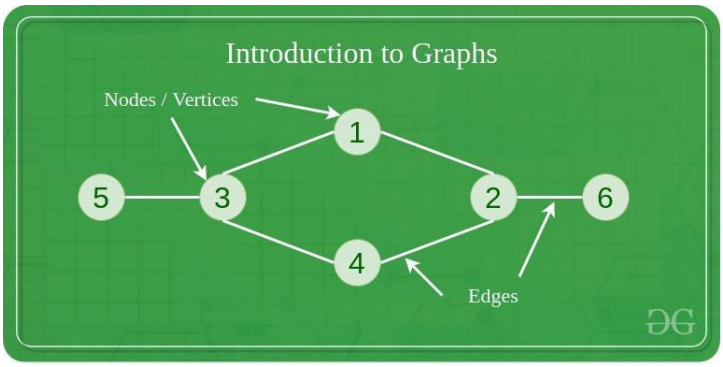
GRAPHS

A **Graph** is a **non-linear data structure** consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices (V) and a set of edges(E). The graph is denoted by G(E, V).

Components of a Graph

- **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labelled or unlabelled.
- **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.



Here are the two most common ways to represent a graph:

1. Adjacency Matrix
2. Adjacency List

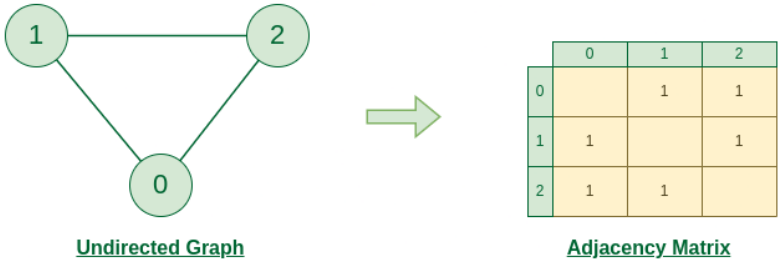
Adjacency Matrix

An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's). Let's assume there are **n** vertices in the graph So, create a 2D matrix **adjMat[n][n]** having dimension n x n.

- *If there is an edge from vertex **i** to **j**, mark **adjMat[i][j]** as **1**.*
- *If there is no edge from vertex **i** to **j**, mark **adjMat[i][j]** as **0**.*

Representation of Undirected Graph to Adjacency Matrix:

The below figure shows an undirected graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** to both cases **adjMat[destination]** and **adjMat[destination])** because we can go either way.



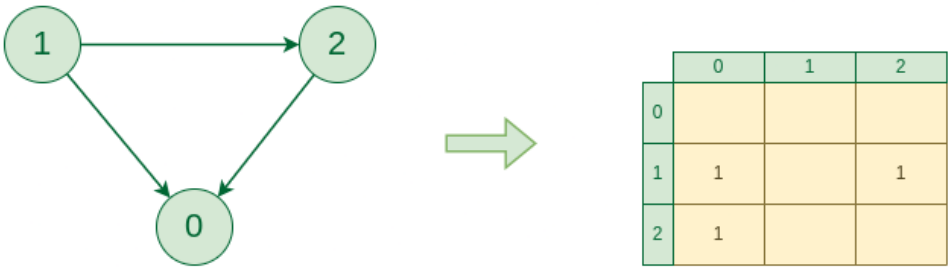
Undirected Graph Adjacency Matrix

Graph Representation of Undirected graph to Adjacency Matrix

Undirected Graph to Adjacency Matrix

Representation of Directed Graph to Adjacency Matrix:

The below figure shows a directed graph. Initially, the entire Matrix is initialized to **0**. If there is an edge from source to destination, we insert **1** for that particular **adjMat[destination]**.



Directed Graph Adjacency Matrix

Graph Representation of Directed graph to Adjacency Matrix

Directed Graph to Adjacency Matrix

Adjacency List

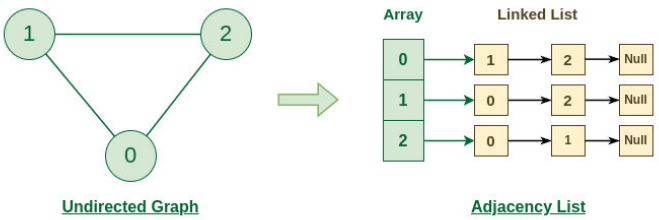
An array of Lists is used to store edges between two vertices. The size of array is equal to the number of **vertices (i.e, n)**. Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex **i**.

