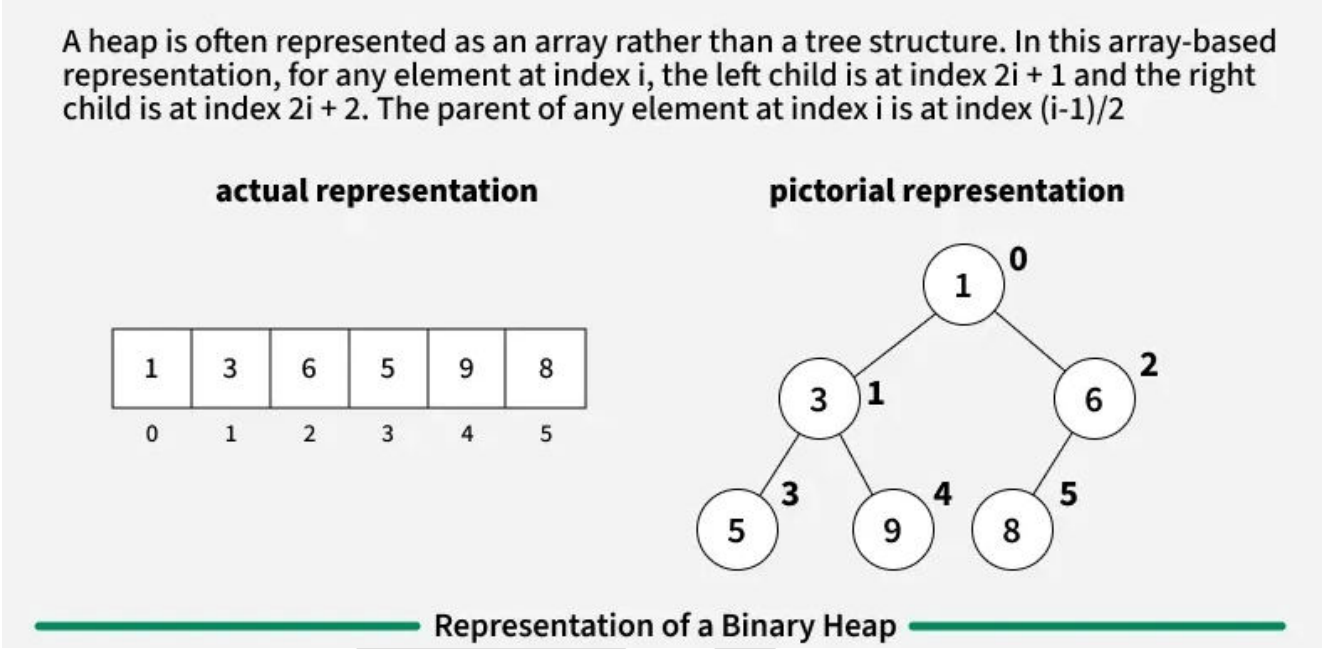


How is Binary Heap represented?
A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.

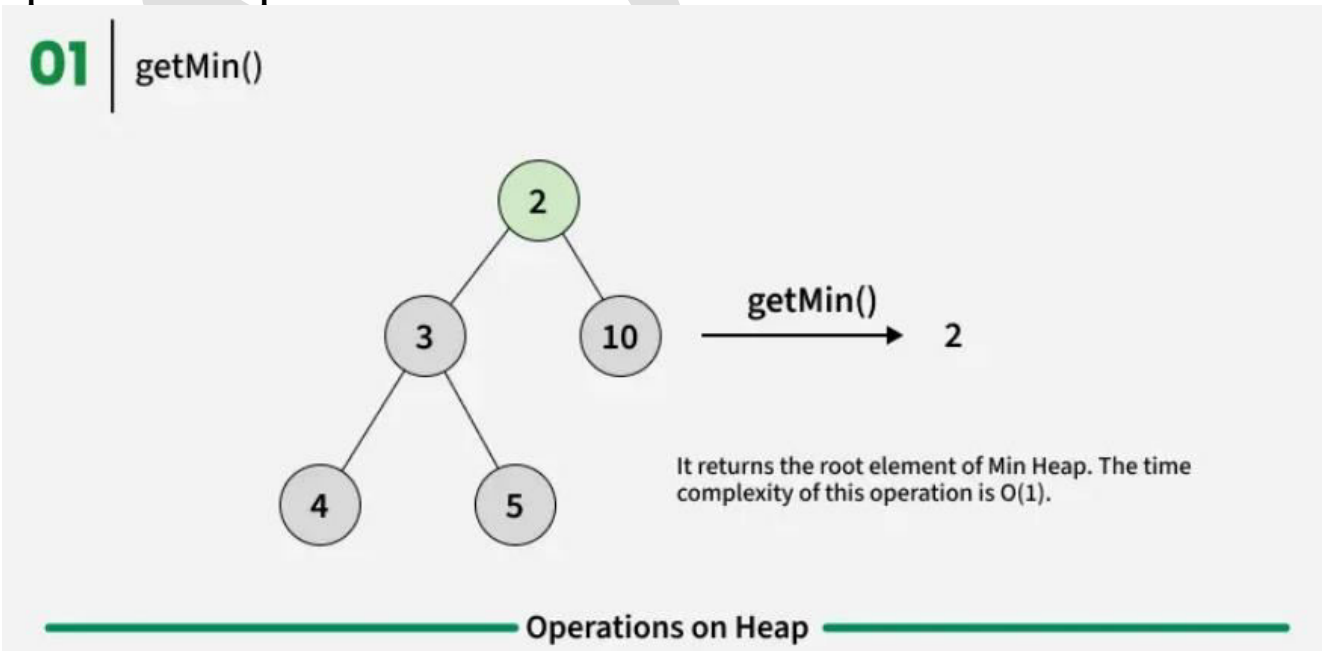
The root element will be at arr[0].
The below table shows indices of other nodes for the ith node, i.e., arr[i]:

arr[(i-1)/2]	Returns the parent node
arr[(2*i)+1]	Returns the left child node
arr[(2*i)+2]	Returns the right child node

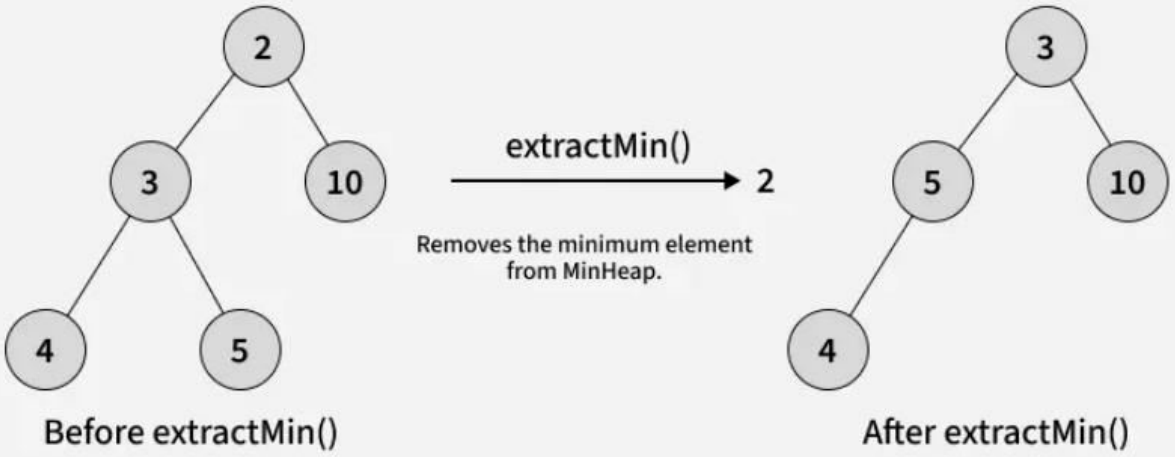
The traversal method use to achieve Array representation is Level Order Traversal.



