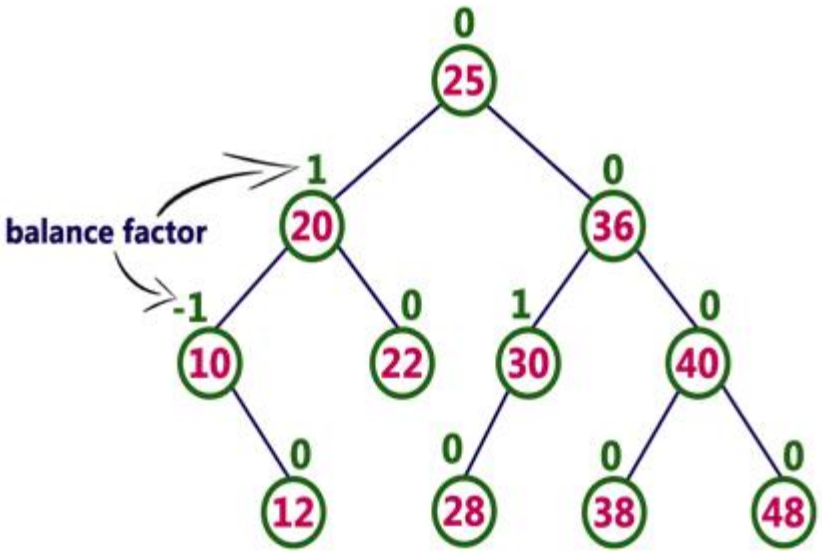
UNIT -1

AVL TREE

AVL tree is a height-balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced if, the difference between the heights of left and right sub trees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. In an AVL tree, every node maintains an extra information known as **balance factor**.

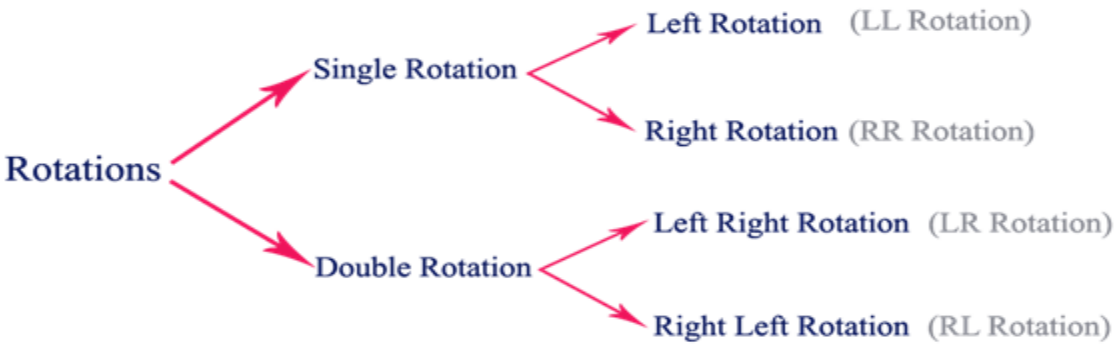
Balance factor of a node is the difference between the heights of the left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**. In the following explanation, we calculate as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree



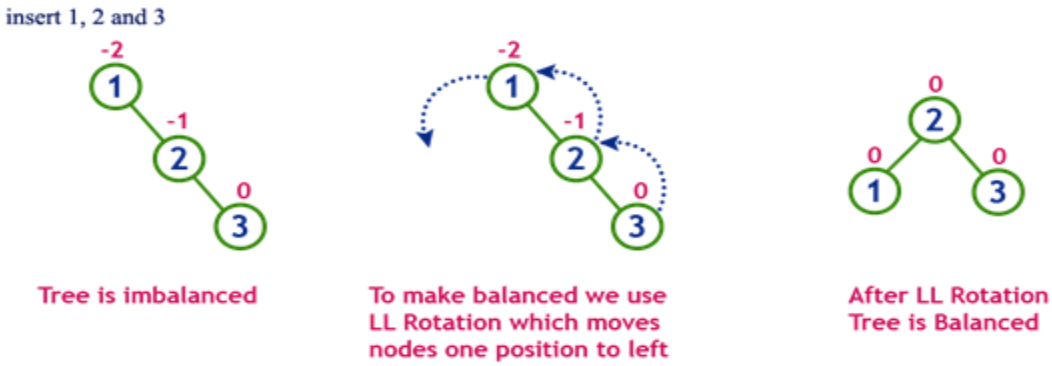
AVL Tree Rotation

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



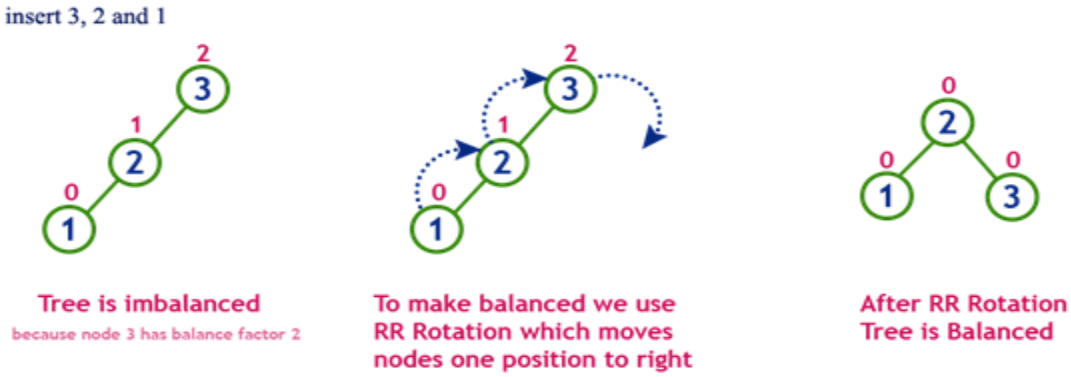
a. Single Left Rotation (LL Rotation)

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation



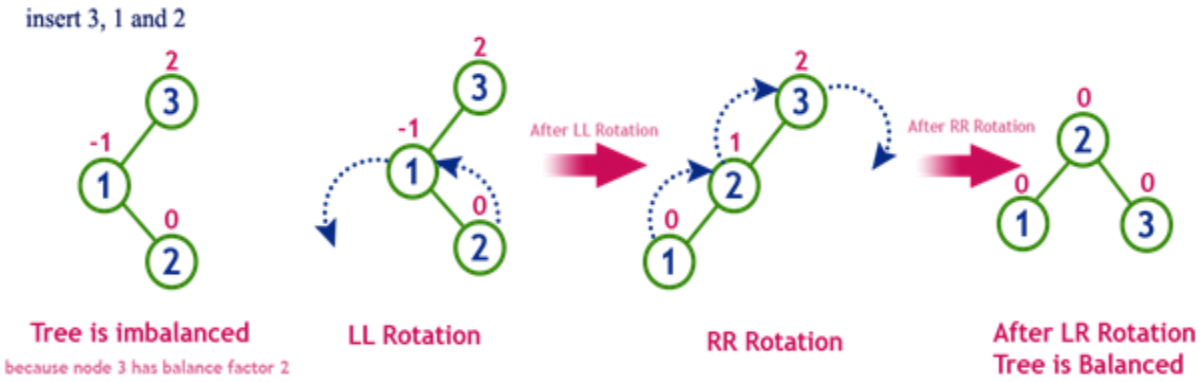
b. Single Right Rotation (RR Rotation)

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...



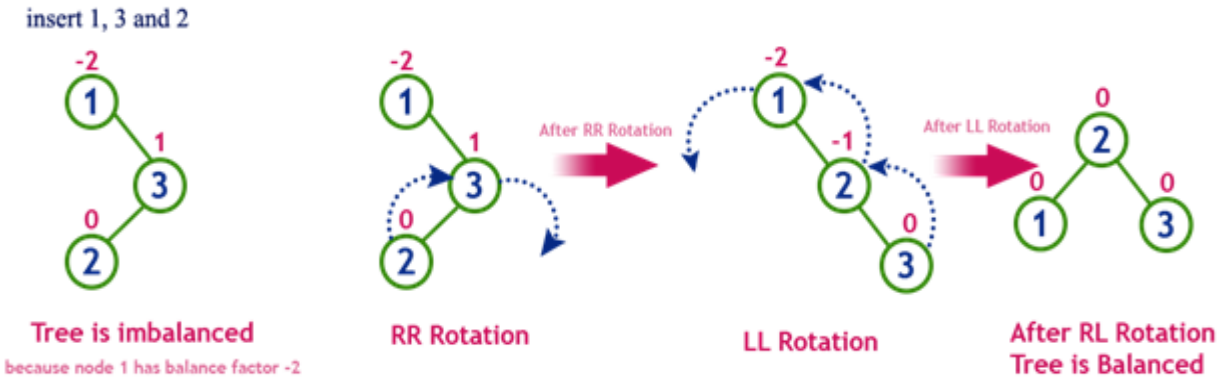
c. Left Right Rotation (LR Rotation)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



d. Right Left Rotation (RL Rotation)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...

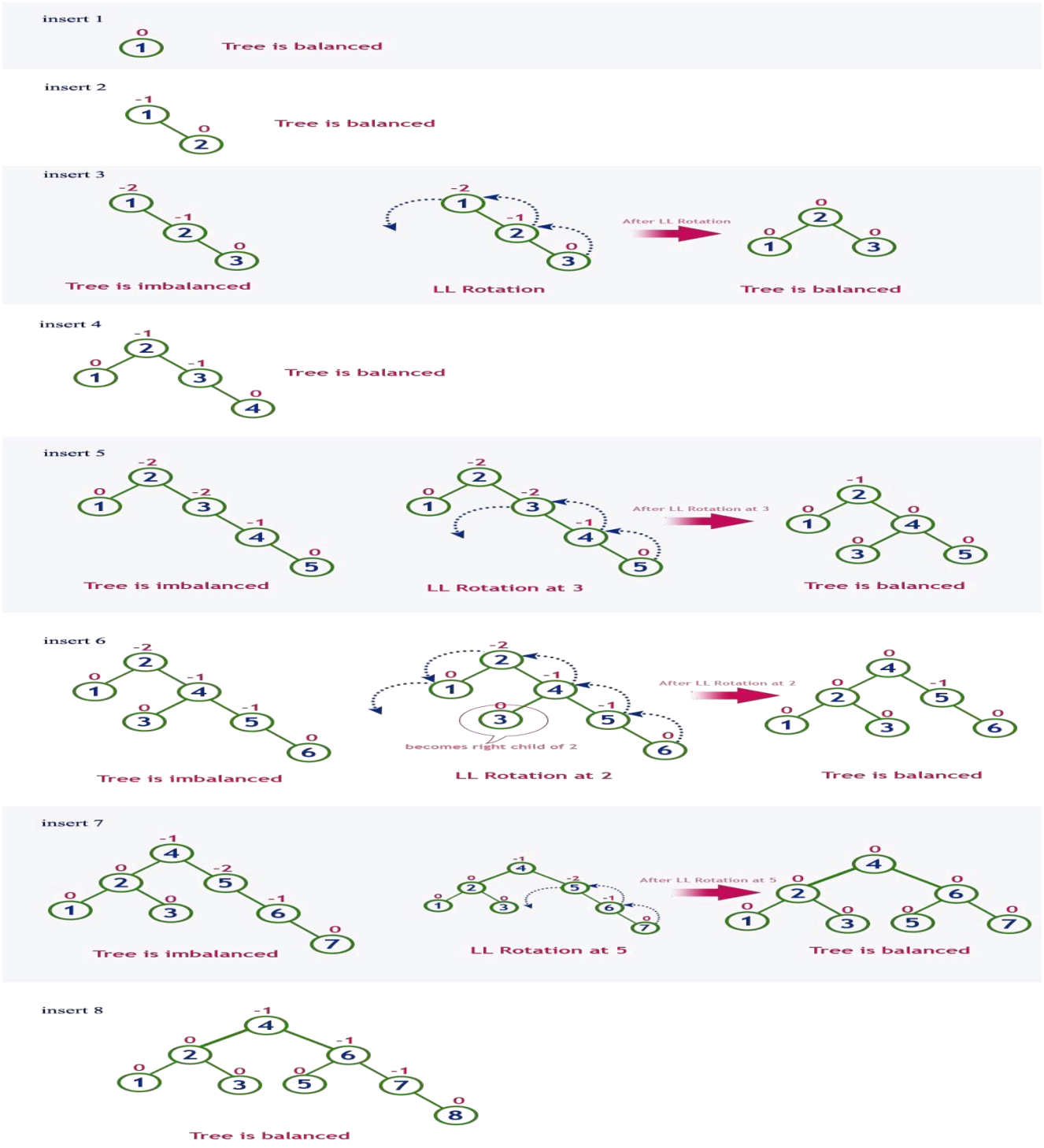


Inserting a New Node in an AVL Tree

Insertion in an AVL tree is also done in the same way as it is done in a binary search tree. In the AVL tree, the new node is always inserted as the leaf node. But the step of insertion is usuallyfollowed by an additional step of rotation. Rotation is done to restore the balance of the tree.However, if