Let's assume there are **n** vertices in the graph So, create an **array of list** of size **n** as **adjList[n].**

- *adjList[0] will have all the nodes which are connected (neighbour) to vertex **0**.*
- *adjList[1] will have all the nodes which are connected (neighbour) to vertex **1** and so on.*

Representation of Undirected Graph to Adjacency list:

The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 0) So, insert vertices 2 and 0 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



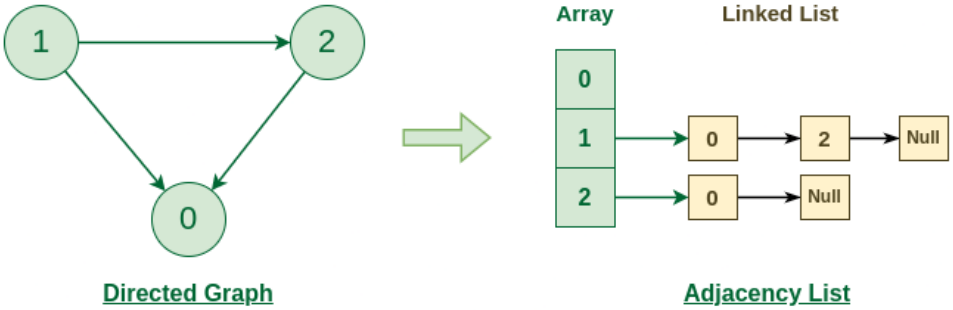
Undirected Graph Adjacency List

Graph Representation of Undirected graph to Adjacency List

Undirected Graph to Adjacency list

Representation of Directed Graph to Adjacency list:

The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.



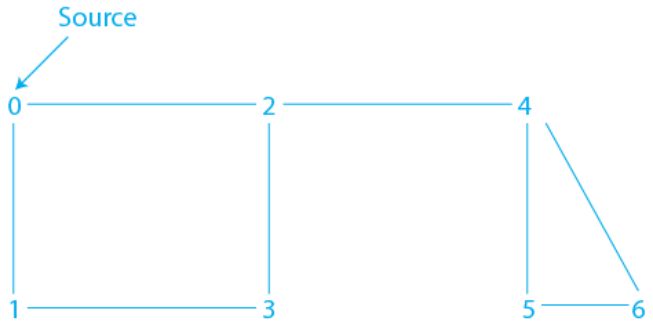
Graph Representation of Directed graph to Adjacency List

Graph Traversal:

Breadth-First Search (BFS) is a fundamental graph traversal algorithm used to explore and analyze graphs and trees in a systematic manner. It starts from a specified source vertex and visits all the vertices in the graph, level by level, before moving to the next depth level. This approach ensures that vertices at lower levels are visited before vertices at deeper levels, making BFS particularly useful for finding the shortest path between nodes and solving problems where proximity matters. BFS program in C can be implemented using data structures like queues to manage the order in which vertices are visited, providing an efficient and reliable way to explore the interconnected structure of graphs and trees.

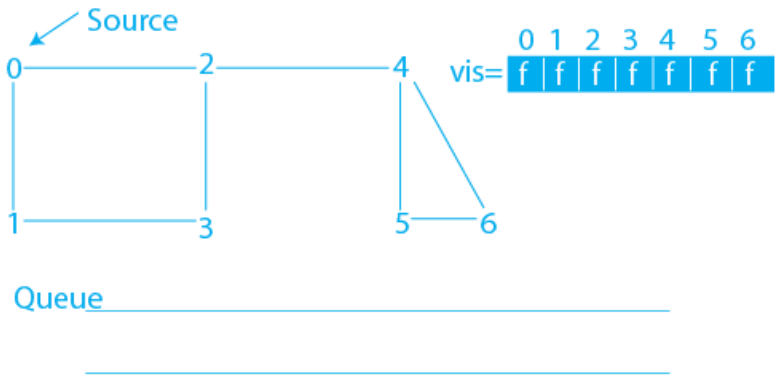
What is Breadth First Search (BFS)?

Consider the graph given below.



So, in the undirected graph shown above, 0 is the source vertex i.e. starting from 0, we have to do the BFS traversal.

