Breadth First Search or BFS for a Graph

The breadth-first search (BFS) algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at the tree's root or graph and searches/visits all nodes at the current depth level before moving on to the nodes at the next depth level. Breadth-first search can be used to solve many problems in graph theory.

Relation between BFS for Graph and Tree traversal:

Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree (See method 2 of this post).

The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- Visited and
- Not visited.

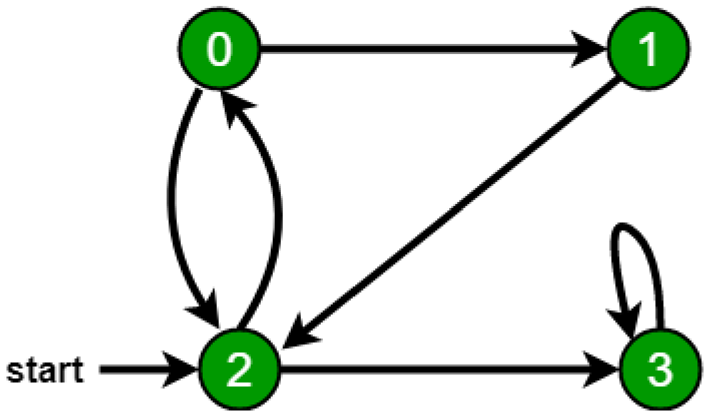
A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a **queue data structure** for traversal.

Algorithm of Breadth-First Search:

- **Step 1:** Consider the graph you want to navigate.
- **Step 2:** Select any vertex in your graph (say **v1**), from which you want to traverse the graph.
- **Step 3:** Utilize the following two data structures for traversing the graph.
- Visited array(size of the graph)
- Queue data structure
- **Step 4:** Add the starting vertex to the visited array, and afterward, you add v1's adjacent vertices to the queue data structure.
- **Step 5:** Now using the FIFO concept, remove the first element from the queue, put it into the visited array, and then add the adjacent vertices of the removed element to the queue.
- **Step 6:** Repeat step 5 until the queue is not empty and no vertex is left to be visited.

Examples:

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.

There can be multiple BFS traversals for a graph. Different BFS traversals for the above graph :
2, 3, 0, 1
2, 0, 3, 1

Implementation of BFS traversal on Graph:

Pseudocode for BFS:

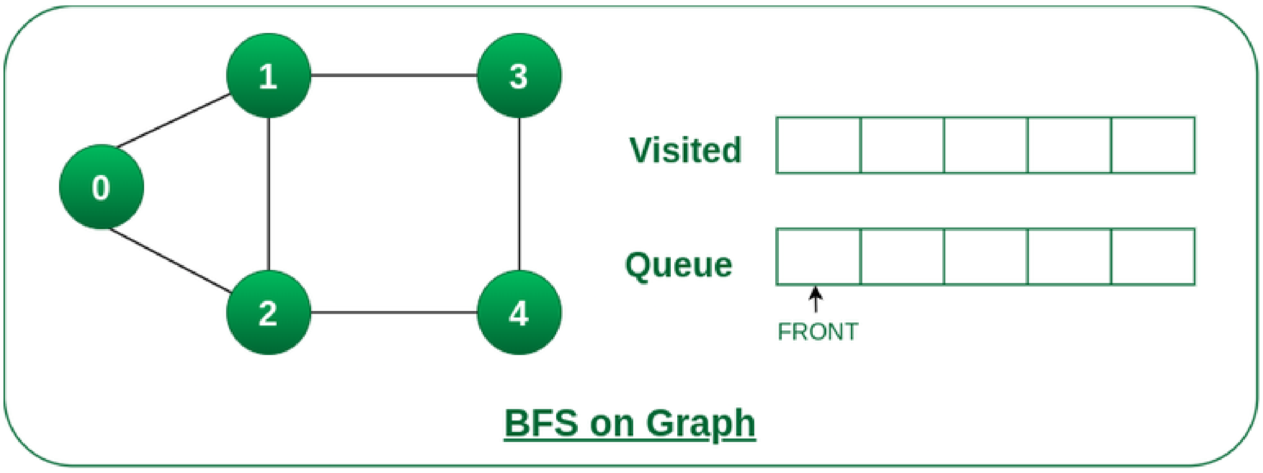
```
// Here, Graph is the graph that we already have and X is the source node
Breadth_First_Search( Graph, X ):
   Let Q be the queue
   Q.enqueue( X )     // Inserting source node X into the queue
   Mark X node as visited.

   While ( Q is not empty )
      Y = Q.dequeue( )     // Removing the front node from the queue

   Process all the neighbors of Y, For all the neighbors Z of Y
   If Z is not visited:
      Q. enqueue( Z )     // Stores Z in Q
      Mark Z as visited
```

