```
#include <iostream>
#include <fstream>

#include <ctime>
#include <ratio>
#include <chrono>
#include <time.h>
#include <stdlib.h>

#include <string>
#include <cstring> // for string tokenizer and c-style string processing

using namespace std;

// implementing the dynamic List ADT using Linked List

class Node{

        private:
                int data;
                Node* nextNodePtr;

        public:
                Node(){}

                void setData(int d){
                        data = d;
                }

                int getData(){
                        return data;
                }

                void setNextNodePtr(Node* nodePtr){
                                nextNodePtr = nodePtr;
                }

                Node* getNextNodePtr(){
                        return nextNodePtr;
                }

};

class List{

        private:
                Node *headPtr;

        public:
```

```
List(){
        headPtr = new Node();
        headPtr->setNextNodePtr(0);
}

Node* getHeadPtr(){
        return headPtr;
}

bool isEmpty(){

        if (headPtr->getNextNodePtr() == 0)
                return true;

        return false;
}


void insert(int data){

        Node* currentNodePtr = headPtr->getNextNodePtr();
        Node* prevNodePtr = headPtr;

        while (currentNodePtr != 0){
                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr->getNextNodePtr();
        }

        Node* newNodePtr = new Node();
        newNodePtr->setData(data);
        newNodePtr->setNextNodePtr(0);
        prevNodePtr->setNextNodePtr(newNodePtr);


}

void insertAtIndex(int insertIndex, int data){

        Node* currentNodePtr = headPtr->getNextNodePtr();
        Node* prevNodePtr = headPtr;

        int index = 0;

        while (currentNodePtr != 0){

                if (index == insertIndex)
                        break;

                prevNodePtr = currentNodePtr;
```

```
                GraphBFS_AssignedQuiz
                currentNodePtr = currentNodePtr->getNextNodePtr();
                index++;
        }

        Node* newNodePtr = new Node();
        newNodePtr->setData(data);
        newNodePtr->setNextNodePtr(currentNodePtr);
        prevNodePtr->setNextNodePtr(newNodePtr);

}


int read(int readIndex){

        Node* currentNodePtr = headPtr->getNextNodePtr();
        Node* prevNodePtr = headPtr;
        int index = 0;

        while (currentNodePtr != 0){

                if (index == readIndex)
                        return currentNodePtr->getData();

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr->getNextNodePtr();

                index++;

        }

        return -1; // an invalid value indicating
                                // index is out of range

}

void modifyElement(int modifyIndex, int data){

        Node* currentNodePtr = headPtr->getNextNodePtr();
        Node* prevNodePtr = headPtr;
        int index = 0;

        while (currentNodePtr != 0){

                if (index == modifyIndex){
                        currentNodePtr->setData(data);
                        return;
                }

                prevNodePtr = currentNodePtr;
```

```
                currentNodePtr = currentNodePtr->getNextNodePtr();

                index++;
            }


        }


        void deleteElementAtIndex(int deleteIndex){

            Node* currentNodePtr = headPtr->getNextNodePtr();
            Node* prevNodePtr = headPtr;
            Node* nextNodePtr = headPtr;
            int index = 0;

            while (currentNodePtr != 0){

                if (index == deleteIndex){
                    nextNodePtr =
currentNodePtr->getNextNodePtr();
                    break;
                }

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr->getNextNodePtr();


                index++;
            }

            prevNodePtr->setNextNodePtr(nextNodePtr);

        }


        void deleteElement(int deleteData){

            Node* currentNodePtr = headPtr->getNextNodePtr();
            Node* prevNodePtr = headPtr;
            Node* nextNodePtr = headPtr;

            while (currentNodePtr != 0){

                if (currentNodePtr->getData() == deleteData){
                    nextNodePtr =
currentNodePtr->getNextNodePtr();
                    break;
                }
```

```
                        prevNodePtr = currentNodePtr;
                        currentNodePtr = currentNodePtr->getNextNodePtr();


                }

                prevNodePtr->setNextNodePtr(nextNodePtr);

        }


        void IterativePrint(){

                Node* currentNodePtr = headPtr->getNextNodePtr();

                while (currentNodePtr != 0){
                        cout << currentNodePtr->getData() << " ";
                        currentNodePtr = currentNodePtr->getNextNodePtr();
                }

                cout << endl;

        }


        int countList(){

                Node* currentNodePtr = headPtr->getNextNodePtr();
                int countElements = 0;

                while (currentNodePtr != 0){
                        currentNodePtr = currentNodePtr->getNextNodePtr();
                        countElements++;
                }

                return countElements;

        }




};


class Queue{

        private:
```

```
        int *array;
        int maxSize; // useful to decide if resizing (doubling the array
size) is needed
        int endOfQueue; // same as endOfArray