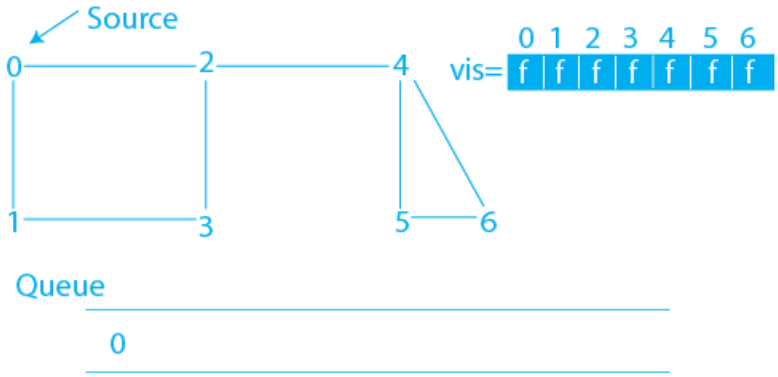
Consider the graph, an empty queue and a boolean array “visited” as shown below.



The visited array contains all the values “False” initially indicating that we have not visited any vertex till now. So, we know that the source is given to us.

The initial step is to put the source into the queue. Please note that the vertices will be marked visited when they will exit the queue, **not while entering the queue.**

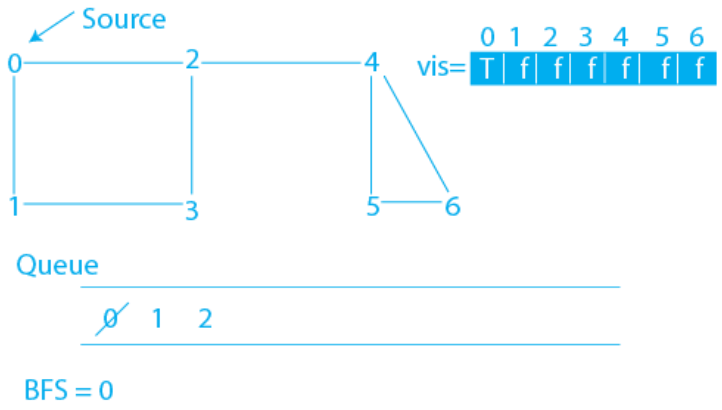
So, the source is kept in the queue as shown below.



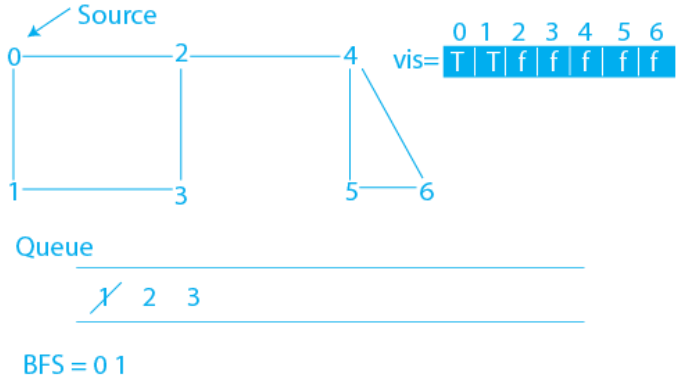
Now, start the repeating steps. So, perform the following repeating steps now till the queue is empty.

1. Take a vertex out of the queue.
2. Mark the vertex as visited in the “visited” boolean array.
3. Add this vertex to the BFS traversal.
4. Add all the unvisited neighbors of this vertex into the queue.

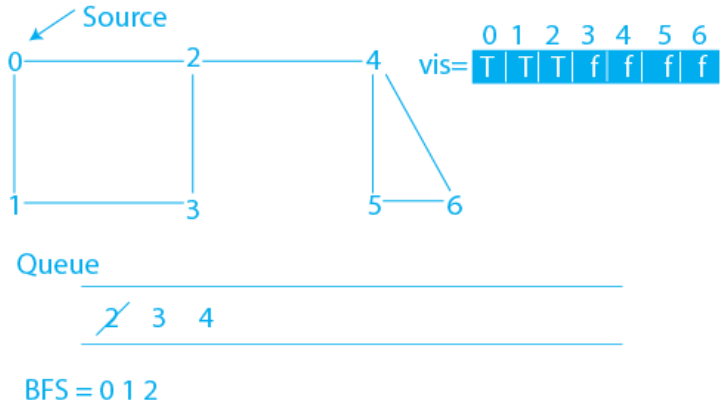
So, follow the step shown above. Remove 0 from the queue, mark it visited, add 0 to the BFS, and add all the unvisited neighbors of 0 (i.e. 1 and 2) into the queue.