Follow the below method to implement BFS traversal.

- Declare a queue and insert the starting vertex.
- Initialize a **visited** array and mark the starting vertex as visited.
- Follow the below process till the queue becomes empty:
  - Remove the first vertex of the queue.
  - Mark that vertex as visited.
  - Insert all the unvisited neighbors of the vertex into the queue.

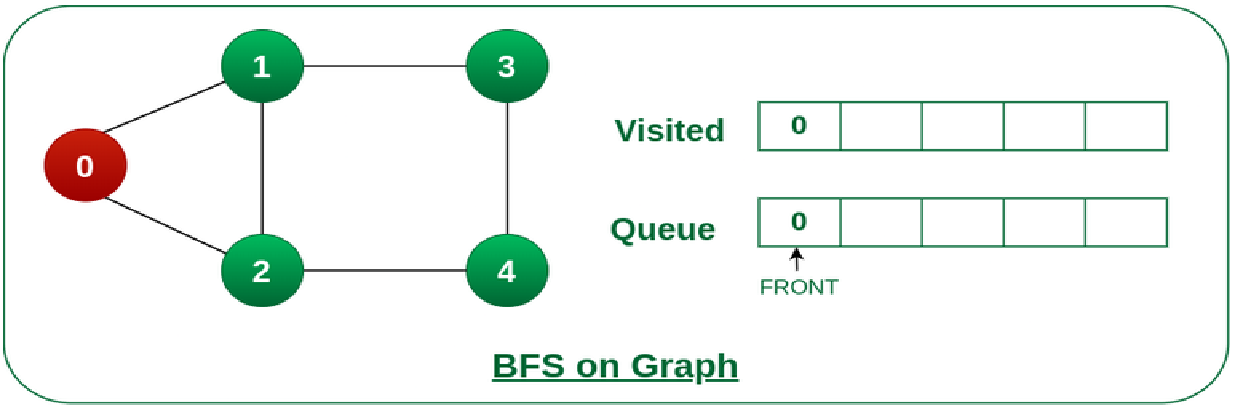
Illustration:

Step1: Initially queue and visited arrays are empty.