Operations on Heap

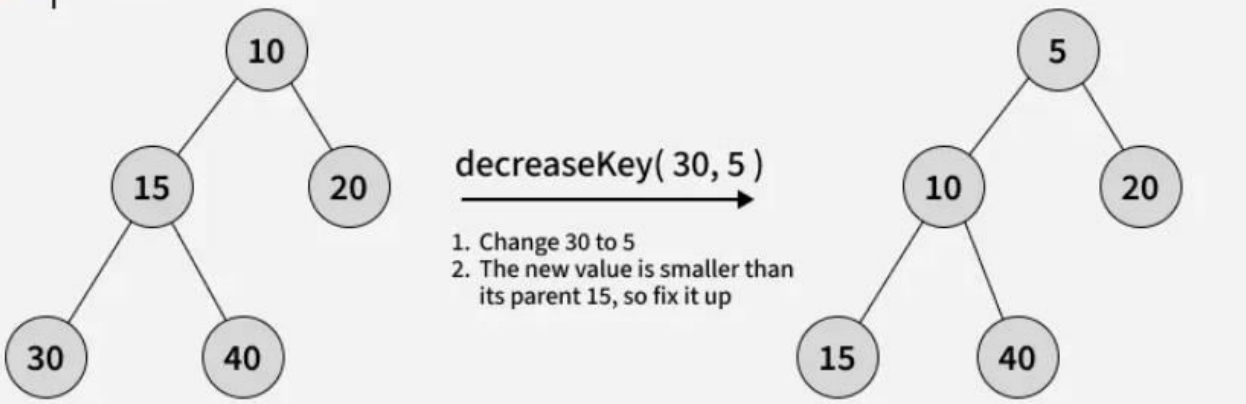


02 extractMin()



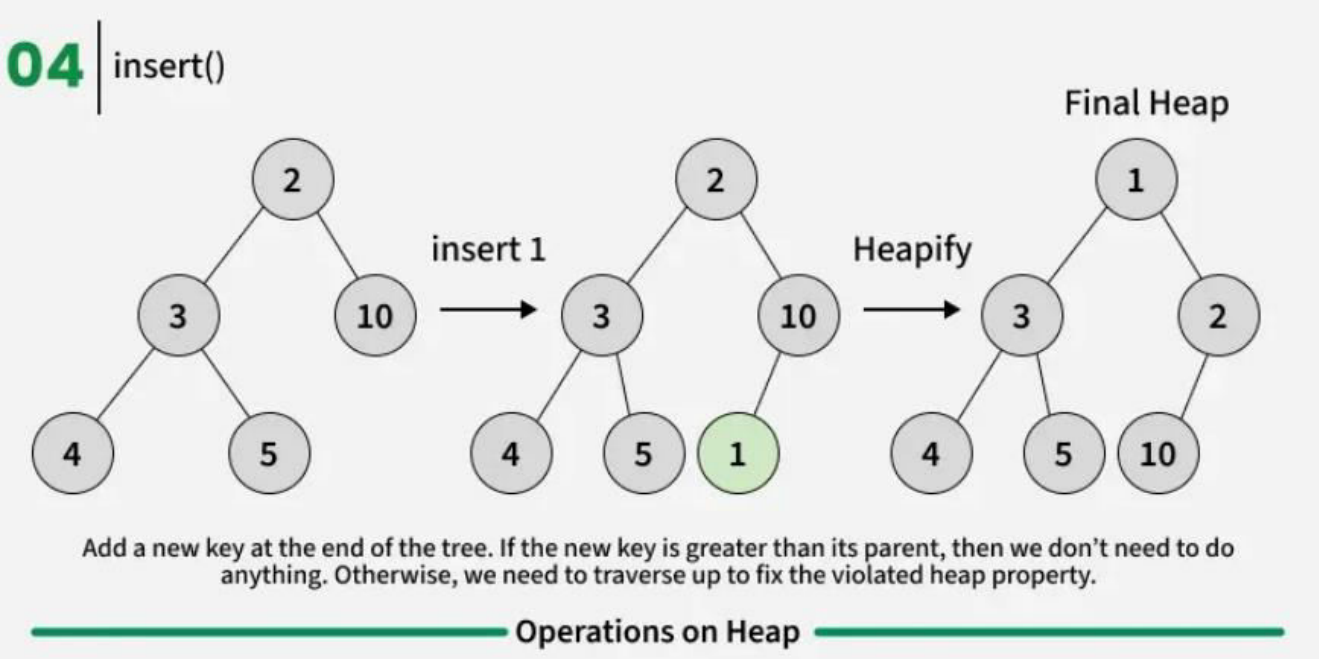
Operations on Heap

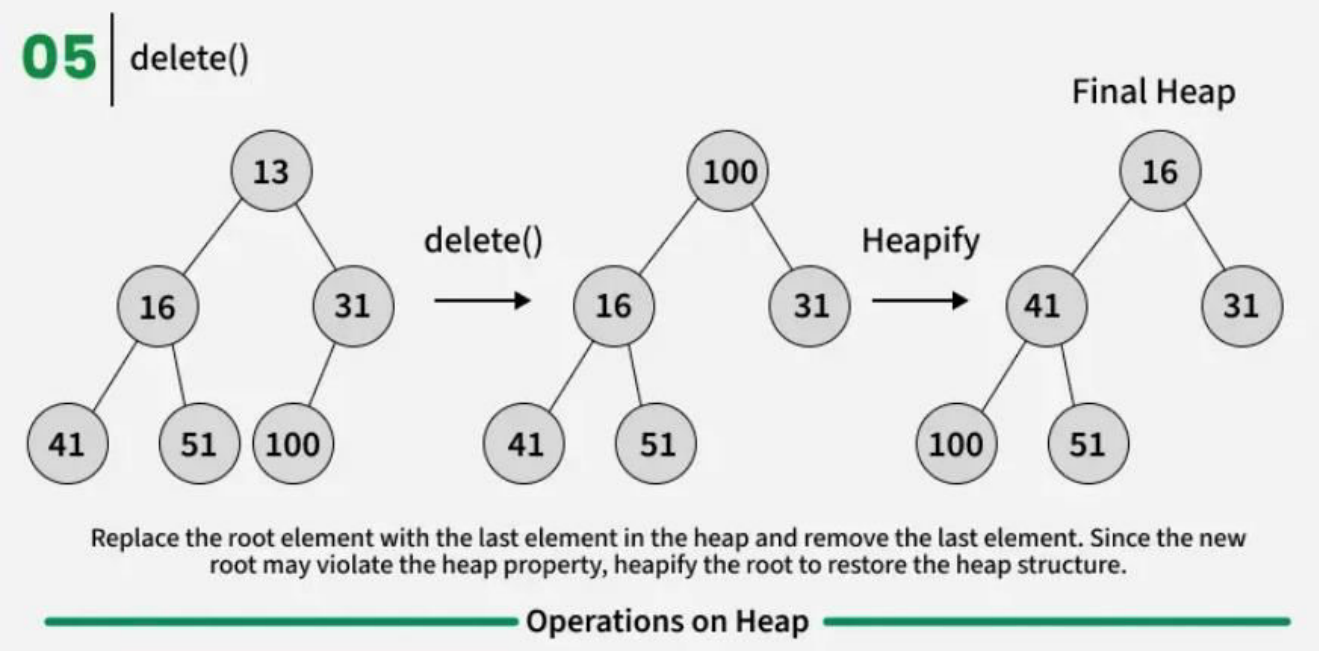
03 decreaseKey()



Decrease the value of the key. If the decreased key value of a node is greater than the parent of the node, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

Operations on Heap





Applications of Heaps

- Heap Sort: Heap Sort uses Binary Heap to sort an array in O(nLogn) time.
- Priority Queue: Priority queues are efficiently implemented using Binary Heaps, which allow operations like insert, delete, extractMax, and decreaseKey in O(log N) time. Binomial and Fibonacci Heaps are advanced types that also support fast union operations.
- Graph Algorithms: The priority queues are especially used in Graph Algorithms like Dijkstra's Shortest Path and Prim's Minimum Spanning Tree.
- Many problems can be efficiently solved using Heaps.

