**Graph traversals**

* **BFS:**

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

**BFS pseudocode:**

Initialize a queue

Initialize a visited hash set (this is just one way to keep track of visited nodes)

Add start node to queue.

Add start node's value to visited set.

While the queue's size is not empty:

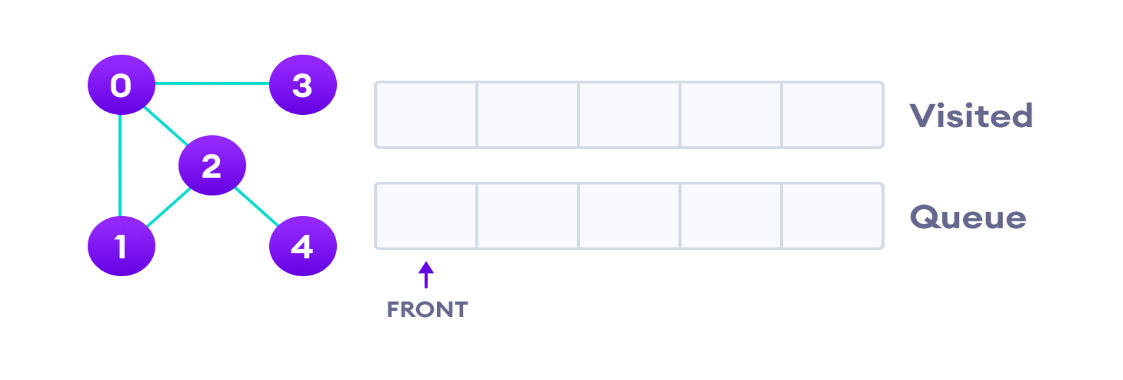
Remove a node.

Print the node's value.

Add all unvisited neighbors into the queue and visited set.

end

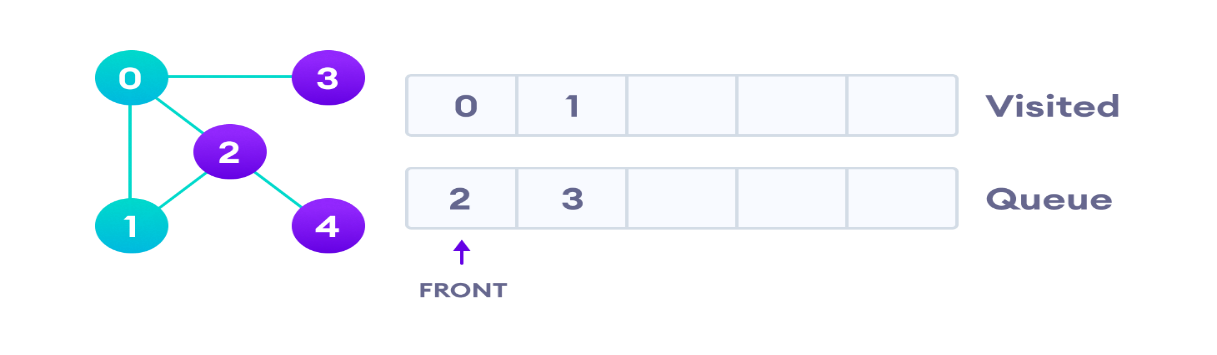
**Example:**



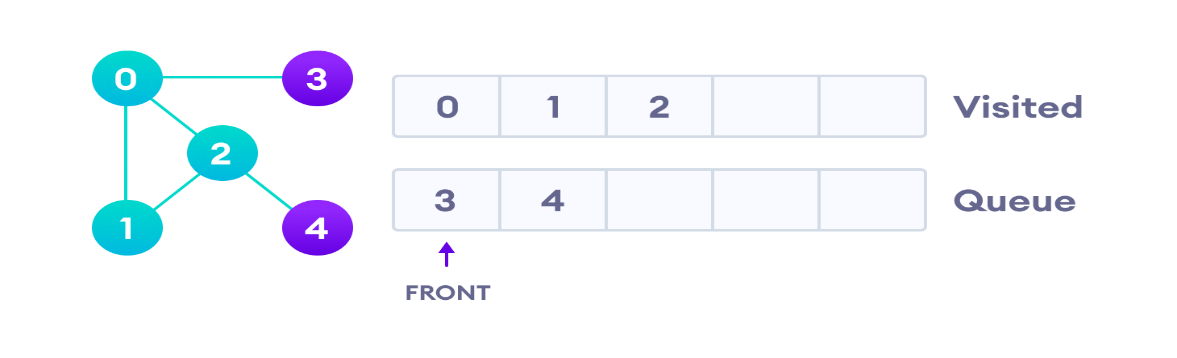
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