insertion of the new node does not disturb the balance factor, that is, if the balance factor of every node is still –1, 0, or 1, then rotations are not required.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.



Insertion Algorithm:

Step 1: Insert the node into the AVL tree like standard Binary Search Tree (BST) insertion.

Step 2: After inserting node if the tree is balanced no need to perform rotations. Otherwise perform steps 3 to 4:

Step 3: If the balance factor of the current node is > 1 or < -1, perform the appropriate rotations to balance the tree.

Step 3.1: Left Rotation Case: Perform Right rotation

Step 3.2: Right Rotation Case: Perform Left rotation

Step 3.3: Left-Right (LR) Rotation Case: Perform a left rotation on the left child, then a right rotation on the current node.

Step 3.4: Right-Left (RL) Rotation Case: Perform a right rotation on the right child, then a left rotation on the current node.

Step 4: After balancing, return the root of the subtree.

Step 5: End

Deletion

A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. To rebalance the balance factor, suitable rotations are performed.

There are three cases for deleting a node:

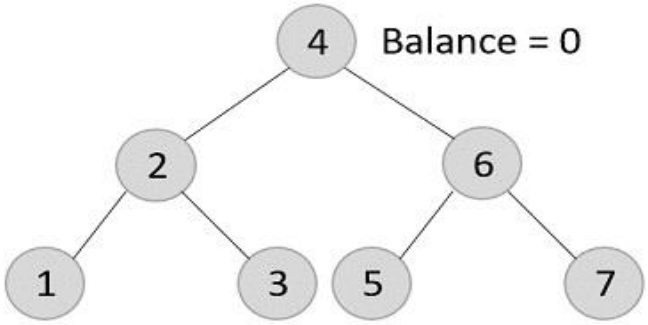
1: If the node to be deleted has two children, replace it with its in-order successor (or predecessor) and delete the successor (or predecessor) node.

2: If the node to be deleted has one child, replace it with its child.

3: If the node to be deleted has no children, simply remove it.

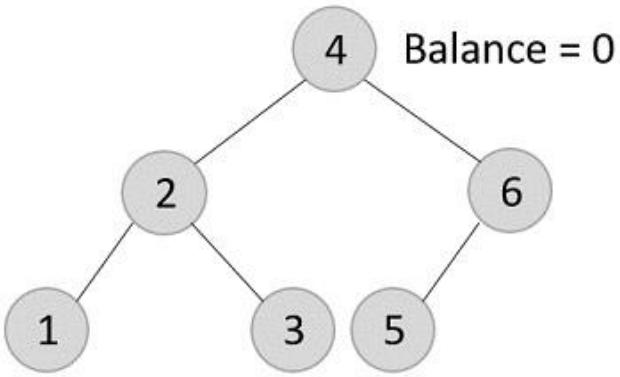
Example:-

Delete 7, 6, 5,2 from the following AVL Tree



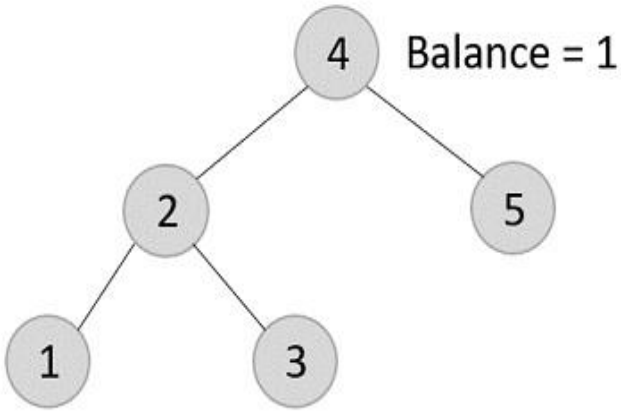
- **Deleting element 7 from the tree above −**

Since the element 7 is a leaf, we normally remove the element without disturbing any other node in the tree



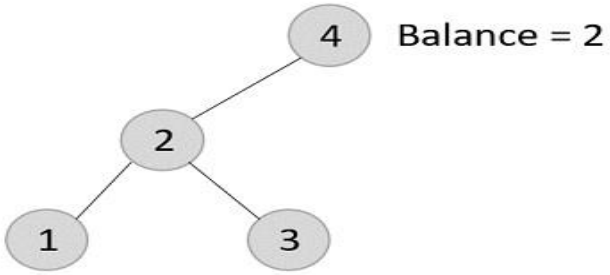
- Deleting element 6 from the output tree achieved −

However, element 6 is not a leaf node and has one child node attached to it. In this case, we replace node 6 with its child node: node 5.

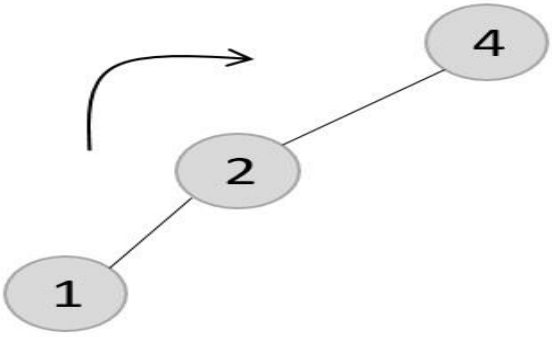


- **Deleting element 5 from the output tree achieved −**

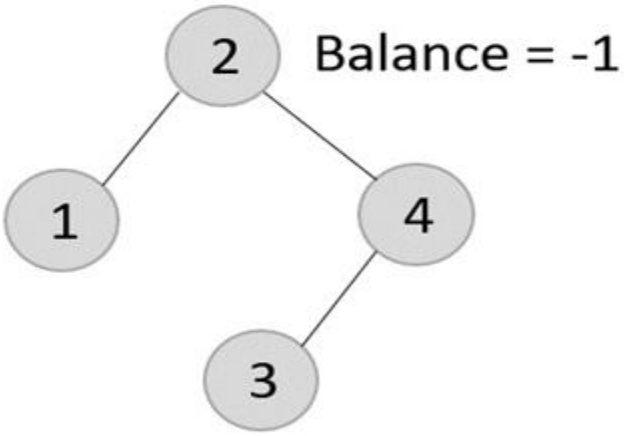
If we delete the element 5 further, we would have to apply the left rotations



The balance factor is disturbed after deleting the element 5, therefore we apply LL rotation (we can also apply the LR rotation here).

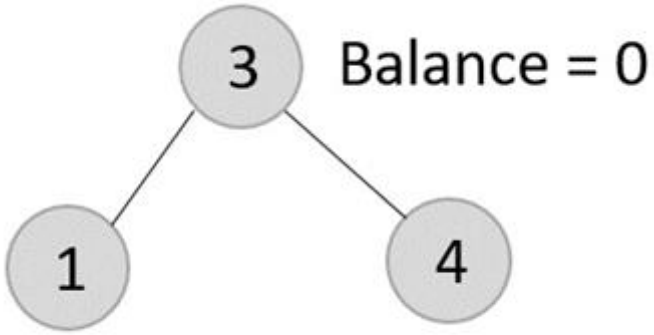


Once the LL rotation is applied on path 1-2-4, the node 3 remains as it was supposed to be the right child of node 2 (which is now occupied by node 4). Hence, the node is added to the right subtree of the node 2 and as the left child of the node 4.



- **Deleting element 2 from the remaining tree −**

As mentioned in scenario 3, this node has two children. Therefore, we find its inorder successor that is a leaf node (say, 3) and replace its value with the inorder successor.

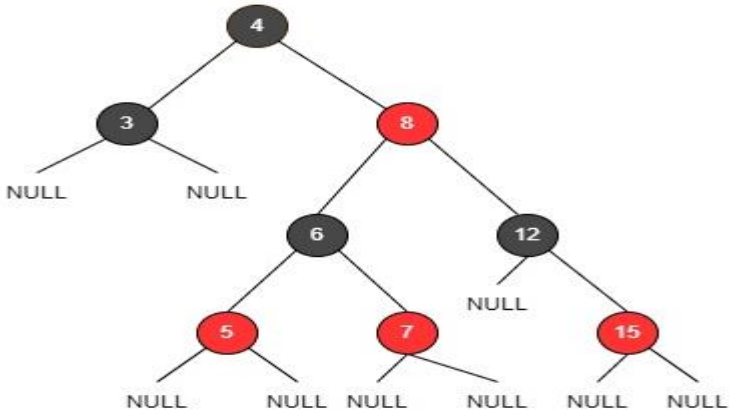


RED-BLACK TREES

Red-Black Trees are another type of the Balanced Binary Search Trees with two coloured nodes: Red and Black. It is a self-balancing binary search tree that makes use of these colours to maintain the balance factor during the insertion and deletion operations.

In Red-Black trees, also known as RB trees, there are different conditions to follow while assigning the colours to the nodes.

- The root node is always black in colour.
- No two adjacent nodes must be red in colour.
- Every path in the tree (from the root node to the leaf node) must have the same amount of black coloured nodes.



Basic Operations of Red-Black Trees

The operations on Red-Black Trees include all the basic operations usually performed on a Binary Search Tree. Some of the basic operations of an RB Tree include −

- Insertion
- Deletion
- Search

Insertion operation

Insertion operation of a Red-Black tree follows the same insertion algorithm of a binary search tree. The elements are inserted following the binary search property and as an addition; the nodes are color coded as red and black to balance the tree according to the red-black tree properties.

Follow the procedure given below to insert an element into a red-black tree by maintaining both binary search tree and red black tree properties.

Case 1 − Check whether the tree is empty; make the current node as the root and color the node black if it is empty.

