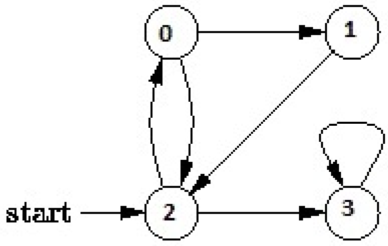
Breadth First Traversal or BFS for a Graph

Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree (See method 2 of this post). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.
For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.

Following are C++ and Java implementations of simple Breadth First Traversal from a given source.

The C++ implementation uses adjacency list representation of graphs. STL's list container is used to store lists of adjacent nodes and queue of nodes needed for BFS traversal.

C++

```
// Program to print BFS traversal from a given source vertex. BFS(int s)
// traverses vertices reachable from s.
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using adjacency list representation
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // Pointer to an array containing adjacency lists
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v’s list.
}

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queue;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue.push_back(s);

    // 'i' will be used to get all adjacent vertices of a vertex
    list<int>::iterator i;

    while(!queue.empty())
    {
        // Dequeue a vertex from queue and print it
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        // Get all adjacent vertices of the dequeued vertex s
        // If a adjacent has not been visited, then mark it visited
        // and enqueue it
        for(i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if(!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}

// Driver program to test methods of graph class
int main()
{
    // Create a graph given in the above diagram
    Graph g(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);

    cout << "Following is Breadth First Traversal "
         << "(starting from vertex 2) \n";
    g.BFS(2);

    return 0;
}
```

Run on IDE

```
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
    private int V;   // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency Lists

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
    }

    // prints BFS traversal from a given source s
    void BFS(int s)
    {
        // Mark all the vertices as not visited(By default
        // set as false)
        boolean visited[] = new boolean[V];

        // Create a queue for BFS
        LinkedList<Integer> queue = new LinkedList<Integer>();

        // Mark the current node as visited and enqueue it
        visited[s]=true;
        queue.add(s);

        while (queue.size() != 0)
        {
            // Dequeue a vertex from queue and print it
            s = queue.poll();
            System.out.print(s+" ");

            // Get all adjacent vertices of the dequeued vertex s
            // If a adjacent has not been visited, then mark it
            // visited and enqueue it
            Iterator<Integer> i = adj[s].listIterator();
            while (i.hasNext())
            {
                int n = i.next();
                if (!visited[n])
                {
                    visited[n] = true;
                    queue.add(n);
                }
            }
        }
    }

    // Driver method to
    public static void main(String args[])
    {
        Graph g = new Graph(4);
```

```
                                        (starting from vertex 2) ");

        g.BFS(2);
    }
}
// This code is contributed by Aakash Hasija
```

Run on IDE

Python

```
# Program to print BFS traversal from a given source
# vertex. BFS(int s) traverses vertices reachable
# from s.
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False]*(len(self.graph))

        # Create a queue for BFS
        queue = []

        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

            # Dequeue a vertex from queue and print it
            s = queue.pop(0)
            print s,

            # Get all adjacent vertices of the dequeued
            # vertex s. If a adjacent has not been visited,
            # then mark it visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True


# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
```

```
# This code is contributed by Neelam Yadav
```

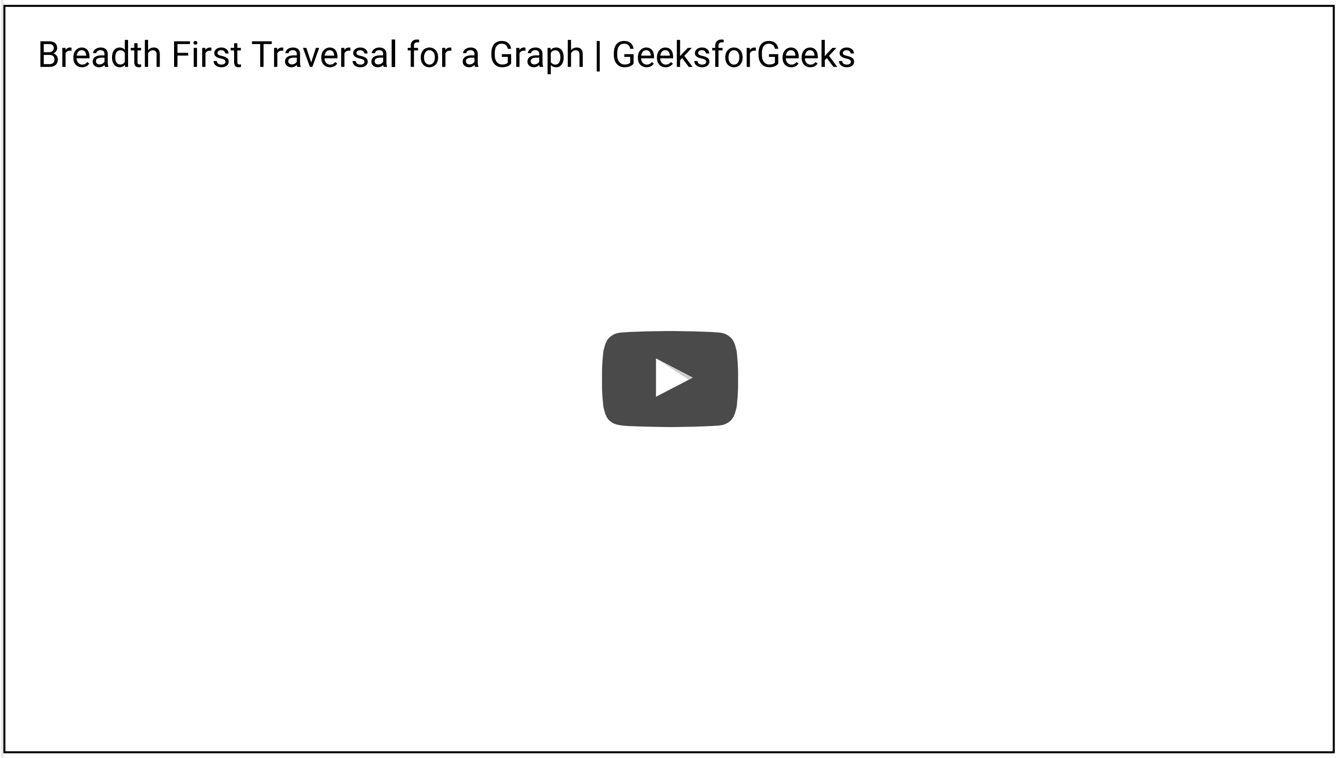
Run on IDE

Output:

```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

Note that the above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To print all the vertices, we can modify the BFS function to do traversal starting from all nodes one by one (Like the DFS modified version) .

Time Complexity: O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.



You may like to see below also :

- Depth First Traversal
- Applications of Breadth First Traversal
- Applications of Depth First Search

GATE CS Corner **Company Wise Coding Practice**

Graph Queue BFS

Recommended Posts:

Depth First Traversal or DFS for a Graph
Level Order Tree Traversal
Graph and its representations
Applications of Depth First Search
Applications of Breadth First Traversal

(Login to Rate and Mark)

2.3 Average Difficulty : **2.3/5.0**
Based on **192** vote(s)

□ Add to TODO List
□ Mark as DONE

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments Share this post!

Contact Us! About Us! Advertise with us!

Privacy Policy