    public:
        Queue(int size){
                array = new int[size];
                maxSize = size;
                endOfQueue = -1;
        }

        bool isEmpty(){

                if (endOfQueue == -1)
                        return true;

                return false;
        }

        void resize(int s){

                int *tempArray = array;

                array = new int[s];

                for (int index = 0; index < min(s, endOfQueue+1); index++){
                        array[index] = tempArray[index];
                }

                maxSize = s;

        }


        void enqueue(int data){ // same as insert 'at the end'

                if (endOfQueue == maxSize-1)
                        resize(2*maxSize);

                array[++endOfQueue] = data;

        }


        int peek(){

                if (endOfQueue >= 0)
                        return array[0];
```

```
                else
                        return -1000000;// an invalid value indicating
                                                        // queue is empty

        }


        int dequeue(){

                if (endOfQueue >= 0){
                        int returnVal = array[0];

                        for (int index = 0; index < endOfQueue; index++)
                                array[index] = array[index+1];

                        endOfQueue--;
                        // the endOfQueue is decreased by one

                        return returnVal;
                }
                else
                        return -1000000; // an invalid value indicating
                                                        // queue is empty
        }




};


class Graph{

        private:
                int numNodes;
                List* adjacencyList;
                int* levelNumbers;

        public:

                Graph(int n){
                        numNodes = n;
                        adjacencyList = new List[numNodes];
                        levelNumbers = new int[numNodes];
                }


                void addEdge(int u, int v){
```

```
                adjacencyList[u].insert(v);
                adjacencyList[v].insert(u);

        }


        List getNeighborList(int id){
                return adjacencyList[id];
        }




        int getLevelNumber(int id){
                return levelNumbers[id];
        }



        void RunBFS(int startNodeID){

                // The BFS function should be modified to determine two
things
                // (1) The level number of the vertices; the level number
for the startNodeID is 0.
                // (2) Whether the graph is bipartite or not and accordingly
a boolean (bool) should be returned.



                bool* visitedNodes = new bool[numNodes];

                for (int id = 0; id < numNodes; id++){
                        levelNumbers[id] = -1;
                        visitedNodes[id] = false;
                }

                levelNumbers[startNodeID] = 0;
                visitedNodes[startNodeID] = true;

                Queue FIFOQueue(1);
                FIFOQueue.enqueue(startNodeID);



                while (!FIFOQueue.isEmpty()){

                        int firstNodeID = FIFOQueue.dequeue();
```

```
                                int neighborListSize =
adjacencyList[firstNodeID].countList();

                                for (int index = 0; index < neighborListSize;
index++){

                                        int neighborID =
adjacencyList[firstNodeID].read(index);

                                        if (!visitedNodes[neighborID]){
                                                visitedNodes[neighborID] = true;
                                                FIFOQueue.enqueue(neighborID);


                                        }




                                }

                        }


                }

};


int main(){

        string graphEdgesFilename;
        cout << "Enter the file name for the edges of the graph: ";
        cin >> graphEdgesFilename;

        int numNodes;
        cout << "Enter number of nodes: ";
        cin >> numNodes;



        Graph graph(numNodes);

        ifstream graphEdgeFileReader(graphEdgesFilename);
```

```
        if (!graphEdgeFileReader){
                cout << "File cannot be opened!! ";
                return 0;
        }

        int numCharsPerLine = 25;

        char *line = new char[numCharsPerLine];
        // '25' is the maximum number of characters per line

        graphEdgeFileReader.getline(line, numCharsPerLine, '\n');
        // '\n' is the delimiting character to stop reading the line

        while (graphEdgeFileReader){

                char* cptr = strtok(line, " ");

                string uNodeToken(cptr);
                int uNodeID = stoi(uNodeToken);

                cptr = strtok(NULL, " ");

                string vNodeToken(cptr);
                int vNodeID = stoi(vNodeToken);

                graph.addEdge(uNodeID, vNodeID);


                graphEdgeFileReader.getline(line, numCharsPerLine, '\n');

        }


        // Add code here to call the BFS function with the startNodeID as 0
        // Make the BFS function to return a boolean (bool) indicating whether
        // the graph is bipartite or not and print the result.

        // Also, write a for loop to print the level number for each vertex


return 0;
}
```