Case 2 − But if the tree is not empty, we create a new node and color it red. Here we face two different cases −

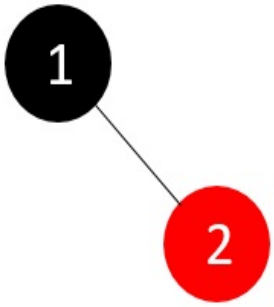
- If the parent of the new node is a black colored node, we exit the operation and tree is left as it is.
- If the parent of this new node is red and the color of the parent's sibling is either black or if it does not exist, we apply a suitable rotation and recolor accordingly.
- If the parent of this new node is red and color of the parent's sibling is red, recolor the parent, the sibling and grandparent nodes to black. The grandparent is recolored only if it is **not** the root node; if it is the root node recolor only the parent and the sibling.

Example

Let us construct an RB Tree for the first 7 integer numbers to understand the insertion operation in detail −

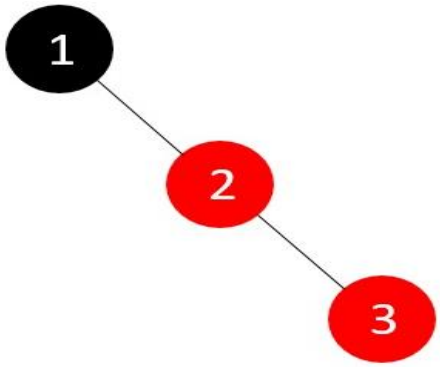
The tree is checked to be empty so the first node added is a root and is colored black.

Now, the tree is not empty so we create a new node and add the next integer with color red,

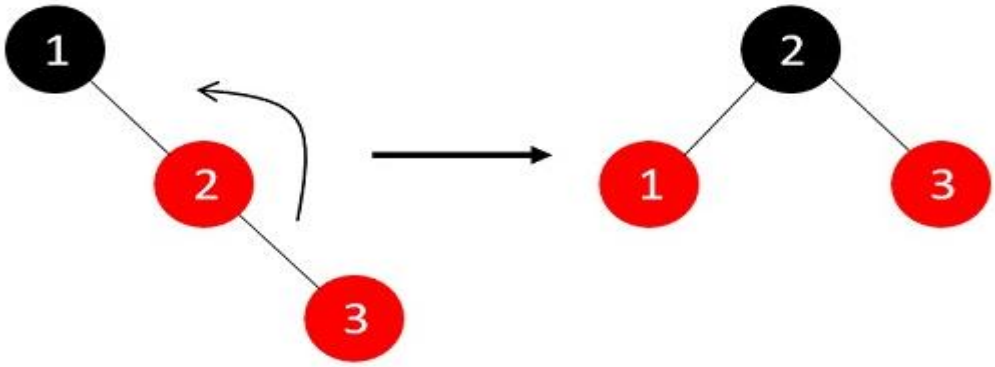


The nodes do not violate the binary search tree and RB tree properties, hence we move ahead to add another node.

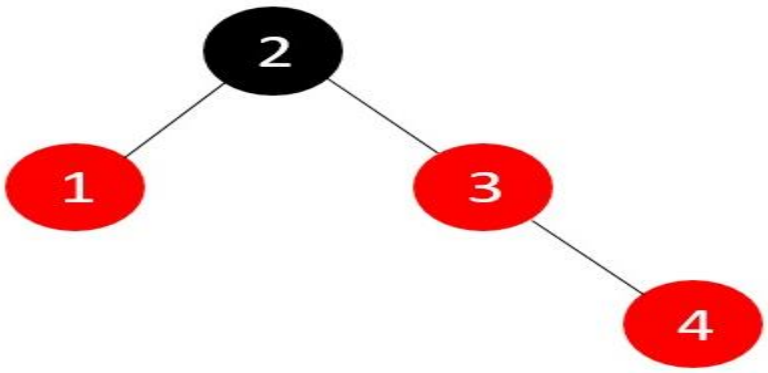
The tree is not empty; we create a new red node with the next integer to it. But the parent of the new node is not a black colored node,



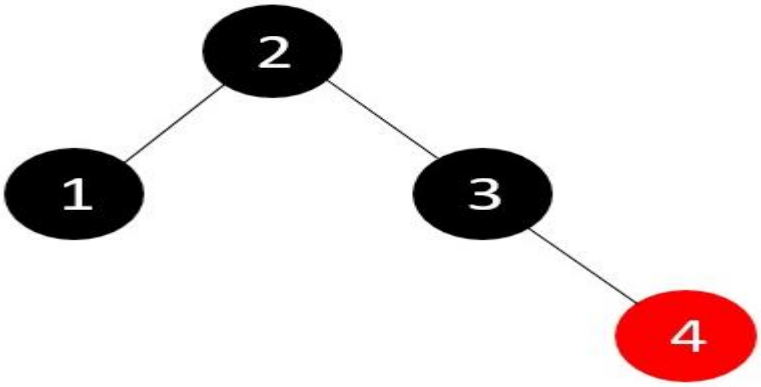
The tree right now violates both the binary search tree and RB tree properties; since parent's sibling is NULL, we apply a suitable rotation and recolor the nodes.



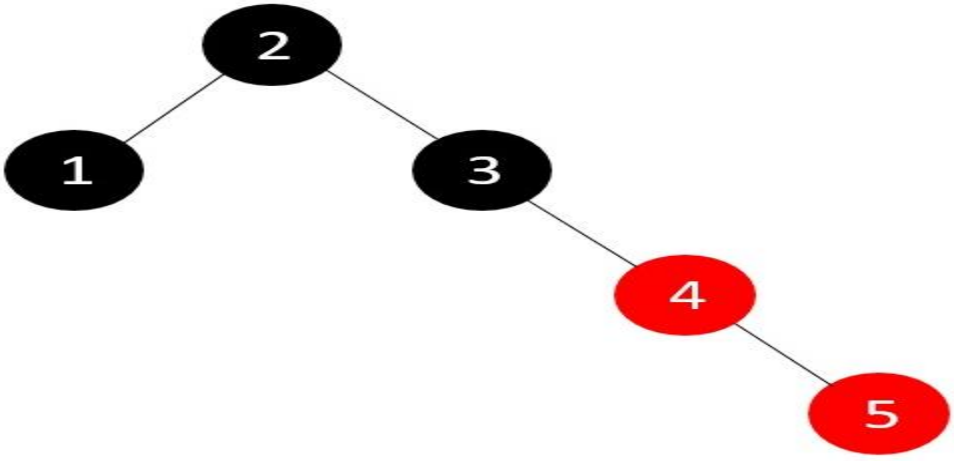
Now that the RB Tree property is restored, we add another node to the tree −



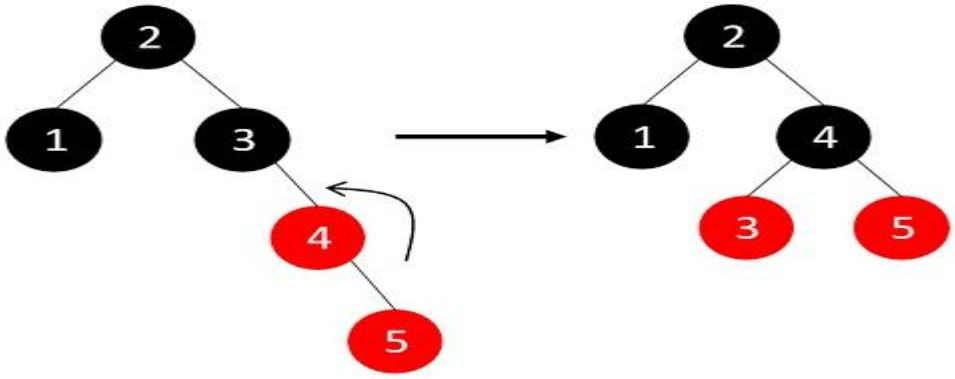
The tree once again violates the RB Tree balance property, so we check for the parent's sibling node color, red in this case, so we just recolor the parent and the sibling.



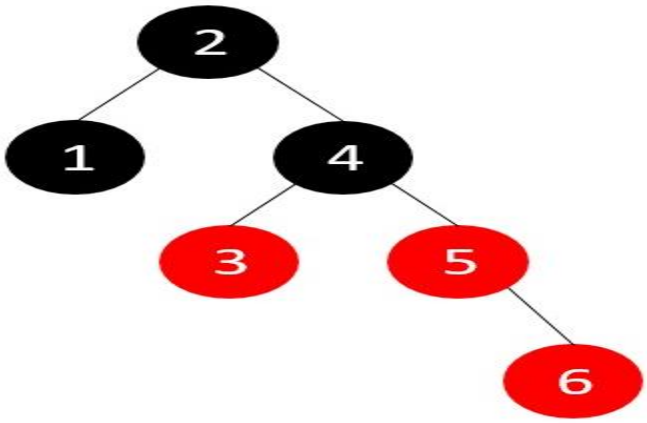
We next insert the element 5, which makes the tree violate the RB Tree balance property once again.



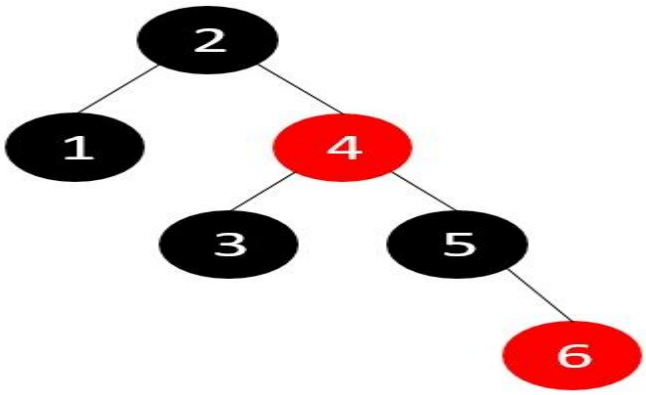
And since the sibling is NULL, we apply suitable rotation and recolor.



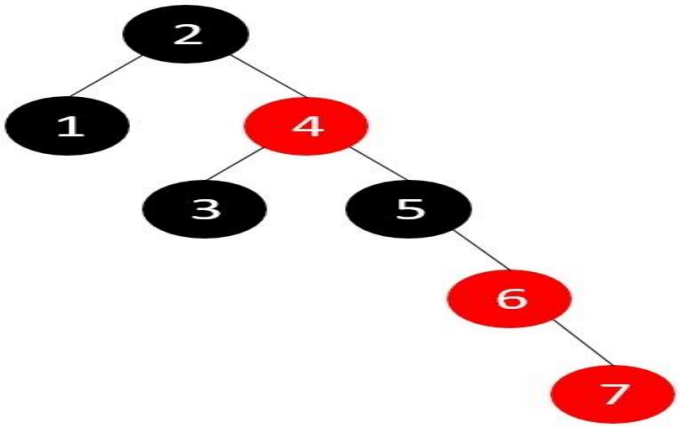
Now, we insert element 6, but the RB Tree property is violated and one of the insertion cases need to be applied −



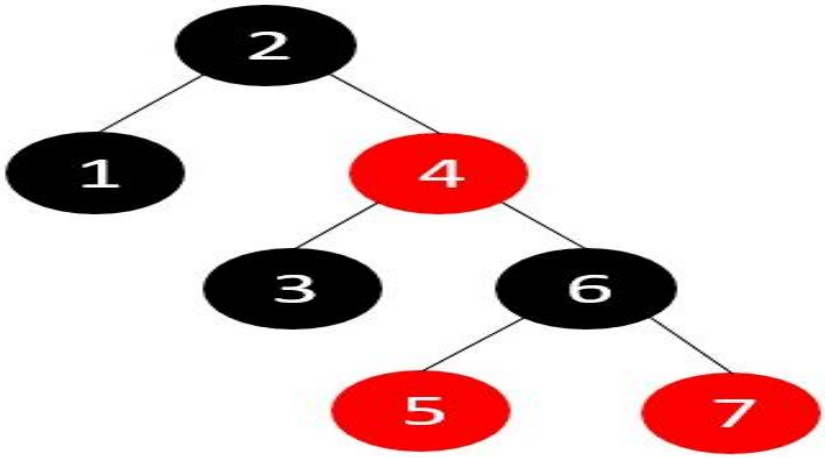
The parent's sibling is red, so we recolor the parent, parent's sibling and the grandparent nodes since the grandparent is not the root node.



