Experiment C13

```
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

// Adjacency List - Adding O(1), Lookup O(N), Space O(N^2) but usually better.
// Each vector position represents a node - the vector inside that position represents that
node's friends.
vector< vector<int> > FormAdjList()
    {
    // Our adjacency list.
    vector< vector<int> > adjList;

    // We have 10 vertices, so initialize 10 rows.
    const int n = 9;

    for(int i = 0; i < n; i++)
        {
        // Create a vector to represent a row, and add it to the adjList.
        vector<int> row;
        adjList.push_back(row);
        }

    // Now "adjList[0]" has a vector<int> in it that represents the friends of vertex 1.
    // (Remember, we use 0-based indexing. 0 is the first number in our vector, not 1.

    // Now let's add our actual edges into the adjacency list.
    // See the picture here:
https://www.srcmake.com/uploads/5/3/9/0/5390645/adjl_4_orig.png

    adjList[0].push_back(2);
    adjList[0].push_back(4);
    adjList[0].push_back(6);

    adjList[1].push_back(4);
    adjList[1].push_back(7);

    adjList[2].push_back(0);
    adjList[2].push_back(5);

    adjList[3].push_back(4);
    adjList[3].push_back(5);

    adjList[4].push_back(1);
    adjList[4].push_back(3);
    adjList[4].push_back(0);

    adjList[5].push_back(2);
    adjList[5].push_back(3);
    adjList[5].push_back(8);

    adjList[6].push_back(0);

    adjList[7].push_back(1);

    adjList[8].push_back(5);

    // Our graph is now represented as an adjacency list.
    return adjList;
    }

// Adjacency Matrix - Adding O(N), Lookup O(1), Space O(N^2)
vector< vector<int> > FormAdjMatrix()
    {
    // We could use an array for the adjMatrix if we knew the size, but it's safer to use a
vector.
    vector< vector<int> > adjMatrix;

    // Initialize the adjMatrix so that all vertices can visit themselves.
    // (Basically, make an identity matrix.)
    const int n = 9;

    for(int i = 0; i < n; i++)
        {
        // Initialize the row.
        vector<int> row;
        adjMatrix.push_back(row);


        for(int j = 0; j < n; j++)
            {
            int value = 0;

            if(i == j)
                { value = 1; }

            adjMatrix[i].push_back(value);
            }
        }


    adjMatrix[0][2] = 1;
```

```
   adjMatrix[2][0] = 1;

   adjMatrix[0][4] = 1;
   adjMatrix[4][0] = 1;

   adjMatrix[0][6] = 1;
   adjMatrix[6][0] = 1;

   adjMatrix[1][4] = 1;
   adjMatrix[4][1] = 1;

   adjMatrix[1][7] = 1;
   adjMatrix[7][1] = 1;

   adjMatrix[2][5] = 1;
   adjMatrix[5][2] = 1;

   adjMatrix[3][4] = 1;
   adjMatrix[4][3] = 1;

   adjMatrix[3][5] = 1;
   adjMatrix[5][3] = 1;

   adjMatrix[5][8] = 1;
   adjMatrix[8][5] = 1;

   // Our adjacency matrix is complete.
   return adjMatrix;
   }

// Given an Adjacency List, do a BFS on vertex "start"
void AdjListBFS(vector< vector<int> > adjList, int start)
   {
   cout << "\nDoing a BFS on an adjacency list.\n";

   int n = adjList.size();
   // Create a "visited" array (true or false) to keep track of if we visited a vertex.
   bool visited[n] = { false };

   // Create a queue for the nodes we visit.
   queue<int> q;

   // Add the starting vertex to the queue and mark it as visited.
   q.push(start);
   visited[start] = true;

   // While the queue is not empty..
   while(q.empty() == false)
```

```
        {
        int vertex = q.front();
        q.pop();

        // Doing +1 in the cout because our graph is 1-based indexing, but our code is 0-based.
        cout << vertex+1 << " ";

        // Loop through all of it's friends.
        for(int i = 0; i < adjList[vertex].size(); i++)
            {
            // If the friend hasn't been visited yet, add it to the queue and mark it as visited
            int neighbor = adjList[vertex][i];

            if(visited[neighbor] == false)
                {
                q.push(neighbor);
                visited[neighbor] = true;
                }
            }
        }
    cout << endl << endl;
    return;
    }

void AdjListDFS(vector< vector<int> > &adjList, int &vertex, vector<bool> &visited)
    {
    // Mark the vertex as visited.
    visited[vertex] = true;

    // Outputting vertex+1 because that's the way our graph picture looks.
    cout << vertex+1 << " ";

    // Look at this vertex's neighbors.
    for(int i = 0; i < adjList[vertex].size(); i++)
        {
        int neighbor = adjList[vertex][i];
        // Recursively call DFS on the neighbor, if it wasn't visited.
        if(visited[neighbor] == false)
            {
            AdjListDFS(adjList, neighbor, visited);
            }
        }
    }
// Given an Adjacency Matrix, do a BFS on vertex "start"
void AdjListDFSInitialize(vector< vector<int> > &adjList, int start)
    {
    cout << "\nDoing a DFS on an adjacency list.\n";
    int n = adjList.size();
    // Create a "visited" array (true or false) to keep track of if we visited a vertex.
    vector<bool> visited;

    for(int i = 0; i < n; i++)
        {
        visited.push_back(false);
        }

    AdjListDFS(adjList, start, visited);

    cout << endl << endl;
    return;
    }

int main()
    {
    cout << "Program started.\n";

    // Get the adjacency list/matrix.
    vector< vector<int> > adjList = FormAdjList();


    // Call BFS on Vertex 5. (Labeled as 4 in our 0-based-indexing.)
    AdjListBFS(adjList, 4);
    AdjListDFSInitialize(adjList, 4);

    cout << "Program ended.\n";

    return 0;
    }
```