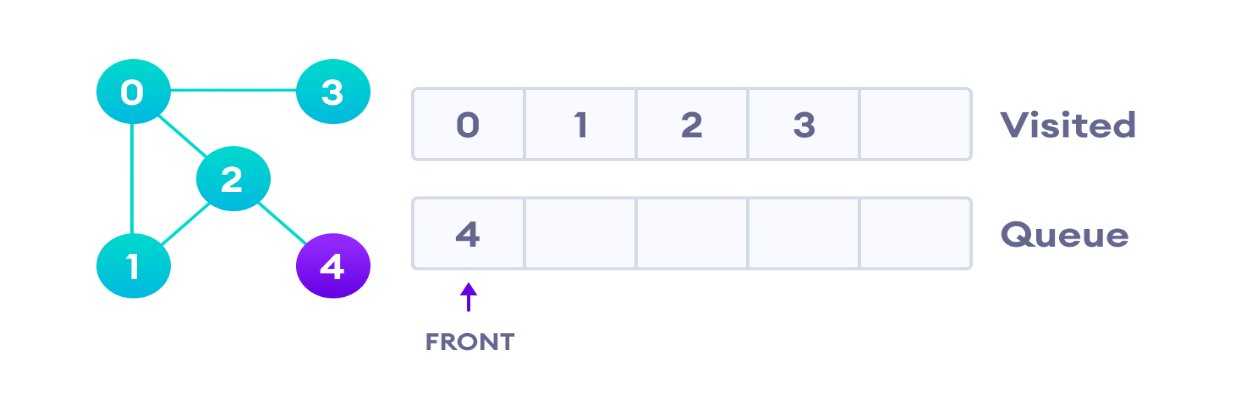


Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

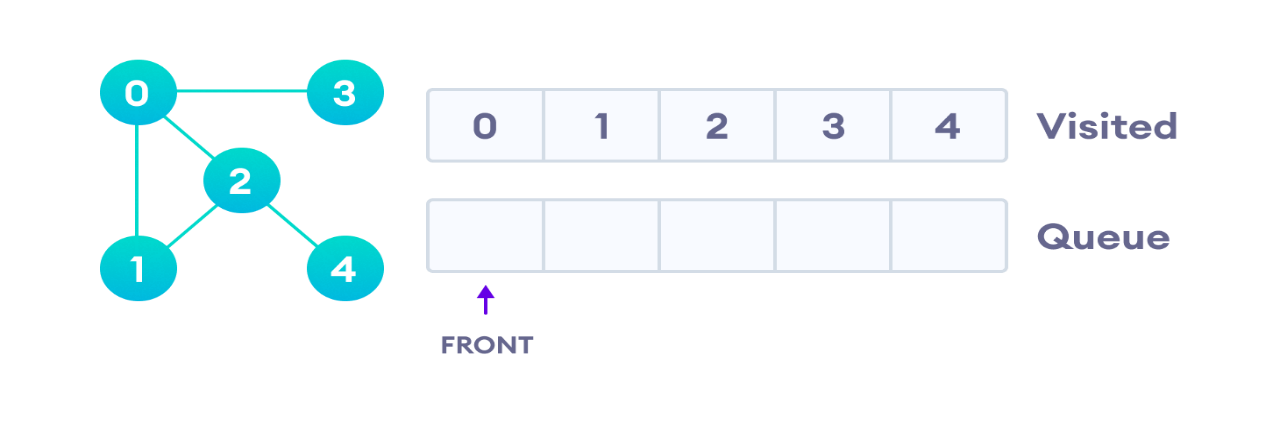


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.





Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.



Since the queue is empty, we have completed the Breadth First Traversal of the graph.

**Implementation:**

// BFS algorithm in Java

import java.util.\*;

public class Graph

{

private int V;

private LinkedList<Integer> adj[];

// Create a graph

Graph(int v)

{

V = v;

adj = new LinkedList[v];

for (int i = 0; i < v; ++i)

adj[i] = new LinkedList();

}

// Add edges to the graph

void addEdge(int v, int w)

{

adj[v].add(w);

}

// BFS algorithm

void BFS(int s)

{

boolean visited[] = new boolean[V];

LinkedList<Integer> queue = new LinkedList();

visited[s] = true;

queue.add(s);

while (queue.size() != 0)

{

s = queue.poll();

System.out.print(s + " ");

Iterator<Integer> i = adj[s].listIterator();

while (i.hasNext())

{

int n = i.next();

if (!visited[n])