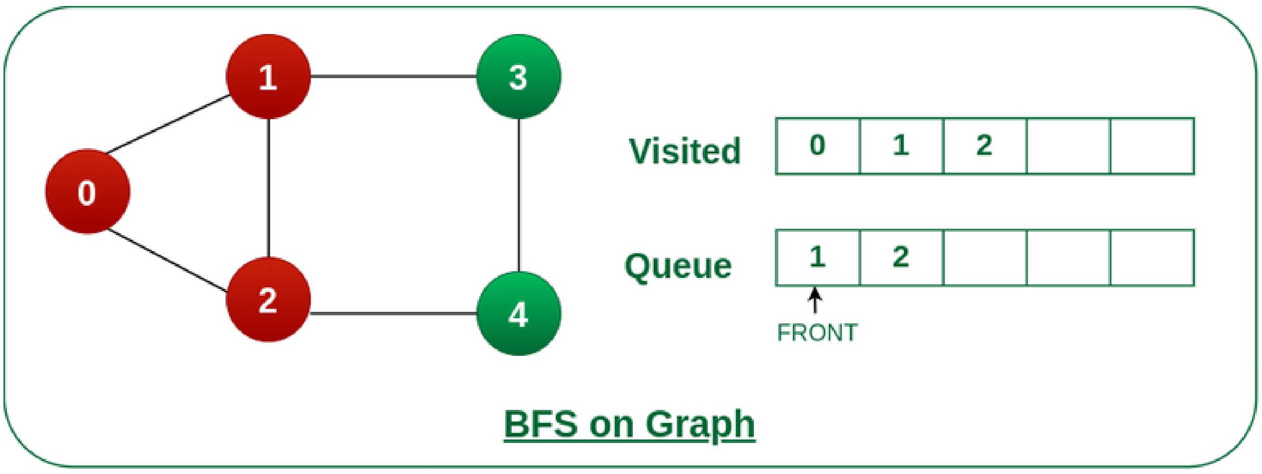
Queue and visited arrays are empty initially.

Step2: Push node 0 into queue and mark it visited.



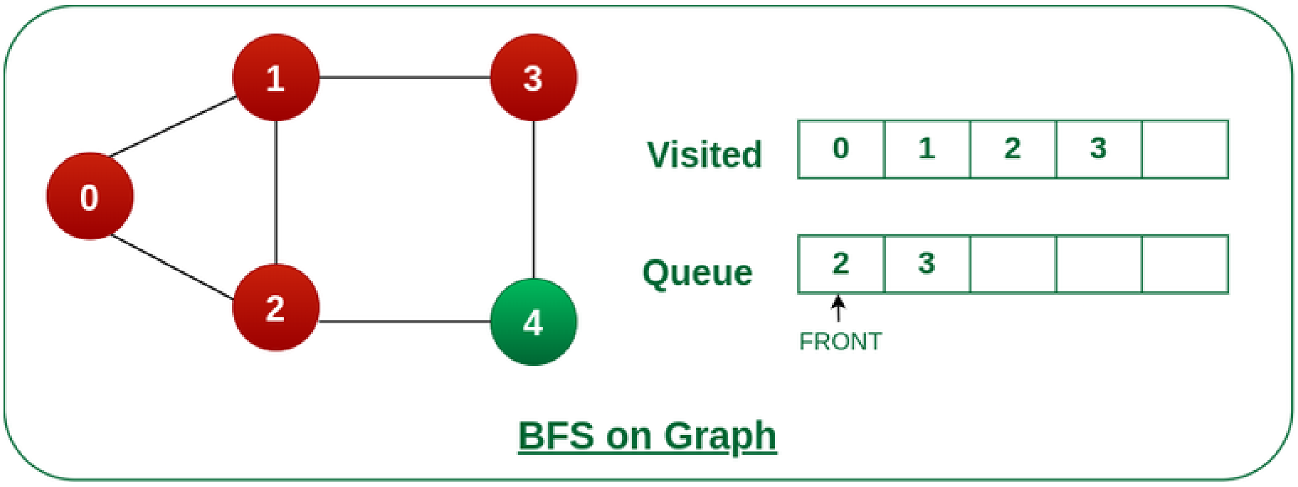
Push node 0 into queue and mark it visited.

Step 3: Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.



