DSAL - EXPERIMENT *6* - C-13

Name → Pratik Pingale
Roll No → *19 CO056*
Batch → C

Experiment → *Represent a given graph using adjacency matrix/list to perform DFS and using adjacency* list to perform BFS. Use the map of the area around the college as the graph. Identify the prominent land marks as nodes *and* perform DFS *and* BFS on that.

---------- CODE ----------

```cpp
#include <iostream>
#include <vector>
#include <list>
using namespace std;

class Graph {
  // Number of vertex
  int v;
  // Pointer to an array containing adjacency lists
  list<int> *adjL;

  // Adjacency matrix
  int **adjM;
  // Visited vector to so that a vertex is not visited more than once
  vector<bool> visited;
public:
  // Constructor create the initial /list
  Graph(int);
  // Function to insert a new edge
  void addEdge(int, int);
  // Function to display the DFS traversal on adjacency matrix
  void DFS(int);
  // Function to display the BFS traversal on adjacency list
  void BFS(int);
};


// Function to fill the empty adjacency matrix & initialize adjacency list
Graph::Graph(int v) {
  this->v = v;
  // Adjacency lists
  adjL = new list<int>[v];

  // Adjacency matrix
  adjM = new int*[v];
  // A visited array of initially false for all vertices
  visited.assign(v, false);
  for (int row = 0; row < v; row++) {
    adjM[row] = new int[v];
    for (int column = 0; column < v; column++) {
      adjM[row][column] = 0;
    }
  }
}


// Function to add an edge to the graph
void Graph::addEdge(int x, int y) {
  // Add y to x's adjacency list.
  adjL[x].push_back(y);
  // Add x to y's adjacency list.
  adjL[y].push_back(x);

  // Considering a bidirectional edge to adjacency matrix
  adjM[x][y] = 1;
  adjM[y][x] = 1;
}

// Function to perform DFS on the graph
void Graph::DFS(int start) {
  // Print the first node
  cout << start << " ";
  // Set current node as visited
  visited[start] = true;

  // For every node of the graph
  for (int i = 0; i < v; i++) {
    // If some node is adjacent to the current node
    // and it has not already been visited
    if (adjM[start][i] == 1 && (!visited[i])) {
      DFS(i);
    }
  }
}

void Graph::BFS(int start) {
  // A visited array of initially false for all vertices
  visited.assign(v, false);

  // Create a queue for BFS
  list<int> queue;
  // Mark the current node as visited and enqueue it
  visited[start] = true;
  queue.push_back(start);

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

    // Get all adjacent vertices of the dequeued
    // vertex start. If a adjacent has not been visited,
    // then mark it visited and enqueue it
    for (int i : adjL[start]) {
      if (!visited[i]) {
        visited[i] = true;
        queue.push_back(i);
      }
    }
  }
}

int main() {
  int v = 8;

  Graph G(v);

  // Graph edges
  int edges[][2] = {{0, 1}, {0, 6}, {0, 5}, {1, 2}, {1, 6}, {2, 3}, {2, 4}, {2, 6}, {2, 7}, {3, 4},
{3, 7}, {4, 5}, {4, 6}, {4, 7}, {5, 6}};

  for (auto edge : edges) {
    G.addEdge(edge[0], edge[1]);
  }

  cout << "\nOperation on following Graph →\n\n\n"
  "               (1)  @@  @  @  @  @ @  @ @@@  (2)\n"
  "                   @@@               @  @ @   @\n"
  "                 @  @              @     @   @     @\n"
  "              @     @            @       @     @        @\n"
  "           @        @         @          @       @          @\n"
  "  (0)    @          @      @             @          @            @\n"
  "      @              @   @                @             @              @\n"
  "  @@@ @  @  @  @  @ @@  (6)              @          (7) @@               @@ (3)\n"
  "  @@@                @@                  @              @@@  @  @  @  @  @@@\n"
  "      @              @   @                @             @              @\n"
  "         @           @      @             @           @              @\n"
  "           @         @        @           @         @          @\n"
  "             @       @           @        @       @        @\n"
  "                @    @             @      @     @      @\n"
  "                  @@@                @   @  @   @\n"
  "              (5)   @@  @  @  @   @  @  @@@@  (4)\n";

  // Perform DFS
  cout << "\n\n  Depth First Traversal (starting from vertex 2) : ";
  G.DFS(2);

  // Perform BFS
  cout << "\n\nBreadth First Traversal (starting from vertex 2) : ";
  G.BFS(2);
  return 0;
}
```

---------- OUTPUT ----------

```
Operation on following Graph ->


               (1)  @@  @  @  @  @ @  @ @@@  (2)
                   @@@               @  @ @   @
                 @  @              @     @   @     @
              @     @            @       @     @        @
           @        @         @          @       @          @
  (0)    @          @      @             @          @            @
      @              @   @                @             @              @
  @@@ @  @  @  @  @ @@  (6)              @          (7) @@               @@ (3)
  @@@                @@                  @              @@@  @  @  @  @  @@@
      @              @   @                @             @              @
         @           @      @             @           @              @
           @         @        @           @         @          @
             @       @           @        @       @        @
                @    @             @      @     @      @
                  @@@                @   @  @   @
              (5)   @@  @  @  @   @  @  @@@@  (4)


  Depth First Traversal (starting from vertex 2) : 2 1 0 5 4 3 7 6
Breadth First Traversal (starting from vertex 2) : 2 1 3 4 6 7 0 5
```