Now, we add the last element, 7, but the parent node of this new node is red.



Since the parent's sibling is NULL, we apply suitable rotations (RR rotation)



Deletion operation

The deletion operation on red black tree must be performed in such a way that it must restore all the properties of a binary search tree and a red black tree. Follow the steps below to perform the deletion operation on the red black tree −

Firstly, we perform deletion based on the binary search tree properties.

Case 1 − If either the node to be deleted or the node's parent is red, just delete it.

Case 2 − If the node is a double black, just remove the double black (double black occurs when the node to be deleted is a black colored leaf node, as it adds up the NULL nodes which are considered black colored nodes too)

Case 3 − If the double black's sibling node is also a black node and its child nodes are also black in color, follow the steps below −

- Remove double black
- Recolor its parent to black (if the parent is a red node, it becomes black; if the parent is already a black node, it becomes double black)
- Recolor the parent's sibling with red
- If double black node still exists, we apply other cases.

Case 4 − If the double black node's sibling is red, we perform the following steps −

- Swap the colors of the parent node and the parent's sibling node.
- Rotate parent node in the double black's direction
- Reapply other cases that are suitable.

Case 5 − If the double black's sibling is a black node but the sibling's child node that is closest to the double black is red, follows the steps below −

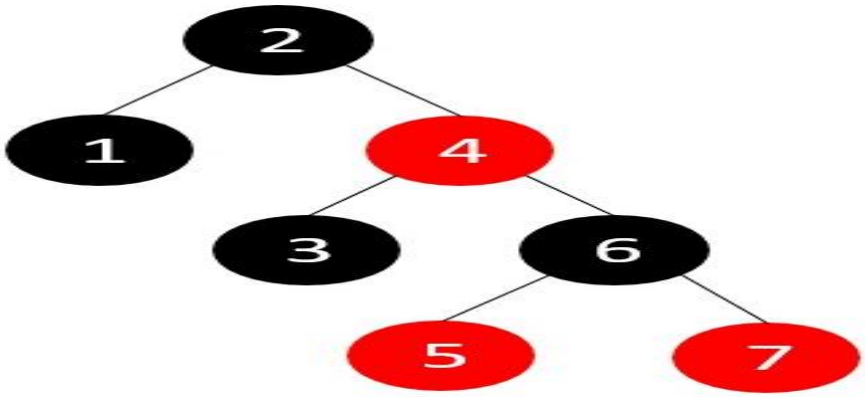
- Swap the colors of double black's sibling and the sibling's child in question
- Rotate the sibling node is the opposite direction of double black (i.e. if the double black is a right child apply left rotations and vice versa)
- Apply case 6.

Case 6 − If the double black's sibling is a black node but the sibling's child node that is farther to the double black is red, follows the steps below −

- Swap the colors of double black's parent and sibling nodes
- Rotate the parent in double black's direction (i.e. if the double black is a right child apply right rotations and vice versa)
- Remove double black
- Change the color of red child node to black.

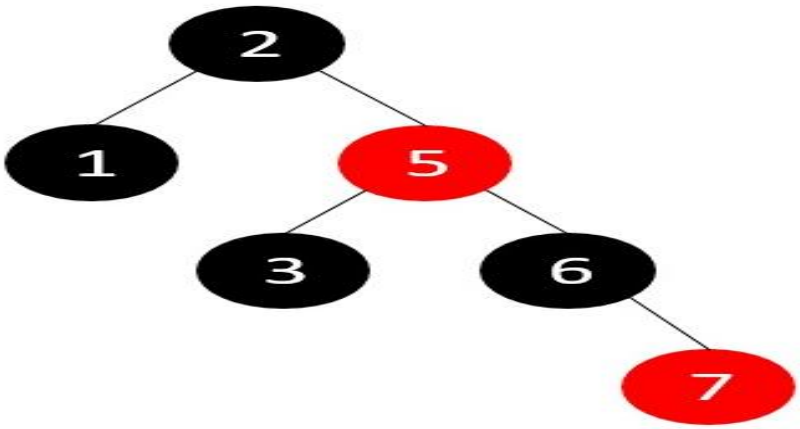
Example

Considering the same constructed Red-Black Tree above, let us delete few elements from the tree.



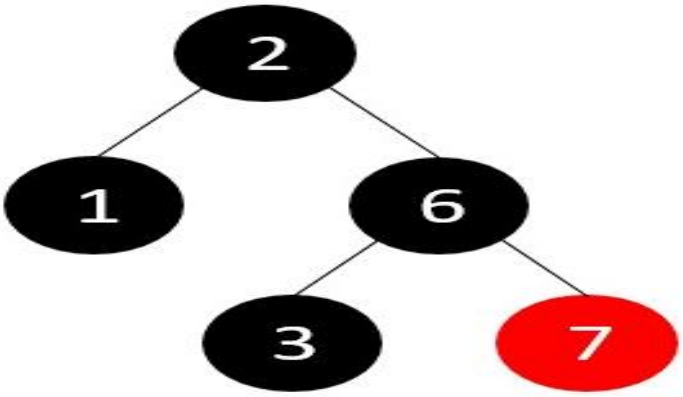
Delete elements 4, 5, 3 from the tree.

To delete the element 4, let us perform the binary search deletion first.

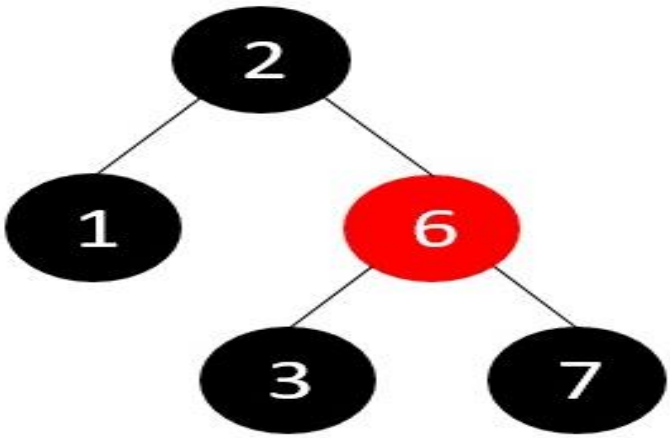


After performing the binary search deletion, the RB Tree property is not disturbed, therefore the tree is left as it is.

Then, we delete the element 5 using the binary search deletion

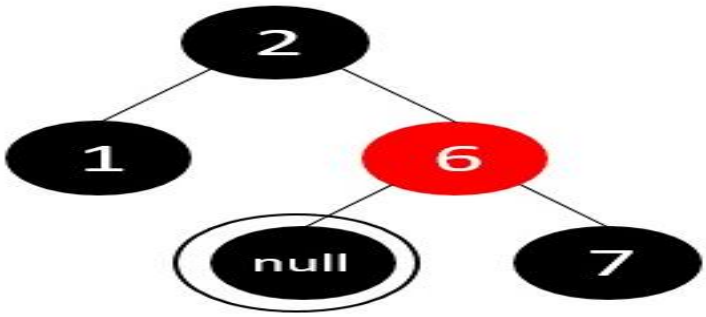


But the RB property is violated after performing the binary search deletion, i.e., all the paths in the tree do not hold same number of black nodes; so we swap the colors to balance the tree.

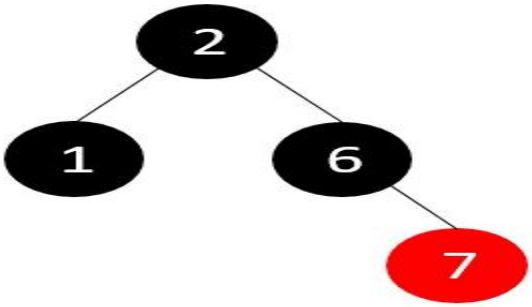


Then, we delete the node 3 from the tree obtained −

Applying binary search deletion, we delete node 3 normally as it is a leaf node. And we get a double node as 3 is a black colored node.



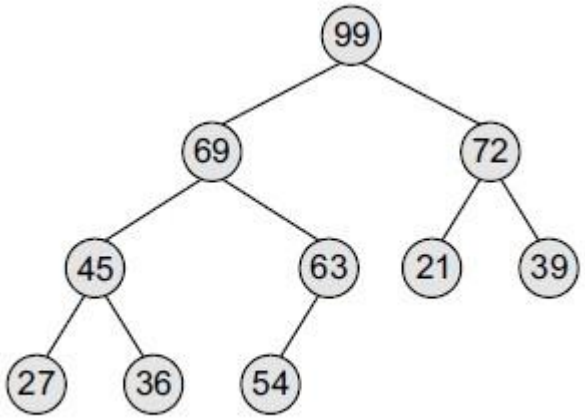
We apply case 3 deletion as double black's sibling node is black and its child nodes are also black. Here, we remove the double black, recolor the double black's parent and sibling.



BINARY HEAPS

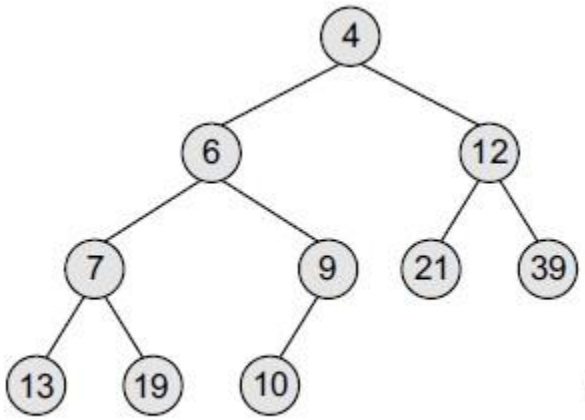
A binary heap is a complete binary tree in which every node satisfies the heap property which states that:
If B is a child of A, then key(A) ≥ key(B)

This implies that elements at every node will be either greater than or equal to the element at its left and right child. Thus, the root node has the highest key value in the heap. Such a heap is commonly known as a *max-heap*.