Graph

Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follows...

Graph is a collection of vertices and arcs in which vertices are connected with arcs

Graph is a collection of nodes and edges in which nodes are connected with edges

Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**.

Example

The following is a graph with 5 vertices and 6 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),(E,D)}.

Graph Terminology

Path - A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Edge -An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

Undirected Edge - An undirected egde is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

Directed Edge - A directed egde is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

Weighted Edge - A weighted egde is a edge with value (cost) on it.Vertex -Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

End vertices or Endpoints - The two vertices joined by edge are called end vertices (or endpoints) of that edge.

Origin - If a edge is directed, its first endpoint is said to be the origin of it.

Destination - If a edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.

Adjacent - If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.

Incident - Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge - A directed edge is said to be outgoing edge on its origin vertex.

Incoming Edge - A directed edge is said to be incoming edge on its destination vertex.

Degree - Total number of edges connected to a vertex is said to be degree of that vertex.

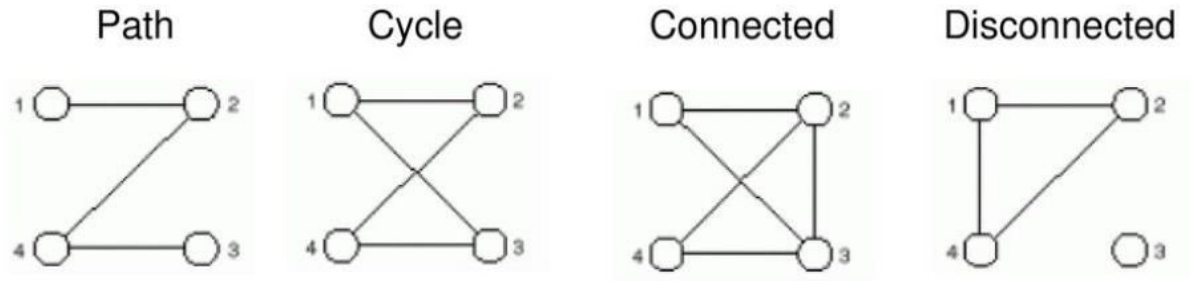
Indegree - Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree - Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Closed Path -A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0=V_N$.

Simple Path - If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simple path.

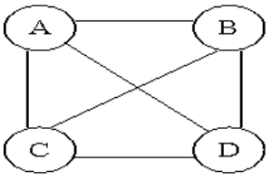
Cycle -A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.



Types of Graph

1.Undirected Graph

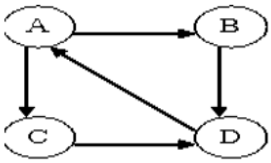
A graph with only undirected edges is said to be undirected graph.



Undirected Graph.

2.Directed Graph

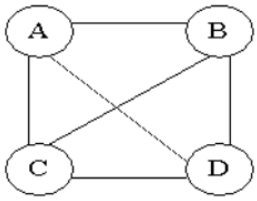
A graph with only directed edges is said to be directed graph.



Directed Graph.

3.Complete Graph

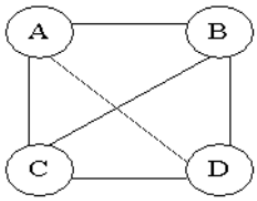
A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in the graph. The following figure shows a complete graph.



A complete graph.

4.Regular Graph

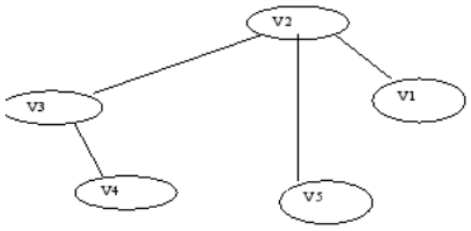
Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.



A regular graph

6.Acyclic Graph

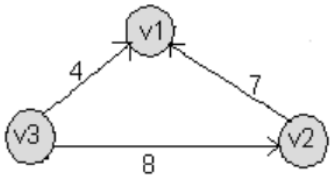
A graph without cycle is called acyclic graphs.



A acyclic graph

7. Weighted Graph

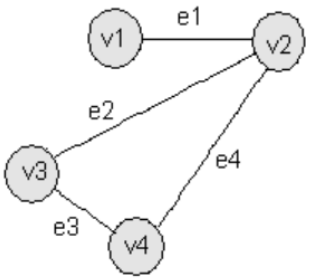
A graph is said to be weighted if there are some non negative value assigned to each edges of the graph. The value is equal to the length between two vertices. Weighted graph is also called a network.



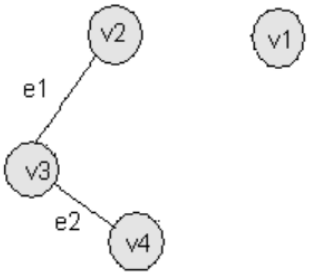
A weighted graph

Connected Graph

A graph G is said to be connected if there is at least one path between every pair of vertices in G. Otherwise,G is disconnected.



A connected graph G



A disconnected graph G

Graph Representations

Graph data structure is represented using following representations...

Adjacency Matrix

Incidence Matrix

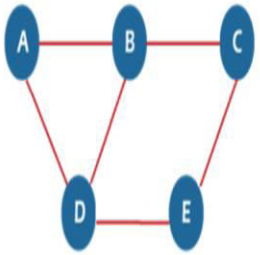
Adjacency List

Adjacency Matrix

The Adjacency Matrix is a V×V matrix where the values are filled with either 0 or 1. If the link exists between Vi and Vj, it is recorded 1; otherwise, 0.

If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.

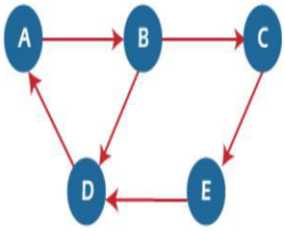
An entry A_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between V_i and V_j.