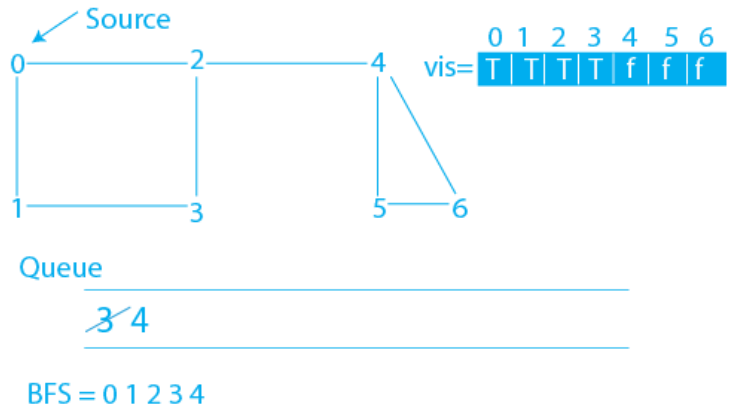
Now, as already discussed, repeat the 4 steps again and again until the queue is empty. Since the queue is not empty right now, we again repeat the 4 steps. So, we will dequeue vertex 1 from the queue, mark it visited, add it to the BFS, and will add 3 to the queue. This is because 0 (which is also a neighbor of vertex 1) is already visited.



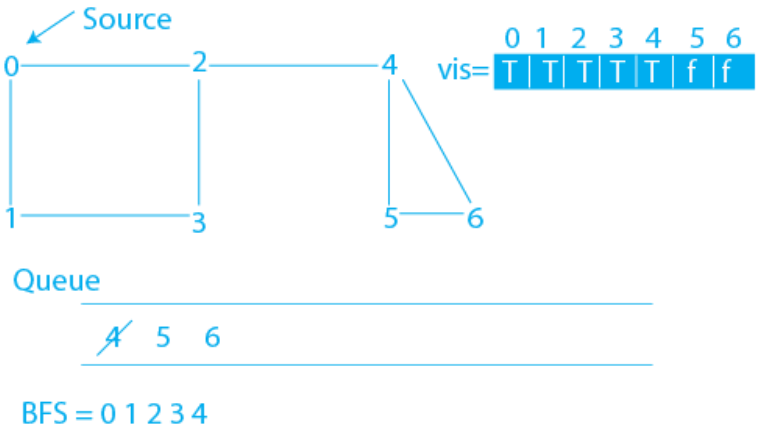
Since the queue is not empty now also, we will again dequeue the next vertex. So, vertex 2 will be dequeued from the queue, will be marked visited, will be added to the BFS, and its unvisited neighbor i.e. vertex 4 will be added to the queue.



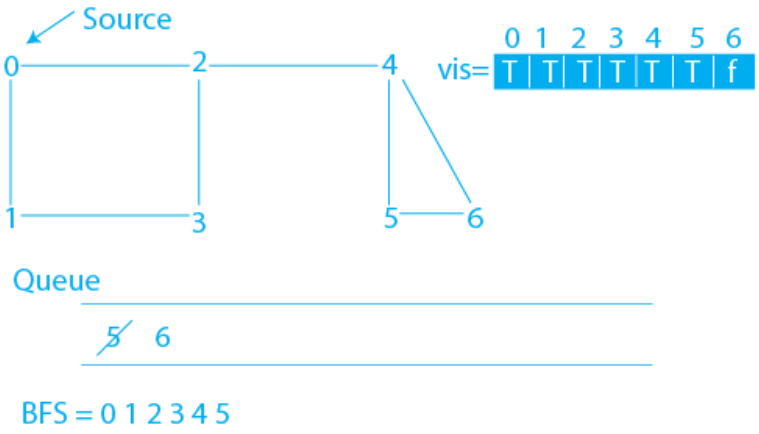
So, the queue is not empty yet. Again, we will now take out vertex 3 from the queue, mark it visited, add it to the BFS, and will add its unvisited neighbor to the queue. Since 3 does not have any unvisited neighbors, nothing will be added to the queue in this step.



