```
// C++ implementation of the approach
//#include <bits/stdc++.h>
#include <iostream>
#define MAX_VERTICES 20
#define SIZE 10


using namespace std;


// Class for queue
class Queue
{
    int arr[MAX_VERTICES];        // array to store Queue elements
    int capacity;   // maximum capacity of the Queue
    int front;      // front points to front element in the Queue (if
any)
    int rear;       // rear points to last element in the Queue
    int count;      // current size of the Queue

public:
    Queue(int size = SIZE);     // constructor
//    ~Queue();                   // destructor

    int dequeue();
    void enqueue(int x);
    int peek();
    int size();
    bool isEmpty();
    bool isFull();
};

// Constructor to initialize Queue
Queue::Queue(int size)
{
    for(int i=0;i<size;i++)
        arr[i] = 0;
    front = 0;
    rear = -1;
    count=0;
    capacity = MAX_VERTICES;
}

// Utility function to remove front element from the Queue
int Queue::dequeue()
{
    // check for Queue underflow
    if (isEmpty())
    {
        cout << "Queue UnderFlow\n";
        return -1; // -1 means Queue was empty
    }

    int removed = arr[front];

    front = (front + 1) % capacity;
```

```
    count--;
    return removed;
}

// Utility function to add an item to the Queue
void Queue::enqueue(int item)
{
    // check for Queue overflow
    if (isFull())
    {
        cout << "Queue OverFlow\n";
        return;
    }

//    cout << "Inserting " << item << '\n';

    rear = (rear + 1) % capacity;
    arr[rear] = item;
    count++;
}

// Utility function to return front element in the Queue
int Queue::peek()
{
    if (isEmpty())
    {
        cout << "Queue UnderFlow\n";
        return -1; // -1 means Queue is empty
    }
    return arr[front];
}


// Utility function to check if the Queue is empty or not
bool Queue::isEmpty()
{
    return (count == 0);
}

// Utility function to check if the Queue is full or not
bool Queue::isFull()
{
    return (count == capacity);
}






class Graph {

        // Number of vertex
        int v;

        // Number of edges
```

```
        int e;

        // Adjacency matrix
        int adj[MAX_VERTICES][MAX_VERTICES]; // assume maximum number
of vertices=20

public:
        // To create the initial adjacency matrix
        Graph(int v, int e);

        // Function to insert a new edge
        void addEdge(int start_v, int end_v);

        // Function to display the BFS traversal
        void BFS(int start);
};

// Function to fill the empty adjacency matrix
Graph::Graph(int v, int e)
{
        this->v = v;
        this->e = e;
        for (int row = 0; row < v; row++) {
                for (int column = 0; column < v; column++) {
                        adj[row][column] = 0;
                }
        }
}

// Function to add an edge to the graph
void Graph::addEdge(int start_v, int end_v)
{

        // Considering a bidirectional edge (undirected graph)
        adj[start_v][end_v] = 1;
        adj[end_v][start_v] = 1;
}

// Function to perform BFS on the graph
void Graph::BFS(int start)
{
        // Visited vector to so that
        // a vertex is not visited more than once
        // Initializing the vector to false as no
        // vertex is visited at the beginning
        bool visited[MAX_VERTICES] = {false};
//      vector<bool> visited(v, false);
    Queue q;
//      vector<int> q;
        q.enqueue(start);

        // Set source as visited
        visited[start] = true;

        int u;
        while (!q.isEmpty()) {
                u = q.dequeue();
```

```
                // Print the current node
                cout << u << " ";
        // For every adjacent vertex to the current vertex
                for (int v1 = 0; v1 < v; v1++) {
                        if (adj[u][v1] == 1 && (!visited[v1])) // if
node v1 is adjacent to u and not visited
            {

                                // Push the adjacent node to the Queue
                                q.enqueue(v1);

                                // Set v1 as visited
                                visited[v1] = true;
                        }
                }
        }
}

// Driver code
int main()
{
        int v = 8, e = 4;

        // Create the graph
        Graph G(v, e);
        G.addEdge(1, 2);
        G.addEdge(1, 5);
        G.addEdge(2, 6);
        G.addEdge(5, 6);
        G.addEdge(6, 4);
        G.addEdge(5, 3);
//      G.addEdge(5, 6);


        G.BFS(1);
}
```