Undirected Graph

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

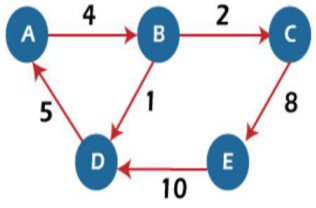
Adjacency Matrix



Directed Graph

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

Adjacency Matrix



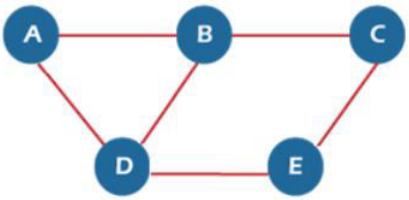
weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

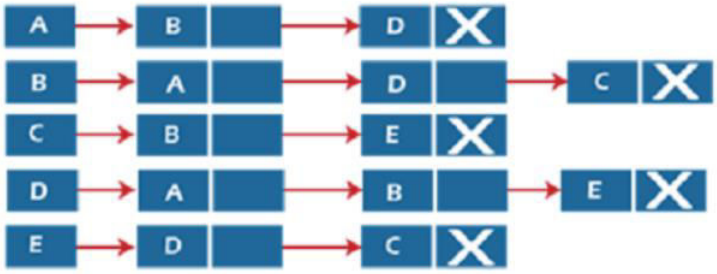
Adjacency Matrix

Adjacency List

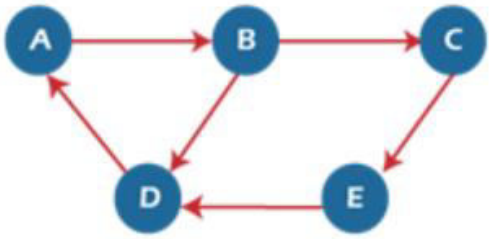
The adjacency list is a list of the vertices directly connected to the other vertices in the graph.



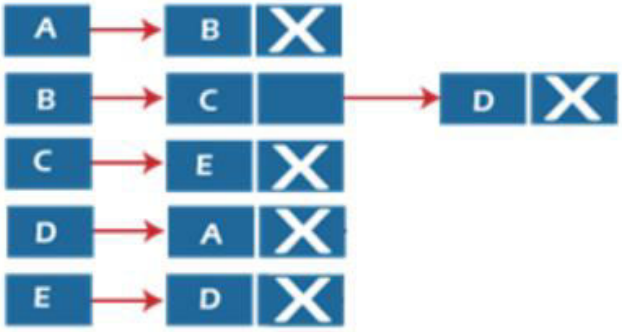
Undirected Graph



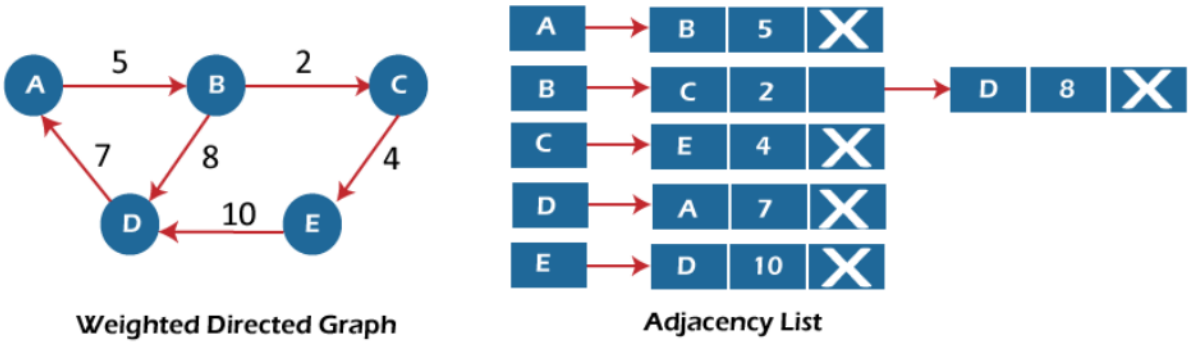
Adjacency List



Directed Graph



Adjacency List



Graph Traversal - DFS

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

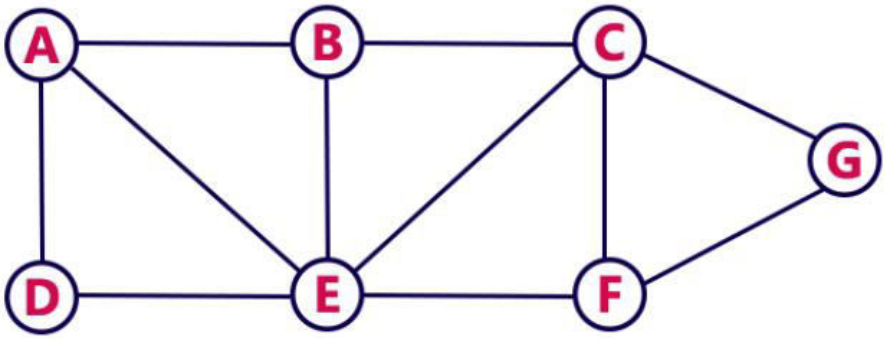
BFS (Breadth First Search)

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

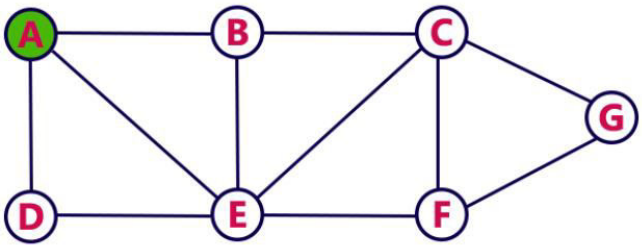
- **Step 1 -** Define a Queue of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3 -** Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5 -** Repeat steps 3 and 4 until queue becomes empty.
- **Step 6 -** When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