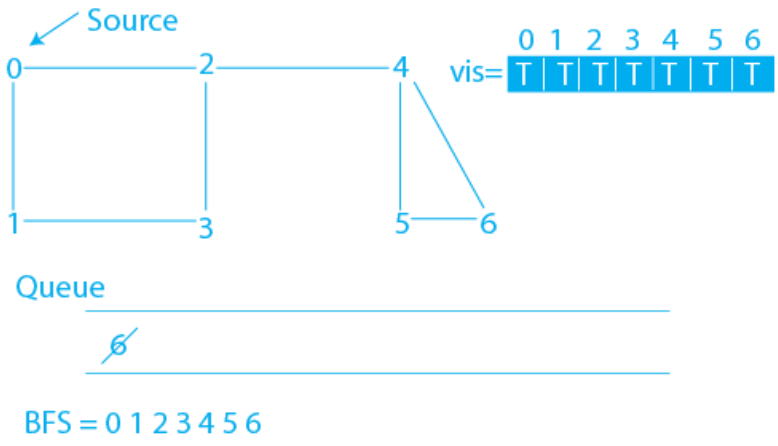
Next, we remove vertex 4, mark it visited, add it to the BFS, and add its unvisited neighbors (5 and 6) to the queue.



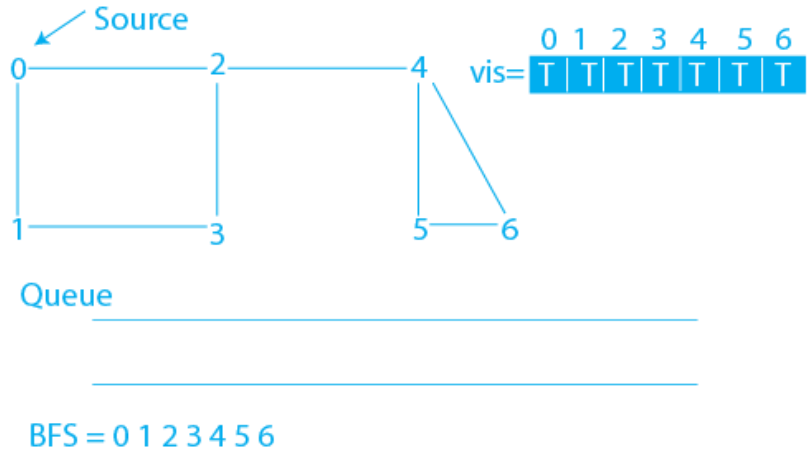
Vertex 5 will be removed from queue no.w It will be marked visited and will be added to the BFS. Since there are no unvisited neighbors of vertex 5, nothing will be added to the queue.



Finally, we will remove vertex 6 from the queue. We will mark it visited, and add it to the BFS, and since there are no unvisited neighbors of vertex 6, nothing will be added to the queue.



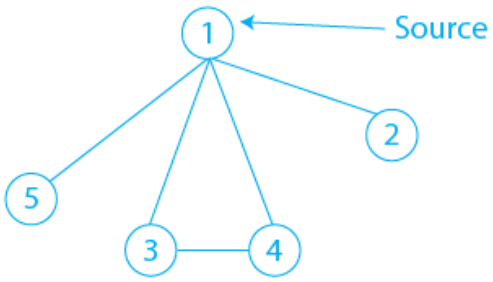
The queue has now become empty and we have to stop our procedure. So, we have got BFS as shown below.



Depth First Search:

DFS stands for Depth First Search and it is used to traverse all the nodes of the graph. With this technique, first, we pick and node and after that, we traverse through all the possible nodes in that path or branch, and then we will do backtracking.

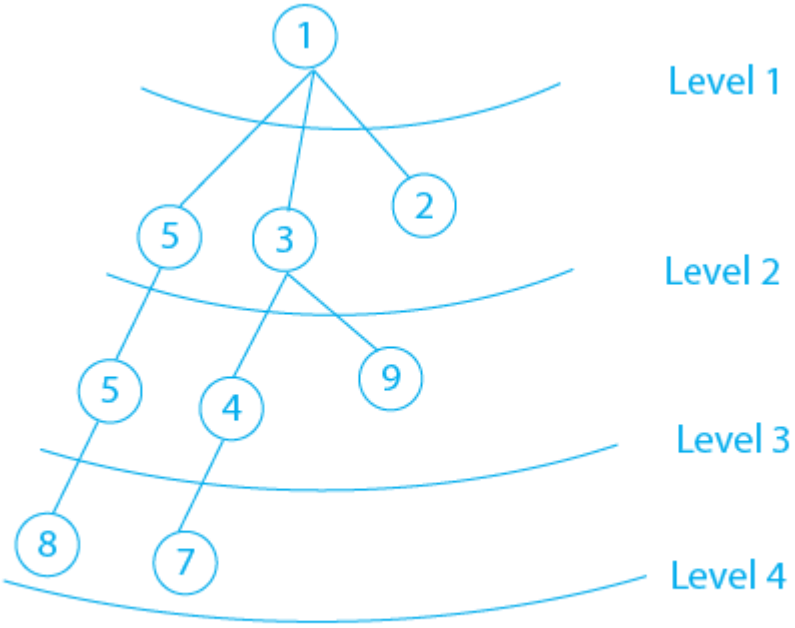
Backtracking means when the DFS algorithm reaches a particular node that does not have any neighbors to visit. When it reaches a node that does not have any neighbors after that, it will backtrack to its previous node. Let's see an example of DFS.