{

visited[n] = true;

queue.add(n);

}

}

}

}

public static void main(String args[])

{

Graph g = new Graph(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

System.out.println("Following is Breadth First Traversal " + "(starting from vertex 2)");

g.BFS(2);

}

}

**BFS Algorithm Applications**

1. To build index by search index
2. For GPS navigation
3. Path finding algorithms
4. In Ford-Fulkerson algorithm to find maximum flow in a network
5. Cycle detection in an undirected graph
6. In [minimum spanning tree](https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree)

* **DFS:**

A standard DFS implementation puts each vertex of the graph into one of two categories:

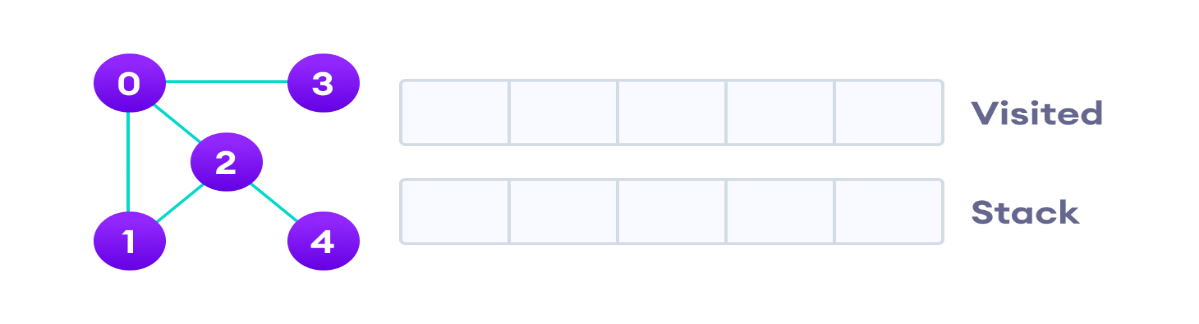
1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

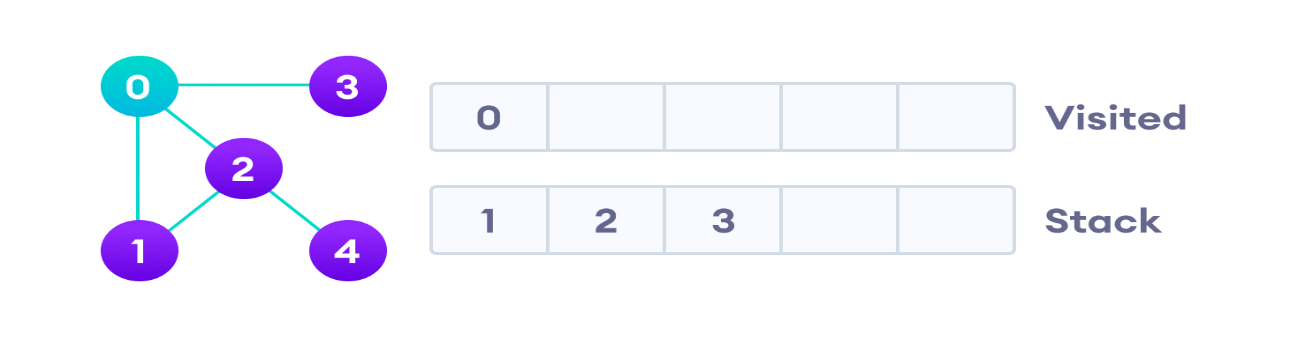
The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