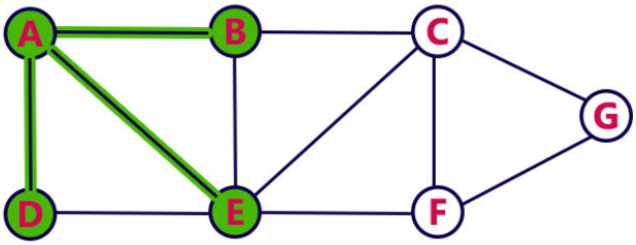


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

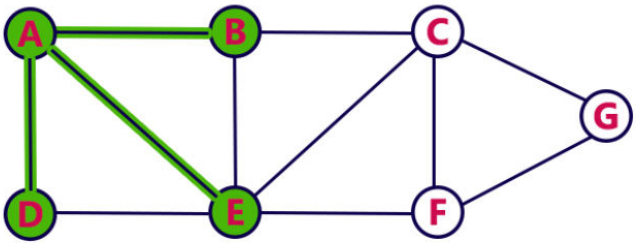


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

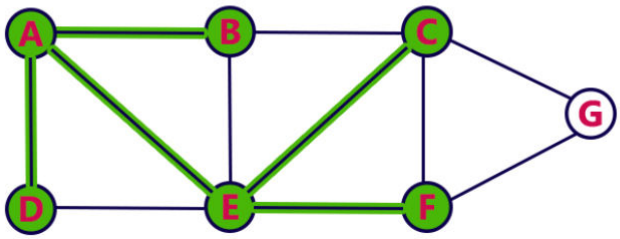


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

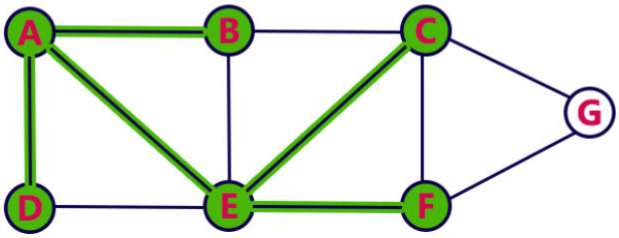


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

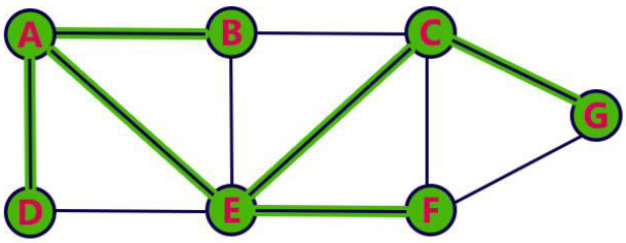


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

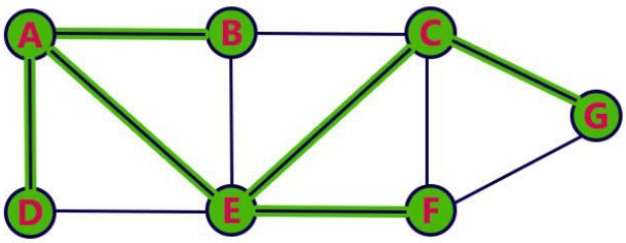


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

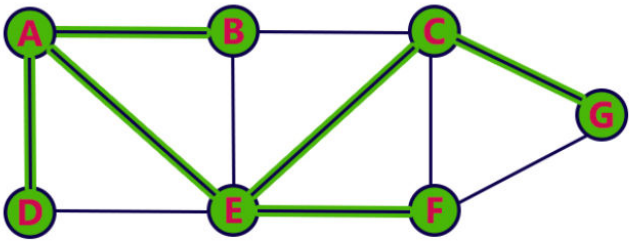


Queue

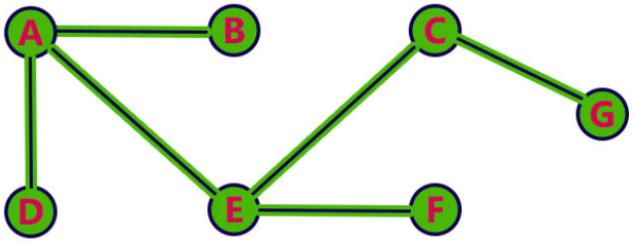
						G

Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



DFS (Depth First Search)

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

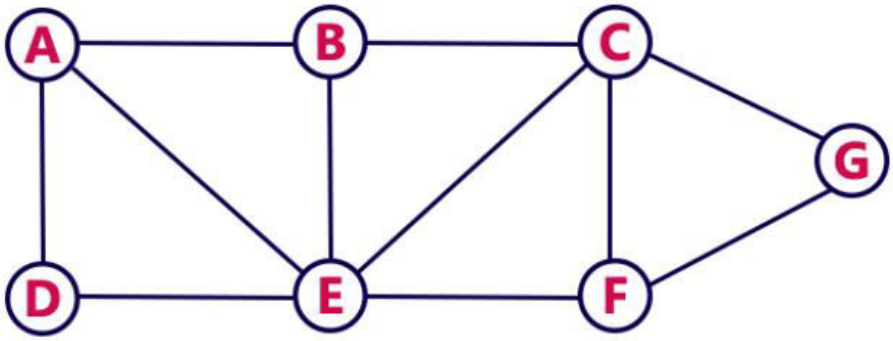
We use the following steps to implement DFS traversal...

- **Step 1 -** Define a Stack of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3 -** Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
- **Step 4 -** Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5 -** When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.
- **Step 6 -** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7 -** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

---

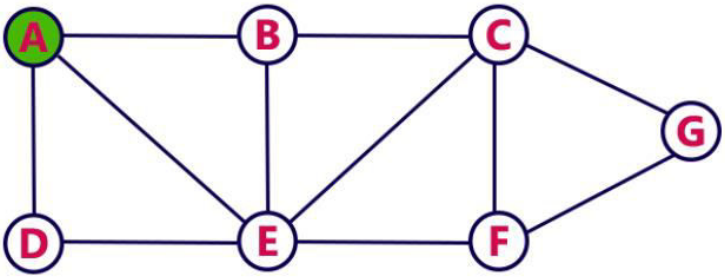
Back tracking is coming back to the vertex from which we reached the current vertex.

Consider the following example graph to perform DFS traversal



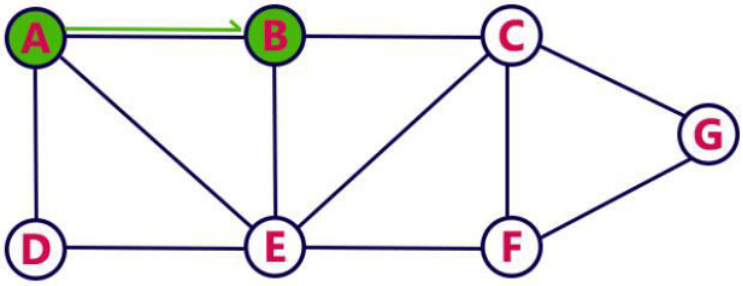
Step 1:

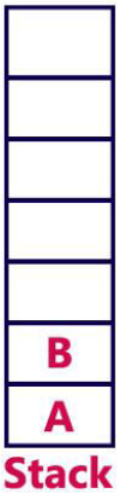
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



Step 2:

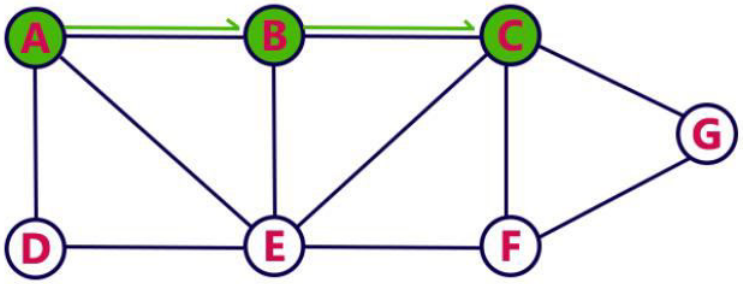
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

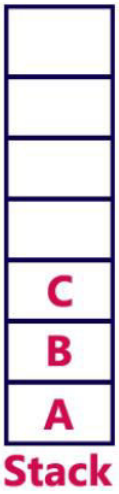




Step 3:

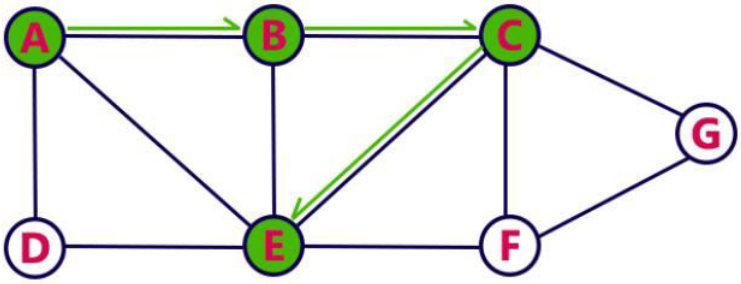
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

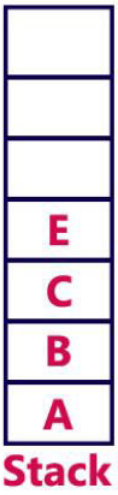




Step 4:

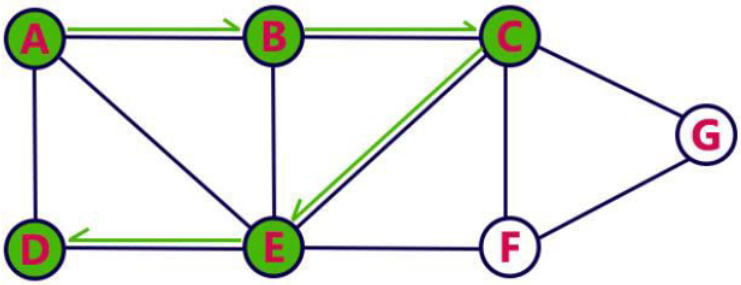
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

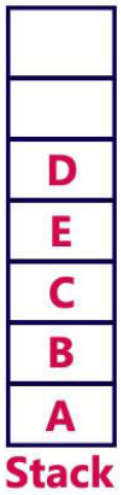




Step 5:

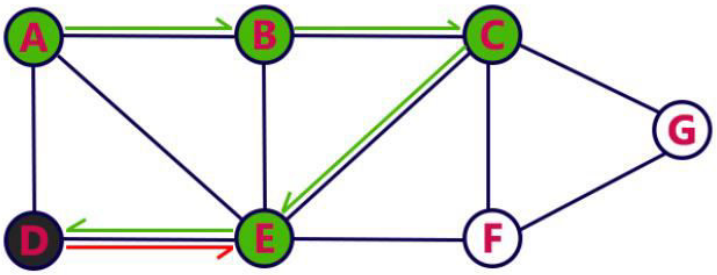
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack





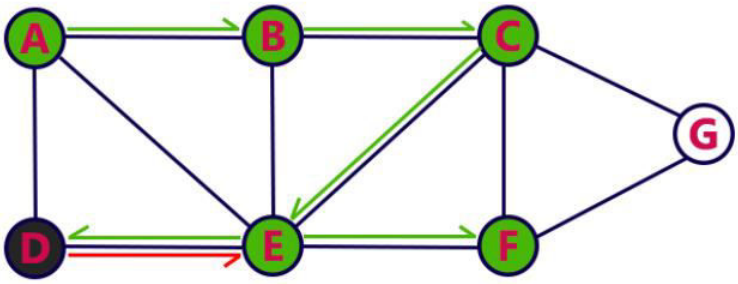
Step 6:

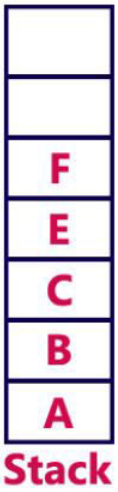
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



Step 7:

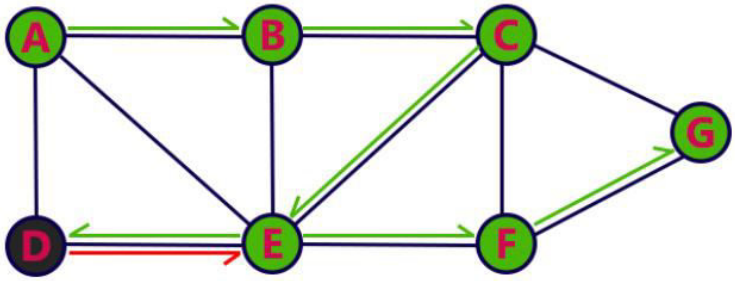
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.





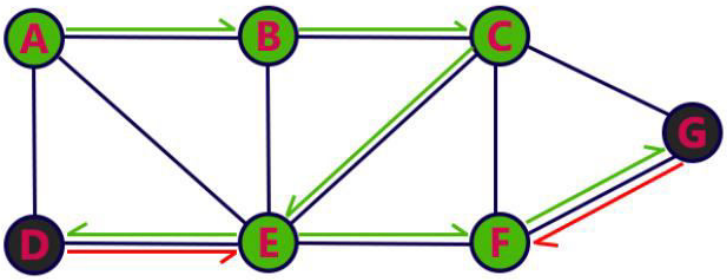
Step 8:

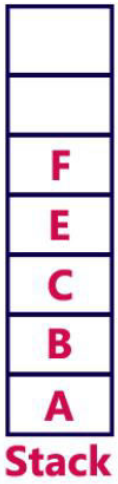
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



Step 9:

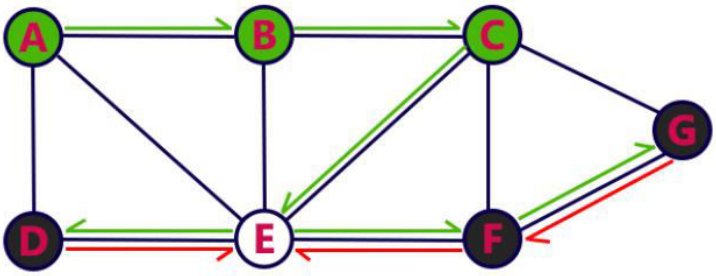
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.





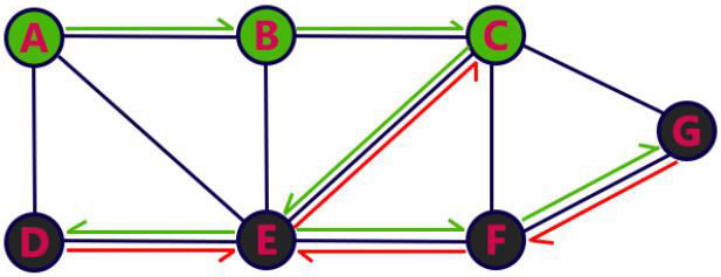
Step 10:

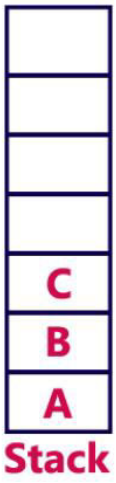
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



Step 11:

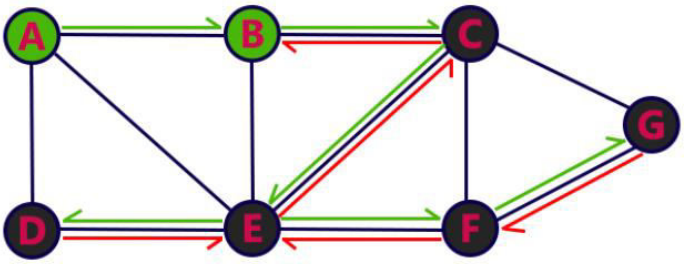
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.





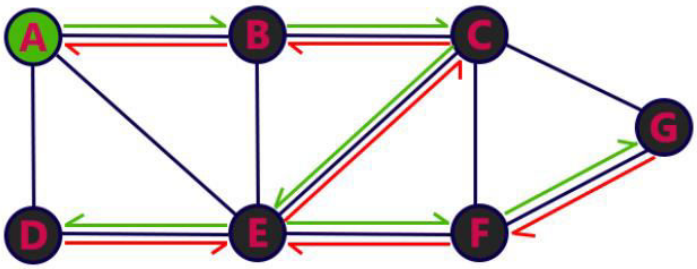
Step 12:

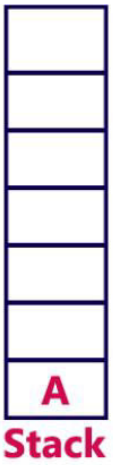
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



Step 13:

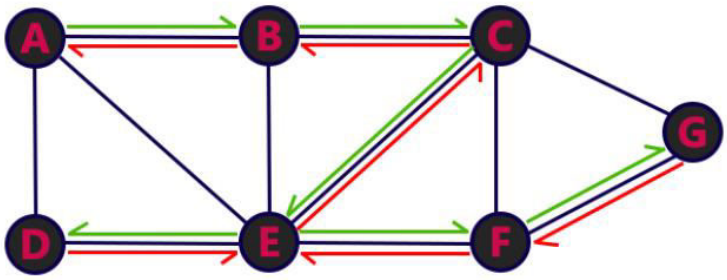
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.