In the above image, the DFS algorithm will start from the source node which is 1 and it has four neighbors (5, 3, 4, 2) we can choose anyone of them. If we choose 5 then our vised nodes are 1 and 5. Now, 5 does not have any neighbors so we will backtrack to its previous node which is 1 and now 1 has three non-visited neighbors (3, 4, 2). If we choose 3 then our path will become 1,5,3. Now, 3 has one non-visited neighbor(4) so we can choose only that node, and our path becomes 1,5,3,4. Now, 4 does not have any non-visited nodes thus we have to backtrack to the previous node 3 and it also does not have any non-visited neighbors thus we will backtrack to previous node 1, and 1 has only one non-visited node (2). We will go to 2 and our path will become 1,5,3,4,2. So the path becomes 1->5->3->4->2 if we use DFS traversal.

How DFS is different from BFS?

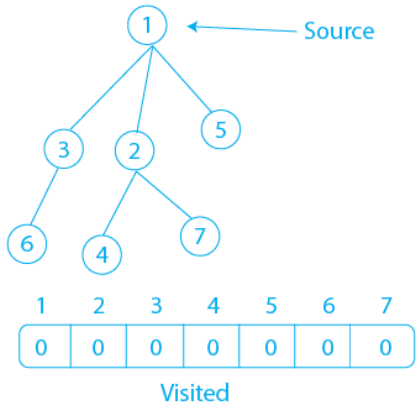
BFS stands for Breadth First Search. In this technique, we traverse through a graph level vise. We need to use the Queue data structure to perform BFS traversal. Let's see how BFS traversal works using an example.



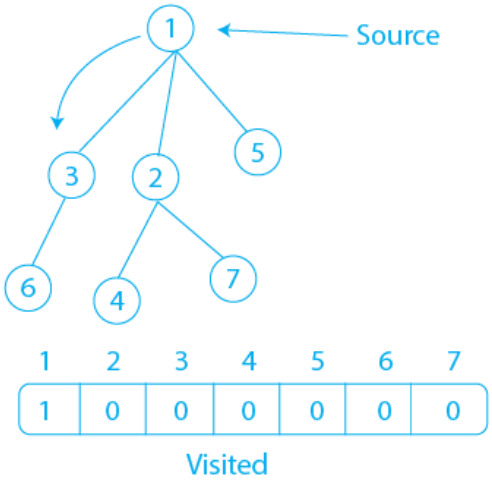
In the above example, first, we will traverse through level 1 thus our path will become 1. After that, we will traverse through level 2 and now our path will be 1, 5, 3, 2. Now, we will traverse through level 3 and our new path will become 1, 5, 3, 2, 6, 4, 9. At the last, we will traverse through level 4 and our new path will become 1, 5, 3, 2, 6, 4, 9, 8, 7. If we use the same example for DFS traversal then our path will become 1, 5, 6, 8, 3, 4, 7, 9, 2.

How DFS works?

Let's see how DFS traversal works using an example. We will also see a dry run of the example to understand it more clearly.

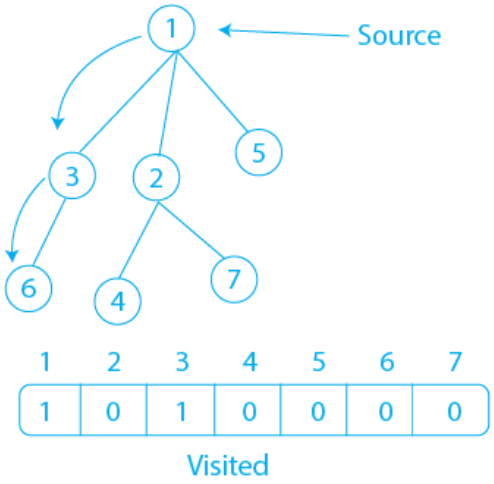


In the above example, we have taken one graph and have also taken an array to keep track of visited and non-visited nodes and this array is initialized with 0 which means all the nodes are non-visited. In this example, we will start traversing from source node 1. Let's see further steps of DFS traversal.