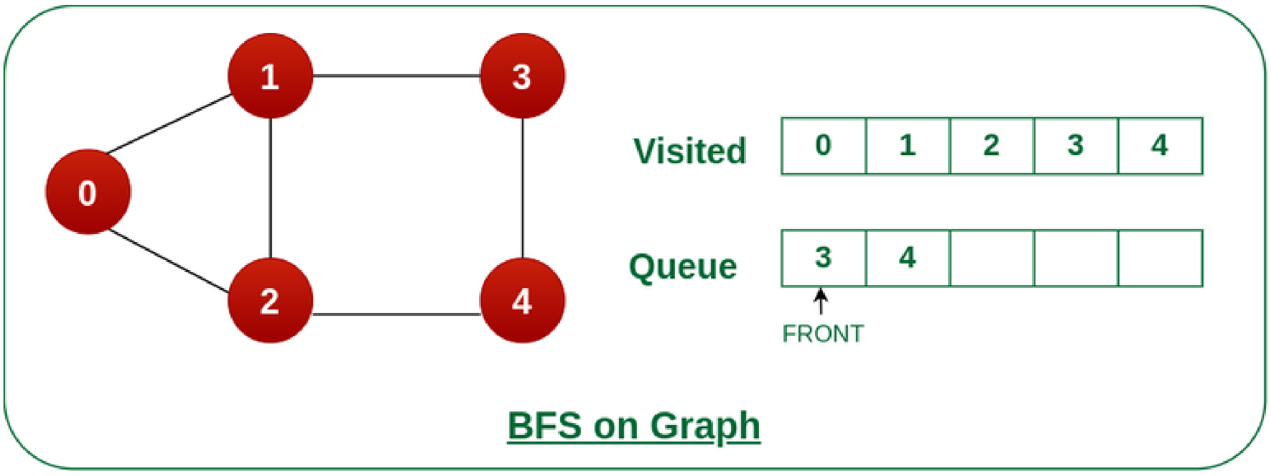
Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

Step 4: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.



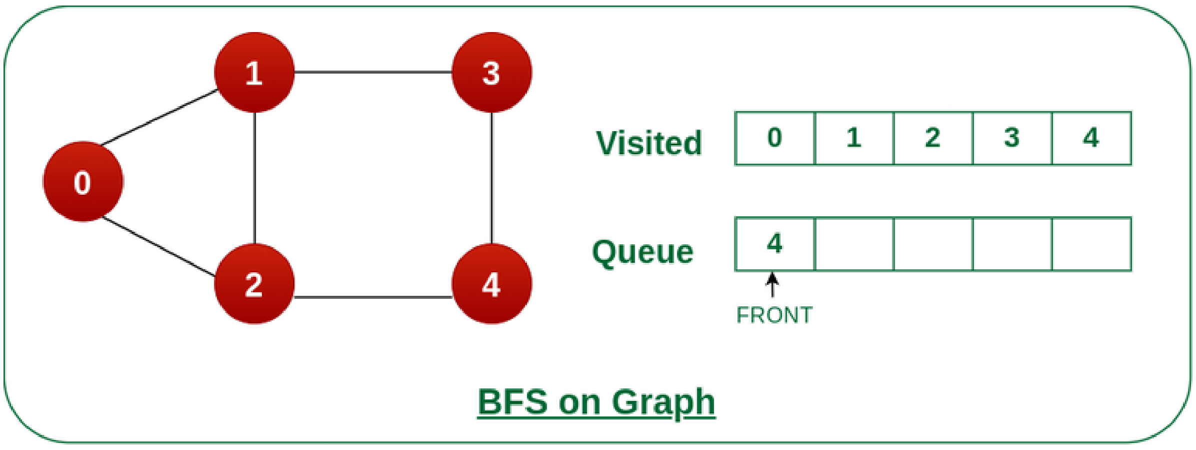
Remove node 1 from the front of queue and visited the unvisited neighbours and push

Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



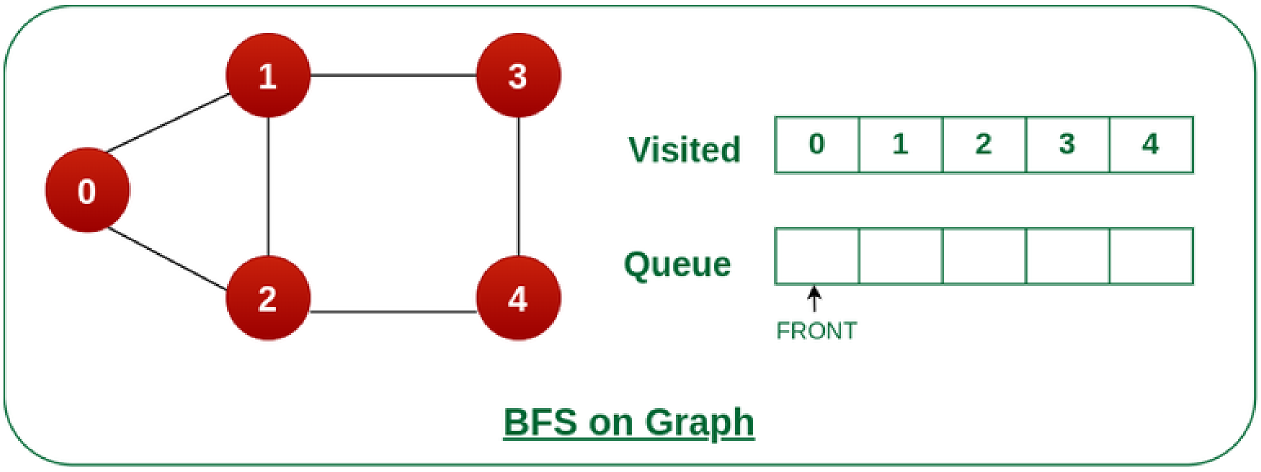
Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

Step 6: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.
As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

Steps 7: Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.
As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.



Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

Now, Queue becomes empty, So, terminate these process of iteration.

The implementation uses an adjacency list representation of graphs. STL's list container stores lists of adjacent nodes and the queue of nodes needed for BFS traversal.

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_VERTICES 50

// This struct represents a directed graph using
// adjacency list representation
typedef struct Graph_t {

    // No. of vertices
    int V;
    bool adj[MAX_VERTICES][MAX_VERTICES];
} Graph;

// Constructor
Graph* Graph_create(int V)
{
    Graph* g = malloc(sizeof(Graph));
    g->V = V;

    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            g->adj[i][j] = false;
        }
    }

    return g;
}

// Destructor
void Graph_destroy(Graph* g) {
    free(g);
```

```
}

// function to add an edge to graph
void Graph_addEdge(Graph* g, int v, int w)
{
    // Add w to v’s list.
    g->adj[v][w] = true;
}

// Prints BFS traversal from a given source s
void Graph_BFS(Graph* g, int s)
{
    // Mark all the vertices as not visited
    bool visited[MAX_VERTICES];
    for (int i = 0; i < g->V; i++) {
        visited[i] = false;
    }

    // Create a queue for BFS
    int queue[MAX_VERTICES];
    int front = 0, rear = 0;

    // Mark the current node as visited and enqueue it
    visited[s] = true;
    queue[rear++] = s;

    while (front != rear) {
        // Dequeue a vertex from queue and print it
        s = queue[front++];
        printf("%d ", 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 (int adjacent = 0; adjacent < g->V;
             adjacent++) {
            if (g->adj[s][adjacent] && !visited[adjacent]) {
                visited[adjacent] = true;
                queue[rear++] = adjacent;
            }
        }
    }
}

// Driver code
int main()
{
    // Create a graph
    Graph* g = Graph_create(4);
    Graph_addEdge(g, 0, 1);
    Graph_addEdge(g, 0, 2);
    Graph_addEdge(g, 1, 2);
    Graph_addEdge(g, 2, 0);
    Graph_addEdge(g, 2, 3);
    Graph_addEdge(g, 3, 3);

    printf("Following is Breadth First Traversal "
           "(starting from vertex 2) \n");
    Graph_BFS(g, 2);
    Graph_destroy(g);

    return 0;
}
```
Output
```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

Time Complexity: O(V+E), where V is the number of nodes and E is the number of edges.
Auxiliary Space: O(V)