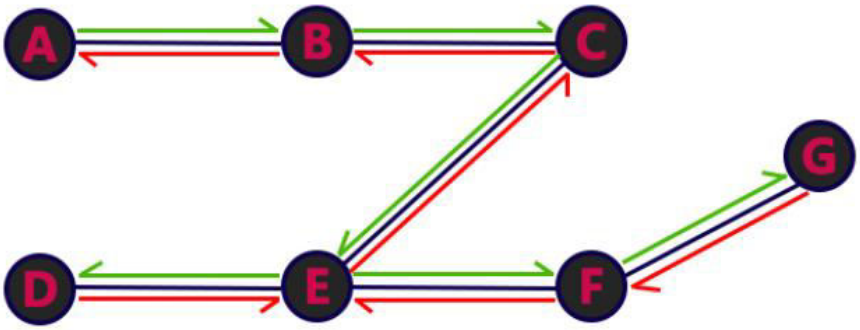


Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



C Program to Implement BFS and DFS

```
#include<stdio.h>

int top=-1,q[20],stack[20],front=-1,rear=-1,arr[20][20],visited[20]={0};


void add(int item)
   {
      if(rear==19)
         printf("QUEUE FULL");
      else
         {
          if(rear==-1)
               {
                  q[++rear] = item;
                  front++;
               }
            else
               q[++rear]=item;
         }
   }

int delete()
{
   int k;

   if ((front>rear)||(front==-1))
      return (0);
   else
```

```
        {
         k=q[front++];
           return(k);
        }
}


void push( int item )
{
   if ( top == 19 )
      printf( "Stack overflow " );
   else
      stack[ ++top ] = item;
}

int pop()
{
   int k;

   if ( top == -1 )
      return ( 0 );
   else
      {
         k = stack[ top-- ];
         return ( k );
      }
}

bfs(int s,int n)
{
int i,p;
add(s);
visited[s]=1;
p=delete();
if(p!=0) printf("%d",p);
while(p!=0)
{
for(i=1;i<=n;i++)
{
if((arr[p][i]!=0)&&(visited[i]==0))
{
add(i);
visited[i]=1;
}
}
p=delete();
if(p!=0) printf("%d",p);
```

```
}
for(i=1;i<=n;i++)
{
if(visited[i]==0) bfs(i,n);
}
}

dfs(int s,int n)
{
int k,i;
push(s);
visited[s]=1;
k=pop();
if(k!=0) printf("%d",k);

while(k!=0)
{
for(i=1;i<=n;i++)
{
if((arr[k][i]!=0)&&(visited[i]==0))
{
push(i);
visited[i]=1;
}
k=pop();
if(k!=0) printf("%d",&k);
}
}
for(i=1;i<=n;i++){
if(visited[i]==0) dfs(i,n);
}
}
int main()
{
int i,j,n,ch,s;
printf("Enter the Number of Verticesn");
scanf("%d",&n);

for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
printf("Enter 1 if %d has a node with %d else 0n",i,j);
scanf("%d",&arr[i][j]);
}
}
printf("Enter Choice 1.BFS 2.DFS n");
```

```
scanf("%d",&ch);
printf("Enter stating vertexn");
scanf("%d",&s);
while(ch!=3)
{
switch(ch)
{
case 1:bfs(s,n);
break;
case 2:dfs(s,n);
break;
}
scanf("%d",&ch);
for(i=0;i<=n;i++){visited[i]=0;}
}
return 0;
}
```

Dijkstra's Algorithm

Dijkstra's algorithm: Finds the minimum-weight path between a pair of vertices in a weighted directed graph.

- Solves the "one vertex, shortest path" problem in weighted graphs.
- Made by famous computer scientist Edsger Dijkstra (look him up!)
- *basic algorithm concept*: Create a table of information about the currently known best way to reach each vertex (cost, previous vertex), and improve it until it reaches the best solution.
- *Example:* In a graph where vertices are cities and weighted edges are roads between cities, Dijkstra's algorithm can be used to find the shortest route from one city to any other.

Dijkstra pseudocode

function **dijkstra**(v_1, v_2):

for each vertex *v*: // Initialize vertex info

v's cost := infinity.

v's previous := none.

v_1's cost := 0.

pqueue := {all vertices, ordered by distance}.

while *pqueue* is not empty:

v := remove vertex from *pqueue* with minimum cost.

mark *v* as visited.

for each unvisited neighbor *n* of *v*:

cost := *v*'s cost + weight of edge (*v*, *n*).

if *cost* < *n*'s cost:

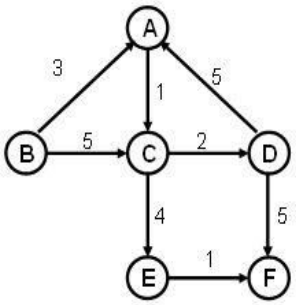
n's cost := *cost*.

n's previous := *v*.

reconstruct path from v_2 back to v_1, following previous pointers.

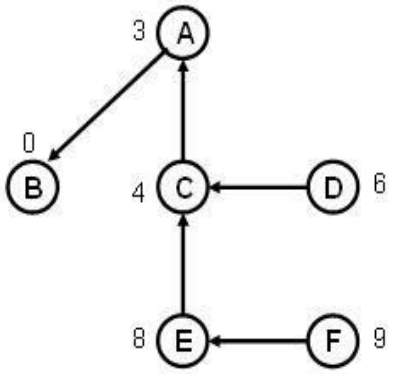
Tracing Dijkstra's algorithm starting at vertex B:

Tracing Dijkstra's algorithm starting at vertex B:



Pass: / Active vertex:	initially	1 B	2 A	3 C	4 D	5 E	6 F	Shortest distance	Predecessor
A	∞	3						3	B
B	0							0	-
C	∞	5	4					4	A
D	∞	∞	∞	6				6	C
E	∞	∞	∞	8	8			8	C
F	∞	∞	∞	∞	11	9		9	E

The resulting vertex-weighted graph is:



Dijkstra example:

dijkstra(A, F);

function **dijkstra**(v_1, v_2):

for each vertex *v*: // Initialize vertex info

v's cost := infinity.

v's previous := none.

v_1's cost := 0.

pqueue := {all vertices,

by distance}.

while *pqueue* is not empty:

v := *pqueue*.removeMin().

mark *v* as visited.

for each unvisited neighbor *n* of *v*:

cost := *v*'s cost + edge(*v*, *n*)'s weight.

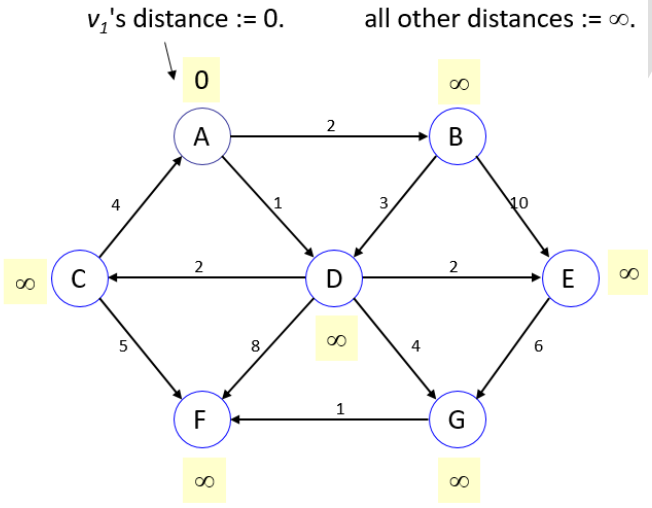
if *cost* < *n*'s cost:

n's cost := *cost*.