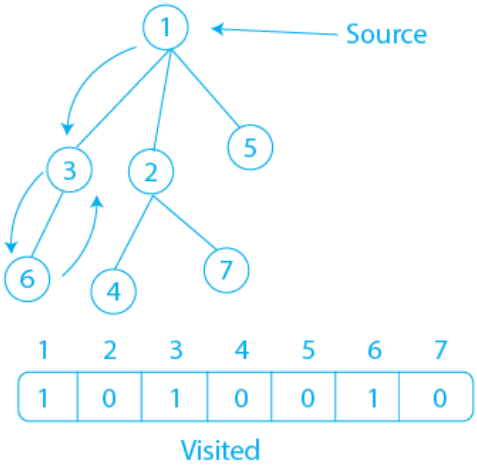


In this step, first, we will mark node 1 as visited and we will look for non-visited neighbors. Node 1 has three non-visited neighbors 3, 2, and 5. We can choose anyone out of these three. We will now go for Node 3.

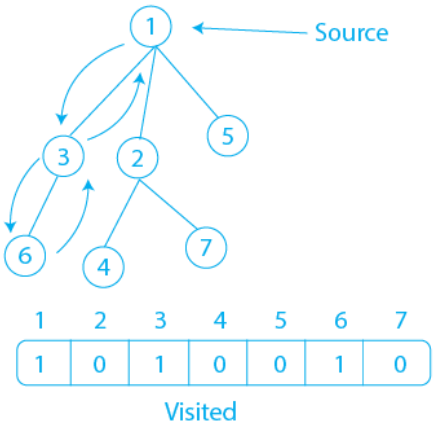
Path till now: 1



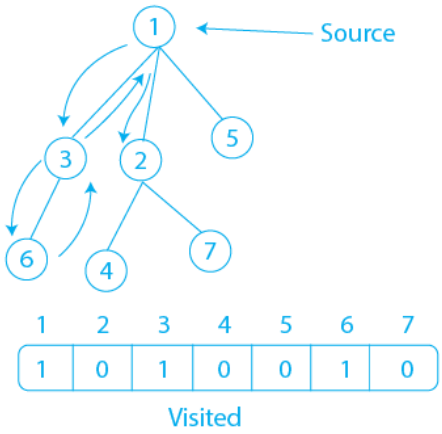
In this step, first, we will mark node 3 as a visited. Node 3 has only one non-visited neighbor 6. Thus, we have the only choice to go for node 6.
Path till now: 1 -> 3



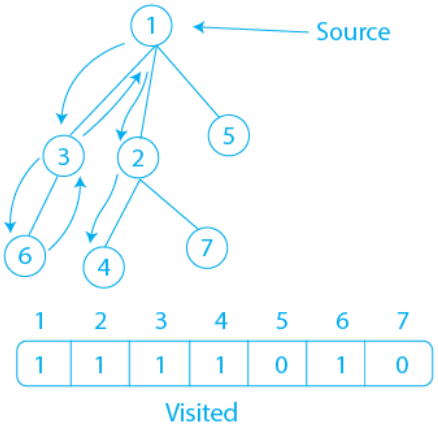
In this step, first, we will mark node 6 as visited. Node 6 does not have any non-visited neighbors so we need to backtrack to its previous node which is 3.
Path till now: 1 -> 3 -> 6



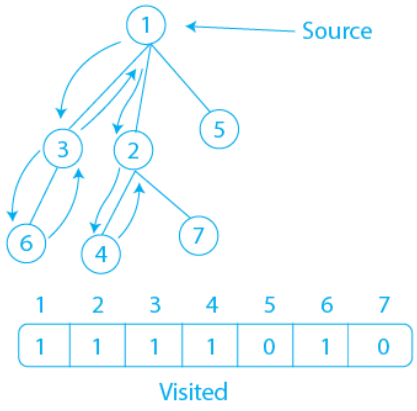
In this step, node 3 is already visited. Node 3 does not have any non-visited neighbors. Thus, we will backtrack to the previous node of 3 which is 1.
Path till now: 1 -> 3 -> 6