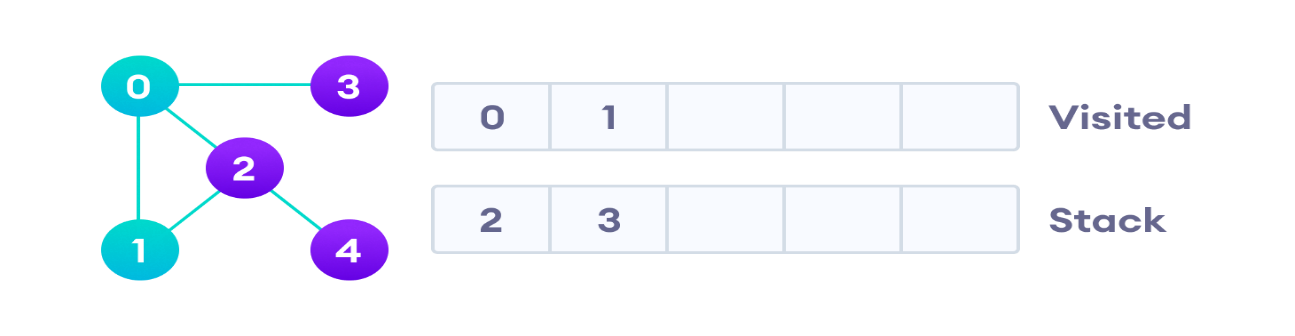
## Depth First Search Example



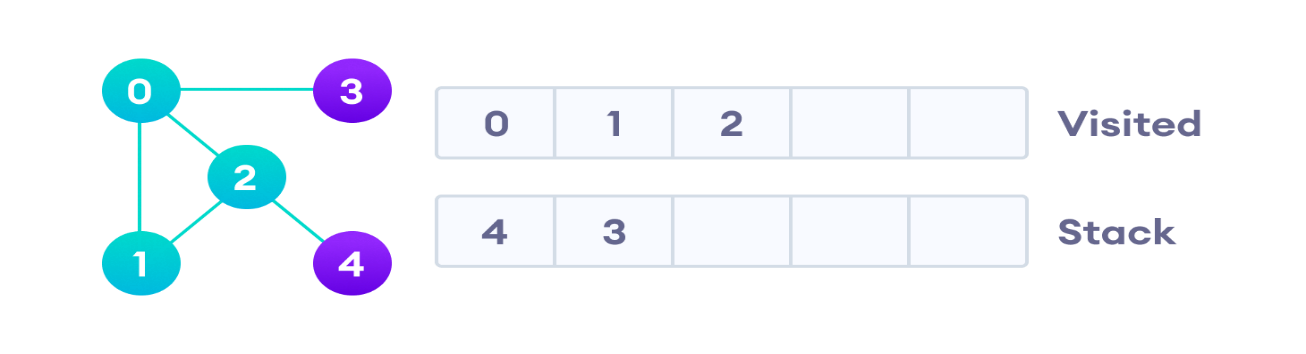
We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

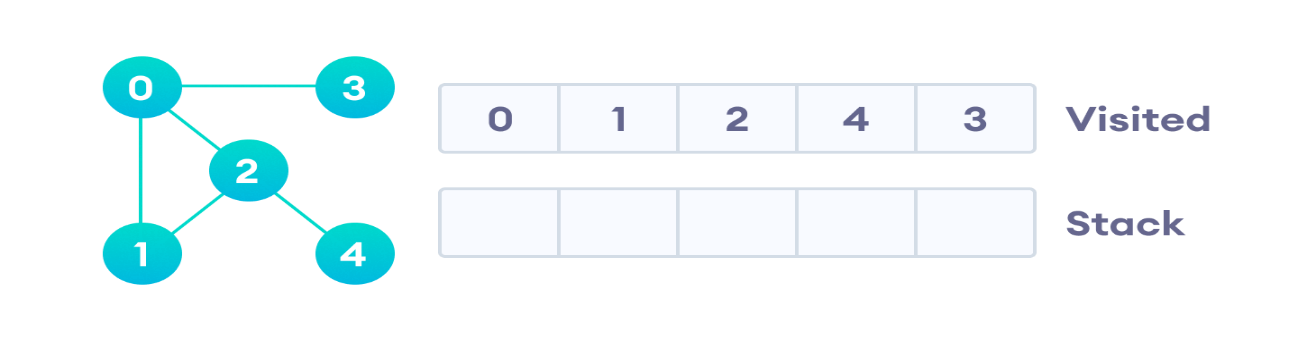


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.





After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

**Implementation:**

// DFS algorithm in Java

import java.util.\*;

class Graph

{

private LinkedList<Integer> adjLists[];

private boolean visited[];

// Graph creation

Graph(int vertices)

{

adjLists = new LinkedList[vertices];

visited = new boolean[vertices];

for (int i = 0; i < vertices; i++)

adjLists[i] = new LinkedList<Integer>();

}

// Add edges

void addEdge(int src, int dest)

{

adjLists[src].add(dest);

}

// DFS algorithm

void DFS(int vertex)

{

visited[vertex] = true;

System.out.print(vertex + " ");

Iterator<Integer> ite = adjLists[vertex].listIterator();

while (ite.hasNext())

{

int adj = ite.next();

if (!visited[adj])

DFS(adj);

}

}

public static void main(String args[])

{

Graph g = new Graph(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 3);

System.out.println("Following is Depth First Traversal");

g.DFS(2);

}

}

## Application of DFS Algorithm

1. For finding the path
2. To test if the graph is bipartite
3. For finding the strongly connected components of a graph
4. For detecting cycles in a graph