Max heap

Alternatively, elements at every node will be either less than or equal to the element at its left and right child. Thus, the root has the lowest key value. Such a heap is called a *min-heap*



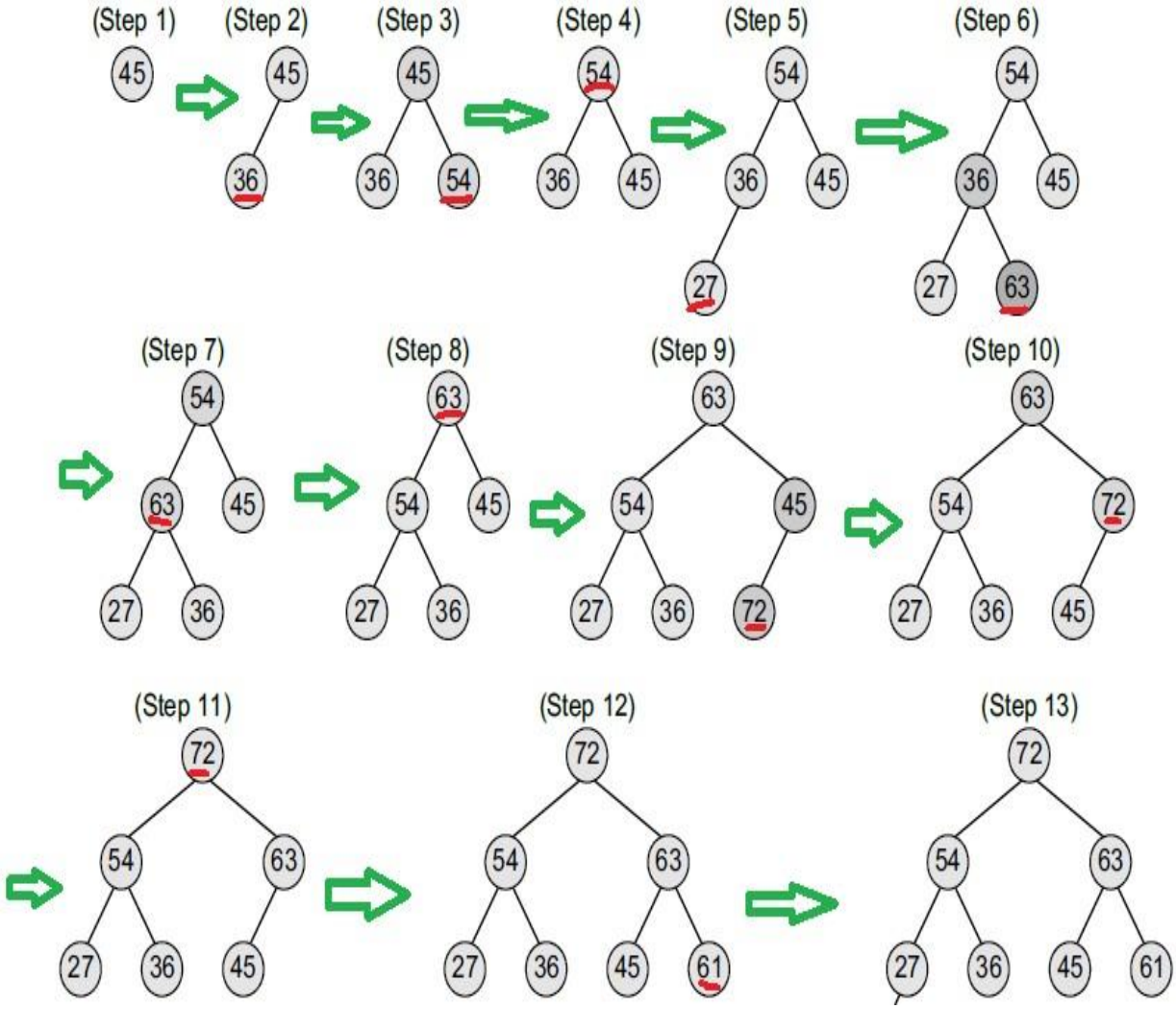
Min heap

The properties of binary heaps are given as follows:

- Since a heap is defined as a complete binary tree, all its elements can be stored sequentially in an array. It follows the same rules as that of a complete binary tree. That is, if an element is at position i in the array, and then its left child is stored at position 2i and its right child at position 2i+1. Conversely, an element at position i have its parent stored at position i/2.
- Being a complete binary tree, all the levels of the tree except the last level are completely filled.
- The height of a binary tree is given as log2n, where n is the number of elements.
- Heaps (also known as partially ordered trees) are a very popular data structure for implementing priority queues.

Example:-

Build a max heap H from the given set of numbers: 45, 36, 54, 27, 63, 72, 61, and 18. Also draw the memory representation of the heap.

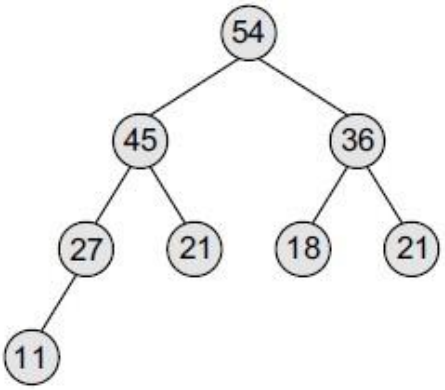


Inserting a New Element in a Binary Heap

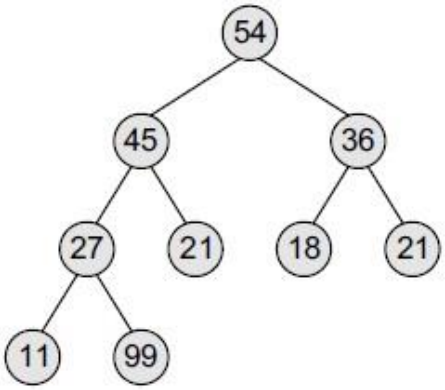
Consider a max heap H with n elements. Inserting a new value into the heap is done in the following two steps:

1. Add the new value at the bottom of H in such a way that H is still a complete binary tree but not necessarily a heap.
2. Let the new value rise to its appropriate place in H so that H now becomes a heap as well. To do this, compare the new value with its parent to check if they are in the correct order. If they are, then the procedure halts, else the new value and its parent's value are swapped and Step 2 is repeated.

The first step says that insert the element in the heap so that the heap is a complete binary tree. So, insert the new value as the right child of node 27 in the heap. This is illustrated in Fig.

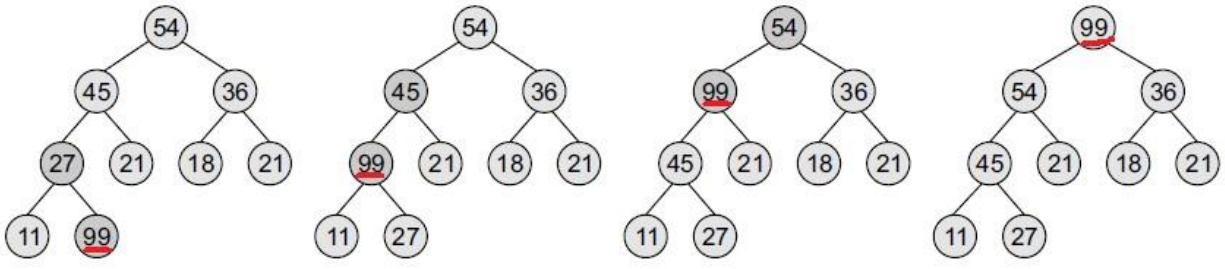


Binary heap



Binary heap after insertion of 99

Now, as per the second step, let the new value rise to its appropriate place in H so that H becomes a heap as well. Compare 99 with its parent node value. If it is less than its parent's value, then the new node is in its appropriate place and H is a heap. If the new value is greater than that of its parent's node, then swap the two values. Repeat the whole process until H becomes a heap. This is illustrated in Fig.

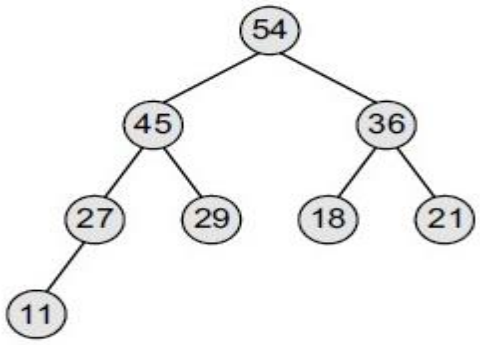


Heapify the binary heap

Deleting an Element from a Binary Heap

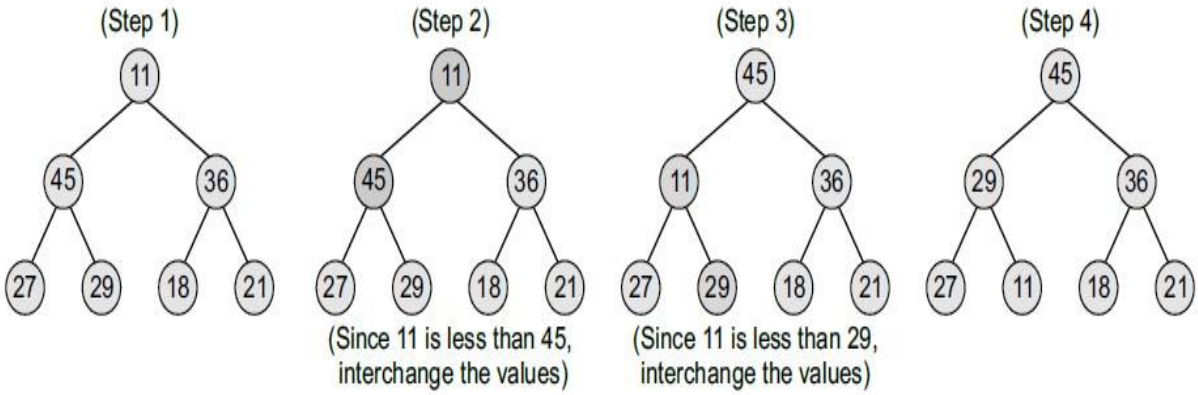
Consider a max heap H having n elements. An element is always deleted from the root of the heap. So, deleting an element from the heap is done in the following three steps:
2. Replace the root node's value with the last node's value so that H is still a complete binary tree but not necessarily a heap.
3. Delete the last node.
4. Sink down the new root node's value so that H satisfies the heap property. In this step, interchange the root node's value with its child node's value (whichever is largest among its children).

Example:-
Consider the binary heap tree



Binary heap

If we want to delete the root value i.e the value 54 in the heap tree. Here, the value of root node =54 and the value of the last node = 11. So, replace 54 with 11 and delete the last node.



GRAPHS

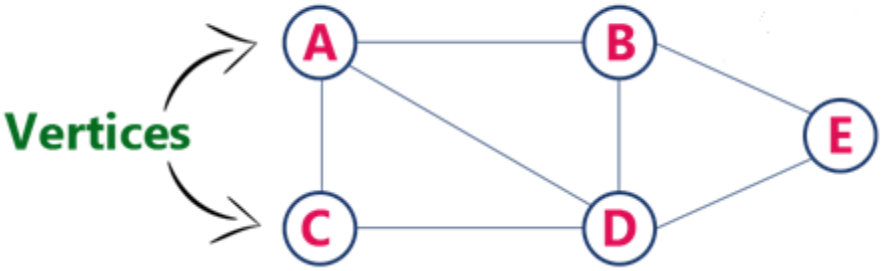
Definition

A graph G is defined as an ordered set (V, E), where V(G) represents the set of vertices and E(G) represents the edges that connect these vertices.

Graph Terminology

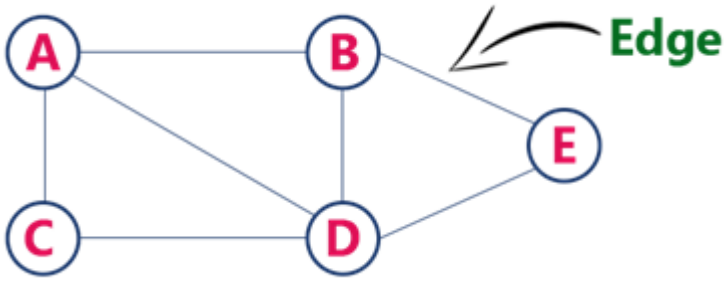
Vertex

Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In the given example graph, A, B, C, D & E are known as vertices.



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 the graph the link between vertices A and B is represented as (A,B). In the example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).



Adjacency − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.

Path − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

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.

Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

TYPES OF GRAPES

Graphs are two types

i. Directed graph

ii. Un directed graph

i. Directed Graphs

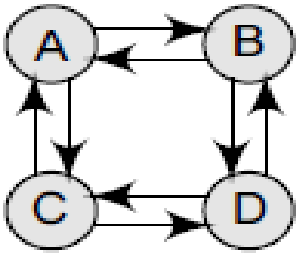
A directed graph G, also known as a *digraph*, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge (u, v),

a. The edge begins at u and terminates at v.

b. u is known as the origin or initial point of e. Correspondingly, v is known as the destination or terminal point of e.

c. u is the predecessor of v. Correspondingly, v is the successor of u.

d. Nodes u and v are adjacent to each other.



ii. Un directed Graph

An undirected graph is a graph in which all the edges are bi-directional i.e. the edges do not point in any specific direction.

REPRESENTATION OF GRAPHS

There are three common ways of storing graphs in the computer's memory. They are:

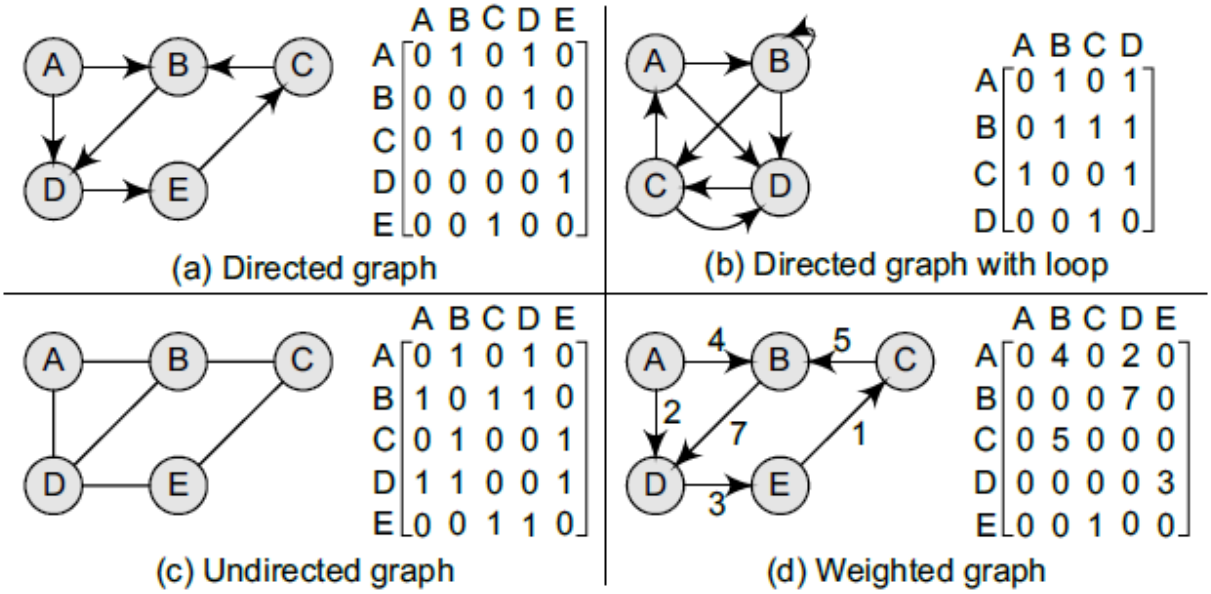
i. Sequential representation by using an adjacency matrix.

ii. Linked representation by using an adjacency list that stores the neighbors of a node using a linked list.

Adjacency Matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them. In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of nX n.

In an adjacency matrix, the rows and columns are labeled by graph vertices. An entry aij in the adjacency matrix will contain 1, if vertices vi and vj are adjacent to each other. However, if the nodes are not adjacent, a ij will be set to zero. Since an adjacency matrix contains only 0s and 1s, it is called a *bit matrix* or a *Boolean matrix*. The entries in the matrix depend on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix. Figure shows some graphs and their corresponding adjacency matrices.



Adjacency List Representation

An adjacency list is another way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

The key advantages of using an adjacency list are:

a. It is easy to follow and clearly shows the adjacent nodes of a particular node.

b. It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.

c. Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

Consider the graph given in Fig., and see how its adjacency list is stored in the memory. For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of edges in G. However, for an undirected graph, the sum of the lengths of all adjacency lists is equal to twice the number of edges in G because an edge (u, v) means an edge from node u to v as well as an edge from v to u. Adjacency lists can also be modified to store weighted graphs. Let us now see an adjacency list for an undirected graph as well as a weighted graph.

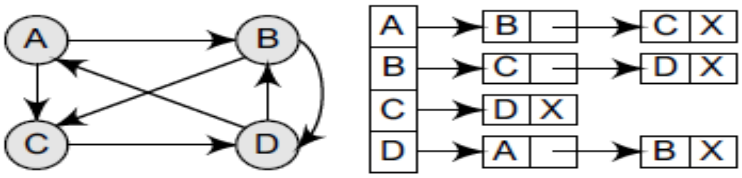
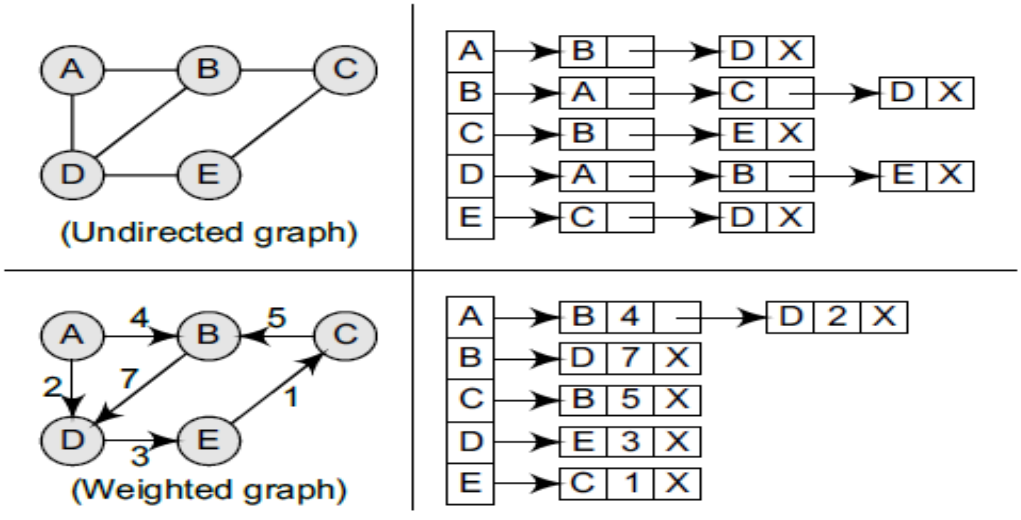


Figure 13.17 Graph G and its adjacency list



GRAPH TRAVERSAL ALGORITHMS

By traversing a graph, we mean the method of examining the nodes and edges of the graph. There are two standard methods of graph traversal. These two methods are:

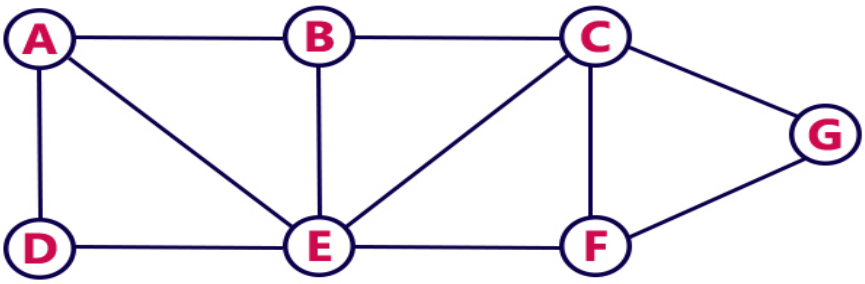
1. Breadth-first search
2. Depth-first search

Breadth-First Search Algorithm

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, explores their unexplored neighbor nodes, and so on, until it finds the goal. That is, we start examining the node A and then all the neighbors of A are examined. In the next step, we examine the neighbors of neighbors of A, so on and so forth. This means that we need to track the neighbors of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

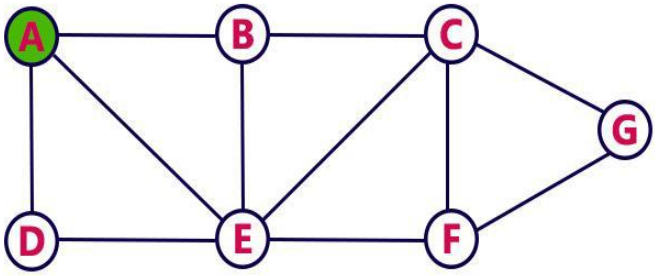
Example :-

Consider the following example graph to perform BFS traversal



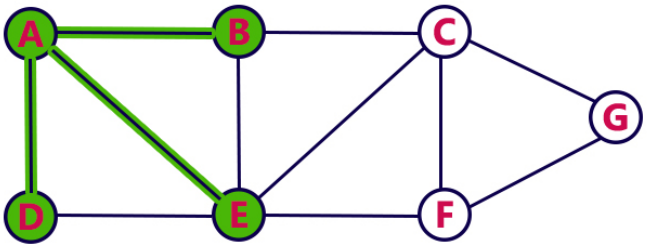
Step 1:

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



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..

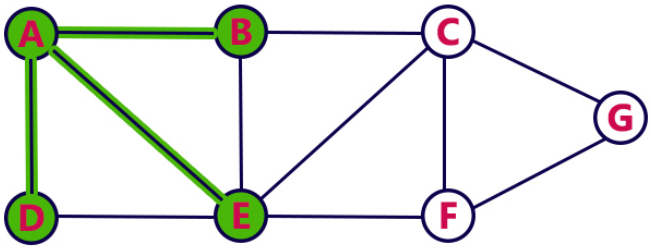


Queue

	D	E	B			

Step 3:

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

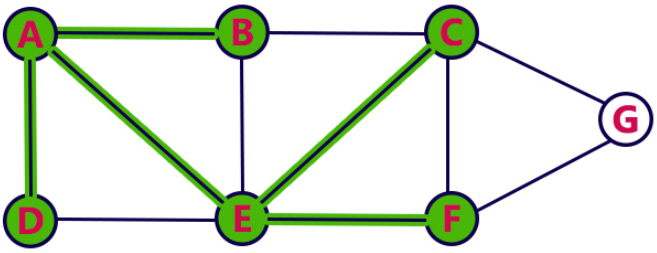


Queue

		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.

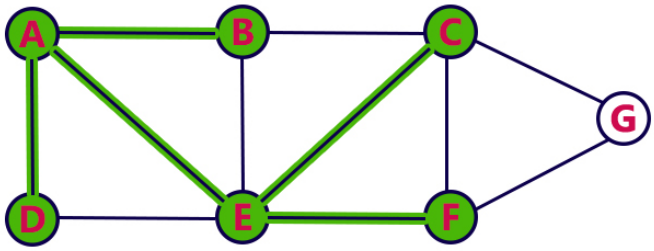


Queue

			B	C	F	

Step 5:

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

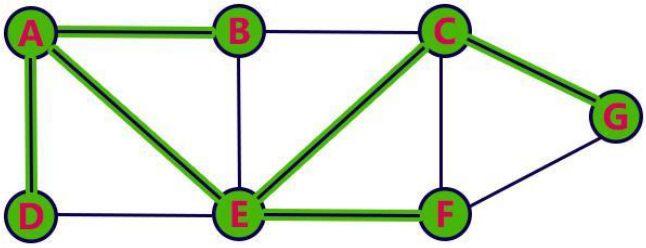


Queue

				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.