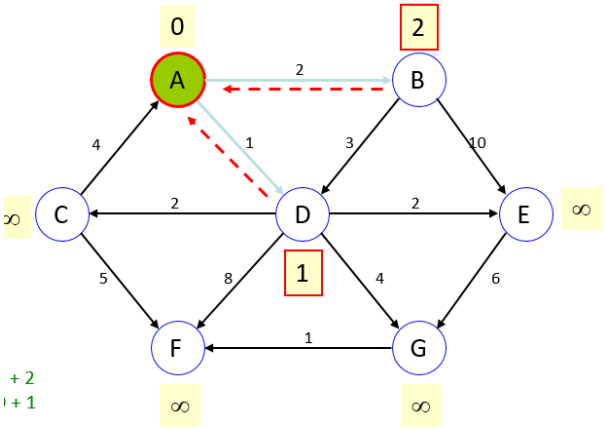
n's previous := *v*.

reconstruct path from v_2 back to v_1,

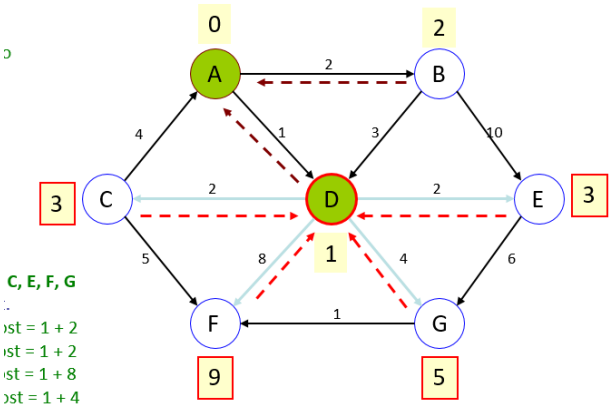
following previous pointers.



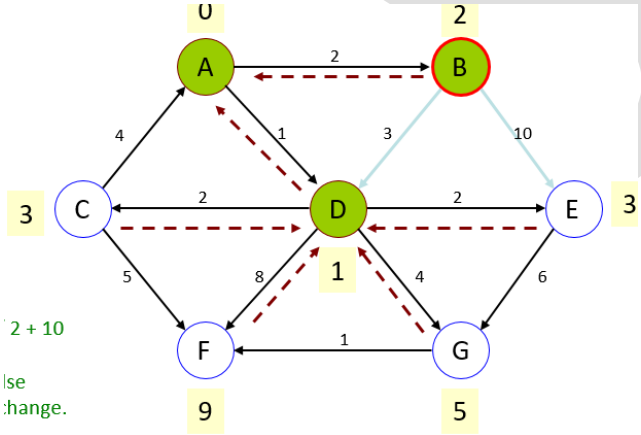
pqueue = [**A:0**, B:∞, C:∞, D:∞, E:∞, F:∞, G:∞]



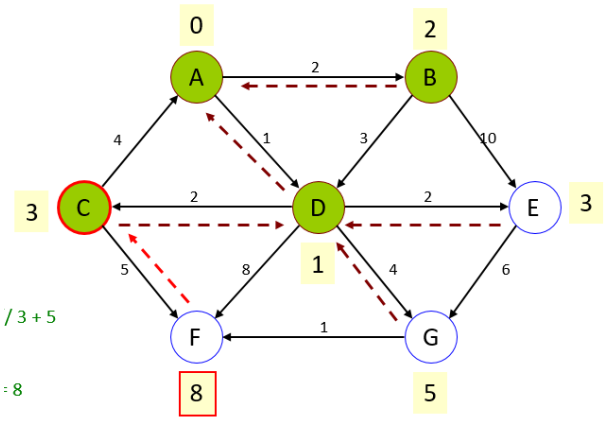
pqueue = [**D:1, B:2**, C:∞, E:∞, F:∞, G:∞]



pqueue = [B:2, **C:3, E:3, G:5, F:9**]



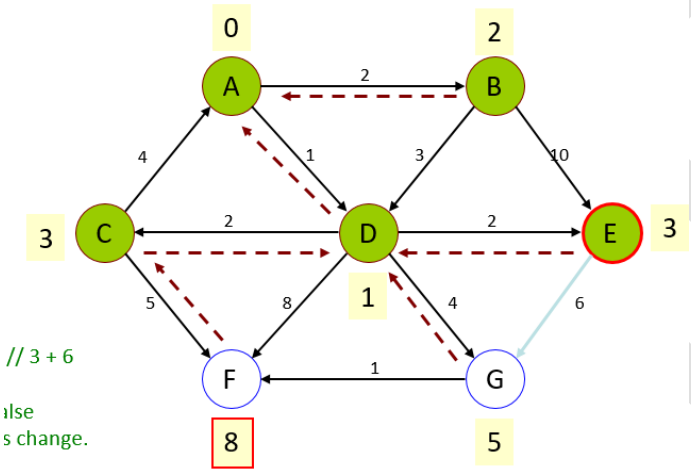
pqueue = [C:3, E:3, G:5, F:9]



/ 3 + 5

: 8

pqueue = [E:3, G:5, **F:8**]

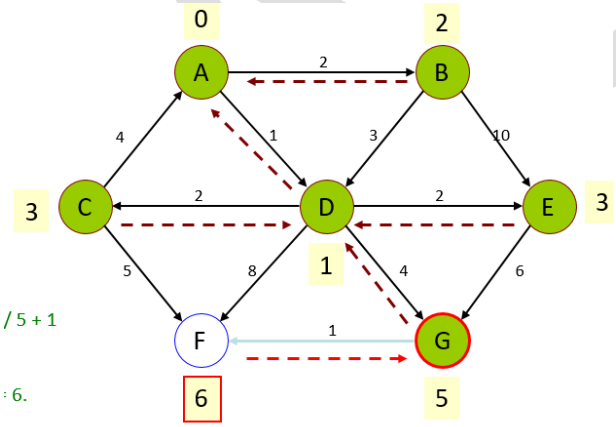


// 3 + 6

ılse

s change.

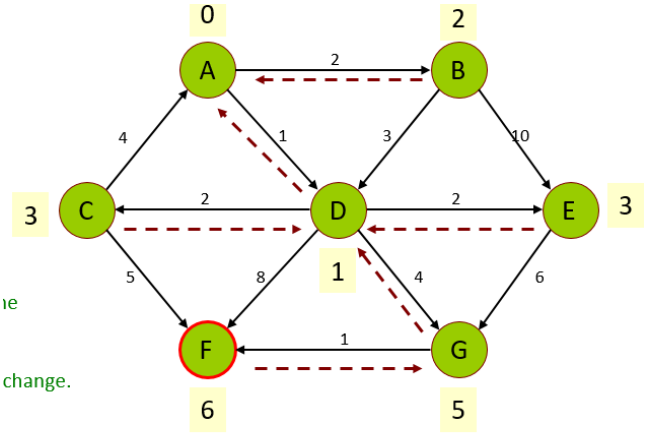
pqueue = [G:5, F:8]



/ 5 + 1

: 6.

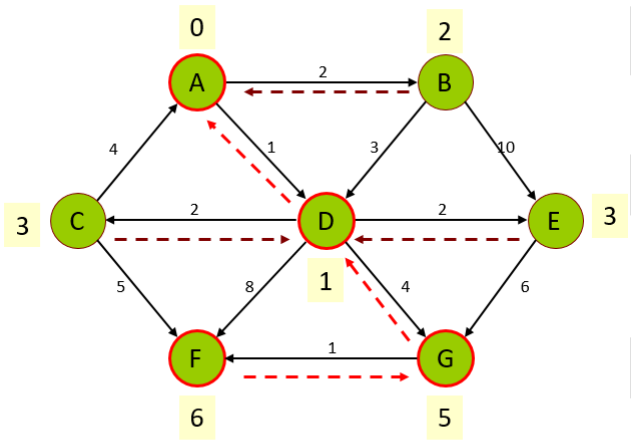
pqueue = [**F:6**]



ıe

change.

pqueue = [1



// path = [A, D, G, F]