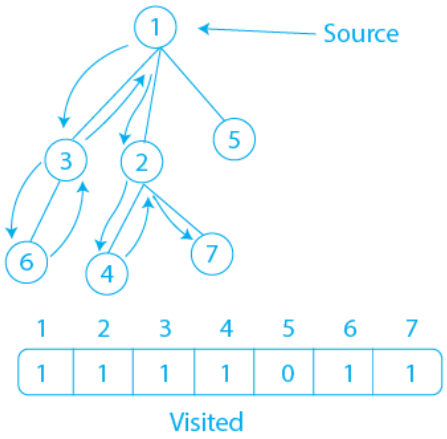
In this step, node 1 is already visited. Node 1 has two non-visited neighbors which are 2 and 5. We can choose any one out of these two nodes. We will choose node 2 first.
Path till now: 1 -> 3 -> 6



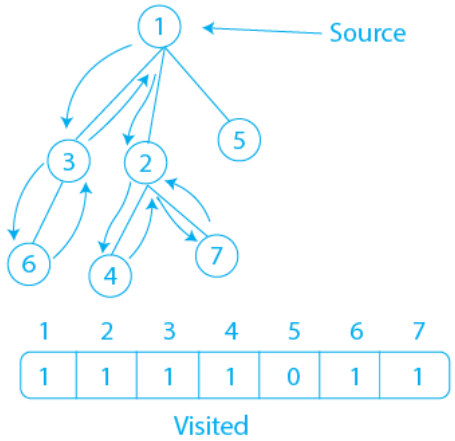
In this step, first, we will mark node 2 as a visited. Node 2 has two non-visited neighbors which are 4 and 7. We can choose any one out of these two. We will go for node 4 first.
Path till now: 1 -> 3 -> 6 -> 2



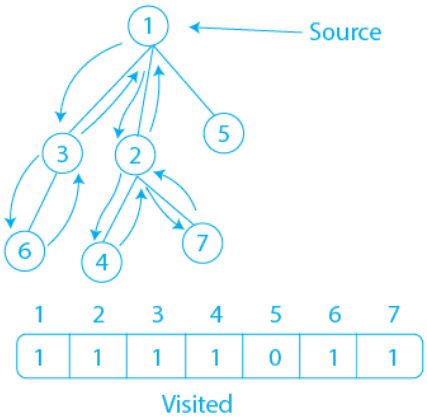
In this step, first, we will mark node 4 as visited. Node 4 does not have any non-visited neighbors so we need to backtrack to its previous node which is 2.
Path till now: 1 -> 3 -> 6 -> 2 -> 4



In this step, node 2 is already visited. Node 2 has only one non-visited neighbor 7. Thus, we have the only choice to go for node 7.
Path till now: 1 -> 3 -> 6 -> 2 -> 4



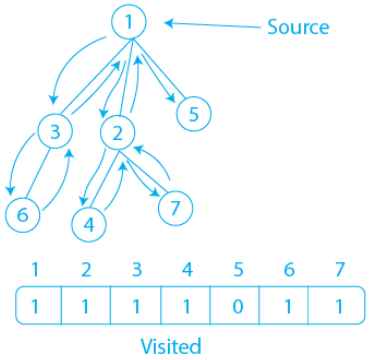
In this step, first, we will mark node 7 as visited. Node 7 does not have any non-visited neighbors so we need to backtrack to its previous node which is 2
Path till now: 1 -> 3 -> 6 -> 2 -> 4 -> 7



.

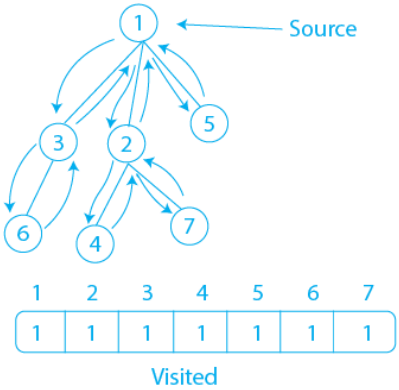
In this step, node 2 is already visited. Node 2 does not have any non-visited neighbors so we need to backtrack to its previous node which is 1.

Path till now: 1 -> 3 -> 6 -> 2 -> 4 -> 7



In this step, node 1 is already visited. Node 1 has only one non-visited neighbor which is 5. We can choose only one node which is 5. We will choose node 5.

Path till now: 1 -> 3 -> 6 -> 2 -> 4 -> 7



In this step, first, we will mark node 5 as visited. Node 5 does not have any non-visited neighbours so we need to backtrack to its previous node which is 1. And now 1 also does not have any non-visited neighbours so DFS traversal will end here.

Path till now: 1 -> 3 -> 6 -> 2 -> 4 -> 7 -> 5