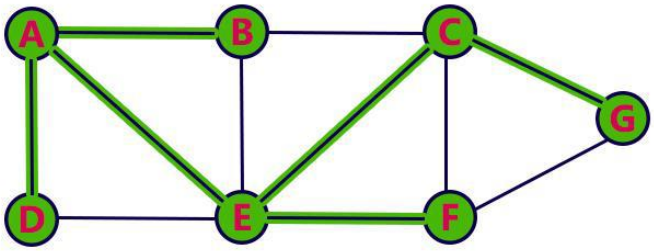


Queue

					F	G

Step 7:

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

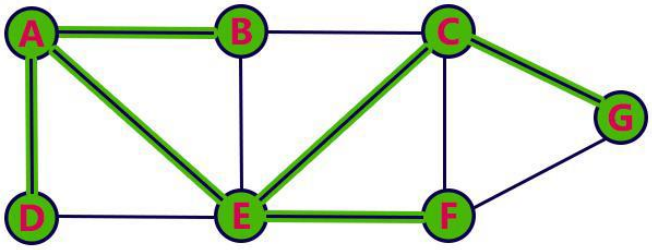


Queue

						G

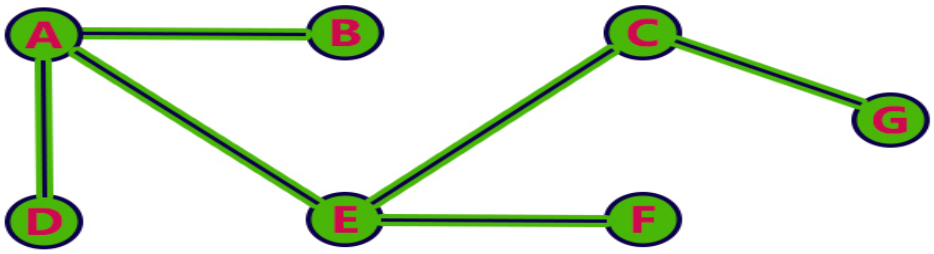
Step 8:

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



Queue

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



Algorithm for breadth-first search

```
Step 1: SET STATUS = 1 (ready state)
        for each node in G
Step 2: Enqueue the starting node A
        and set its STATUS = 2
        (waiting state)
Step 3: Repeat Steps 4 and 5 until
        QUEUE is empty
Step 4: Dequeue a node N. Process it
        and set its STATUS = 3
        (processed state).
Step 5: Enqueue all the neighbours of
        N that are in the ready state
        (whose STATUS = 1) and set
        their STATUS = 2
        (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

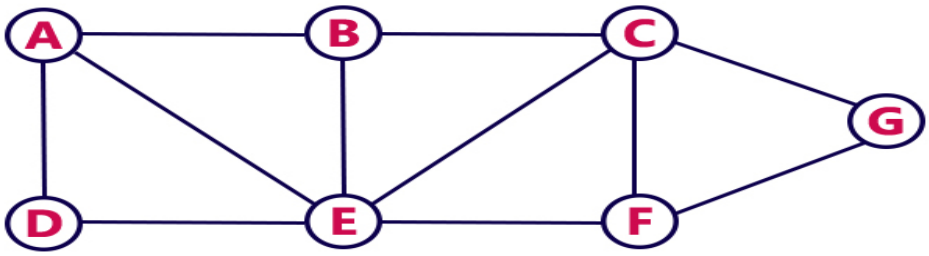
Depth-first Search Algorithm(DFS)

The depth-first search algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.

In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbor of A, then a neighbour of neighbour of A, and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node. Depth-first search are implemented by using stack.

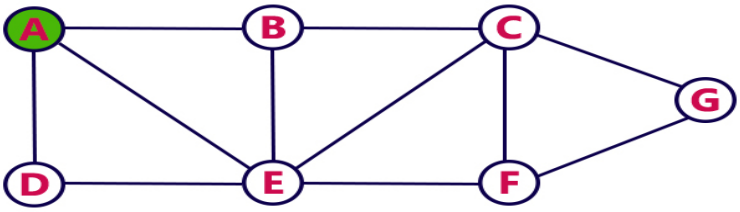
Example:-

Consider the following example graph to perform DFS traversal



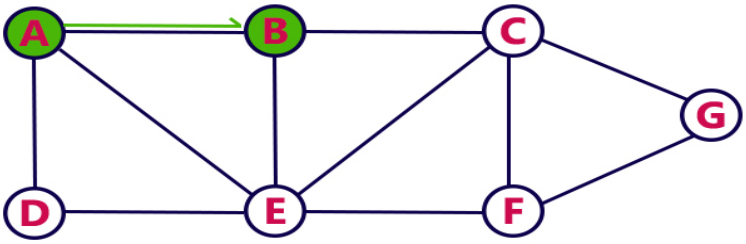
Step 1:

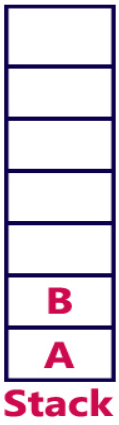
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



Step 2:

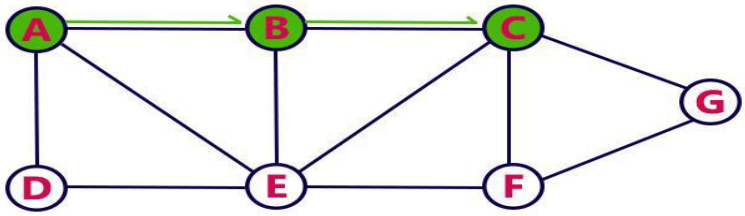
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

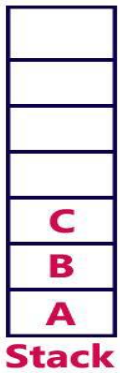




Step 3:

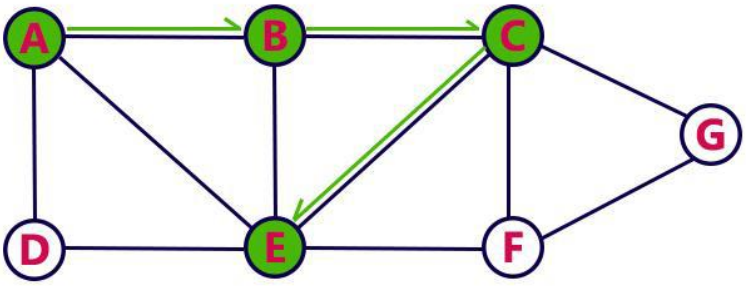
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

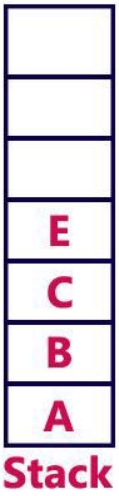




Step 4:

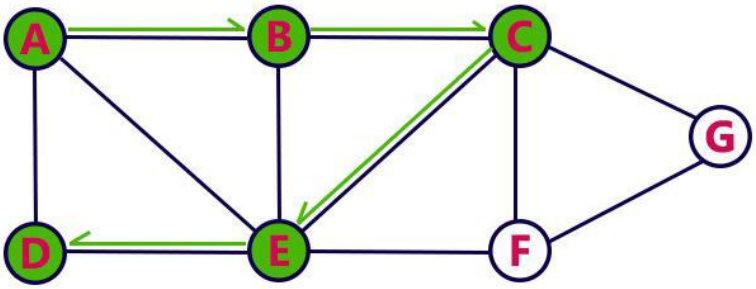
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

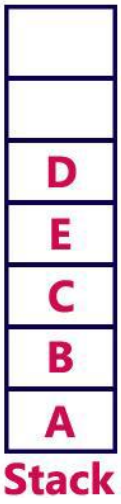




Step 5:

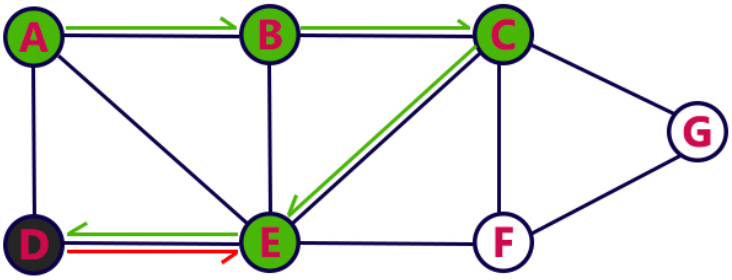
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack





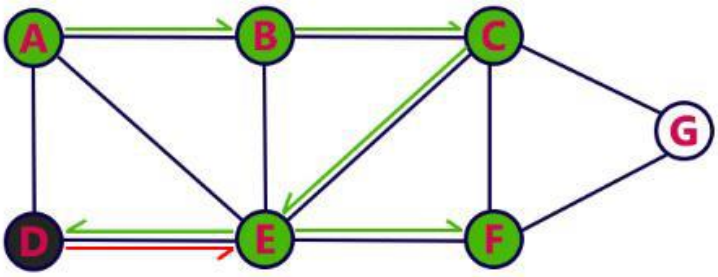
Step 6:

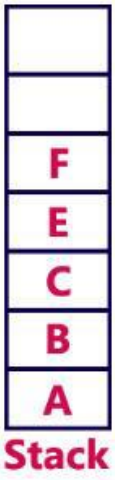
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



Step 7:

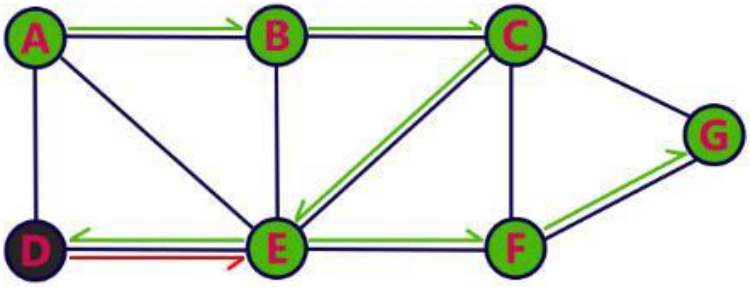
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

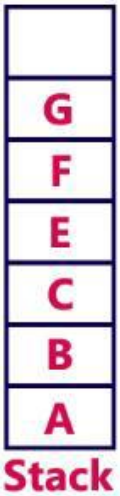




Step 8:

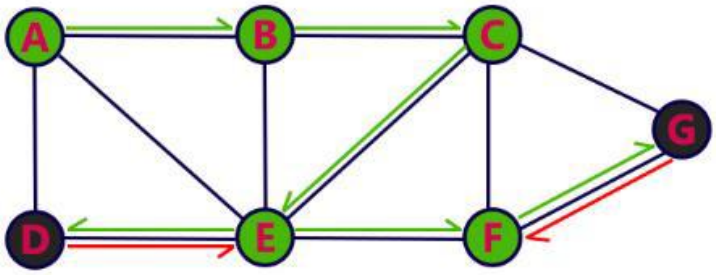
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.





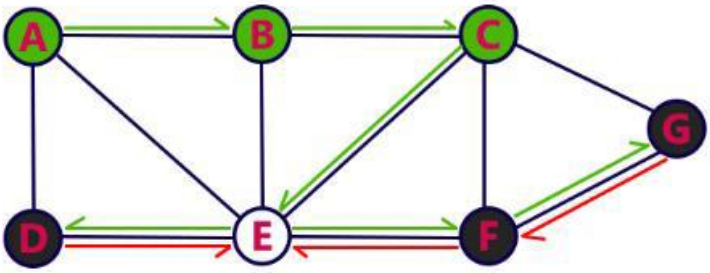
Step 9:

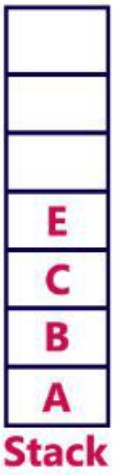
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



Step 10:

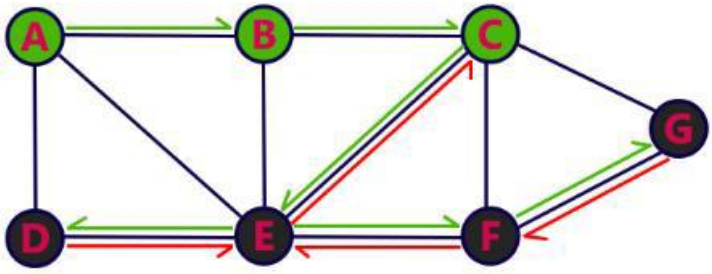
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

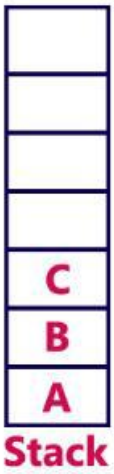




Step 11:

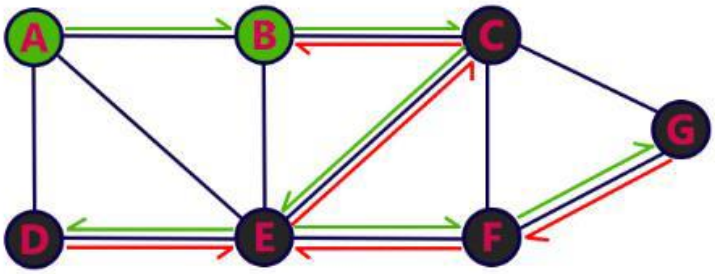
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

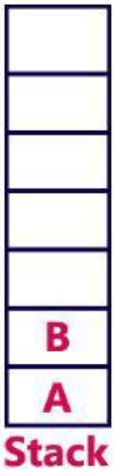




Step 12:

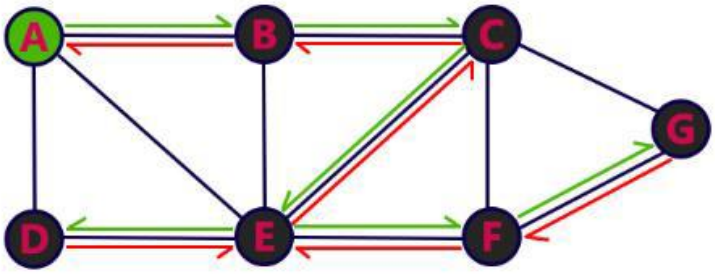
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

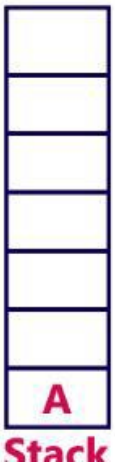




Step 13:

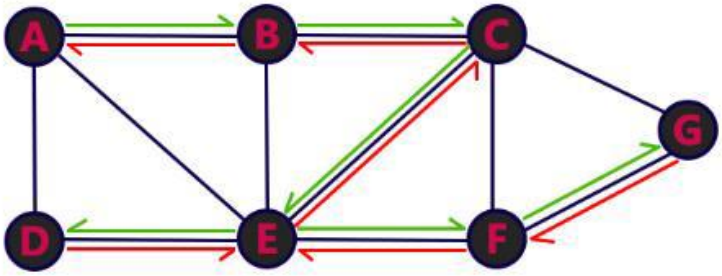
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.



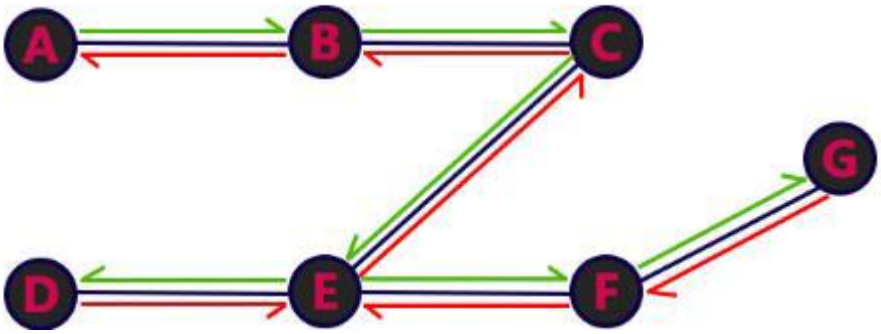


Step 14:

- 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.



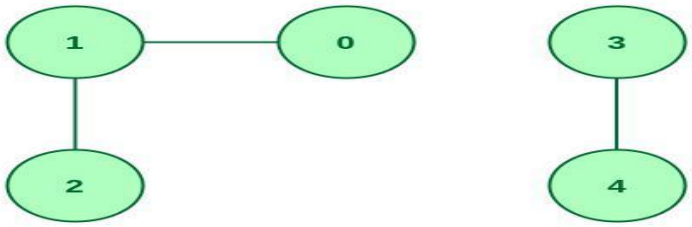
Algorithm for depth-first search

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set
        its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4:   Pop the top node N. Process it and set its
          STATUS = 3 (processed state)
Step 5:   Push on the stack all the neighbours of N that
          are in the ready state (whose STATUS = 1) and
          set their STATUS = 2 (waiting state)
        [END OF LOOP]
Step 6: EXIT
```

CONNECTED COMPONENT

Connected component in an undirected graph refers to a group of vertices that are connected to each other through edges, but not connected to other vertices outside the group.

For example in the graph shown below, {0, 1, 2} form a connected component and {3, 4} form another connected component.

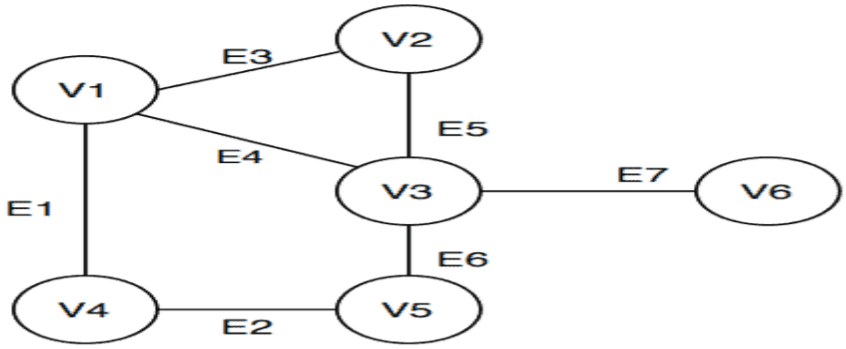


Characteristics of Connected Component:

- A connected component is a set of vertices in a graph that are connected to each other.
- A graph can have multiple connected components.
- Inside a component, each vertex is reachable from every other vertex in that component.

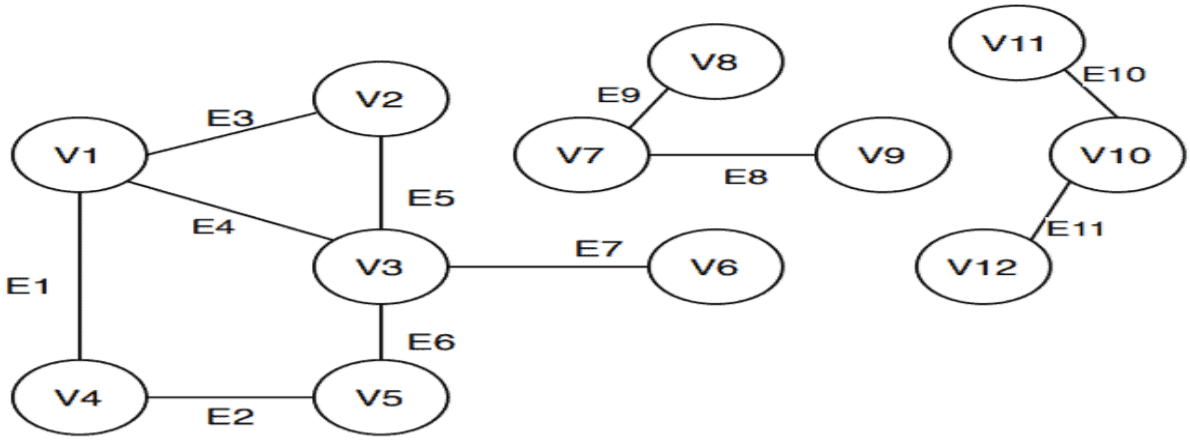
One Connected Component

In this example, the given undirected graph has one connected component:



More Than One Connected Component

In this example, the undirected graph has three connected components:



BI-CONNECTED COMPONENTS

A bi-connected component (or bi-connected component) is a concept from graph theory that applies to undirected graphs. It's a maximal subgraph in which any two vertices are connected to each other by at least two disjoint paths, meaning the removal of any single vertex does not disconnect the subgraph. In other words, it is a component of the graph that remains connected even after the removal of any single vertex.

Key Concepts:

1. **Biconnected Graph**: A graph is biconnected if it is connected and does not contain any articulation points (vertices that, if removed, would disconnect the graph).
2. **Articulation Points**: These are vertices that, when removed, increase the number of connected components in the graph. Biconnected components are formed by removing these points.
3. **Biconnected Components**: Also known as blocks, these are maximal subgraphs where any two vertices are connected by two distinct paths. A graph can have multiple biconnected components, connected to each other at articulation points.

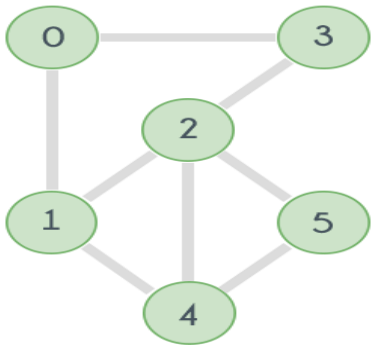
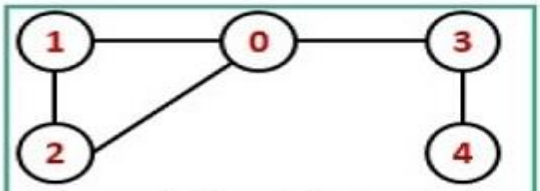


Fig. 1

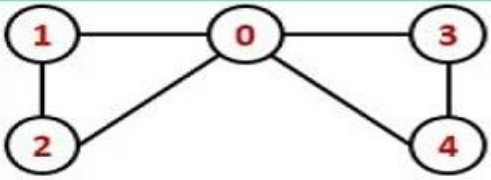
Euler Path and Circuit

The Euler path is a path, by which we can visit every edge exactly once. We can use the same vertices for multiple times.

The Euler Circuit is a special type of Euler path. When the starting vertex of the Euler path is also connected with the ending vertex of that path, then it is called the Euler Circuit.



The graph has Eulerian Paths, for example "4 3 0 1 2 0", but no Eulerian Cycle. Note that there are two vertices with odd degree (4 and 0)



The graph has Eulerian Cycles, for example "2 1 0 3 4 0 2". Note that all vertices have even degree

To detect the path and circuit, we have to follow these conditions −

- The graph must be connected.
- When exactly two vertices have odd degree, it is a Euler Path.
- Now when no vertices of an undirected graph have odd degree, then it is a Euler Circuit.