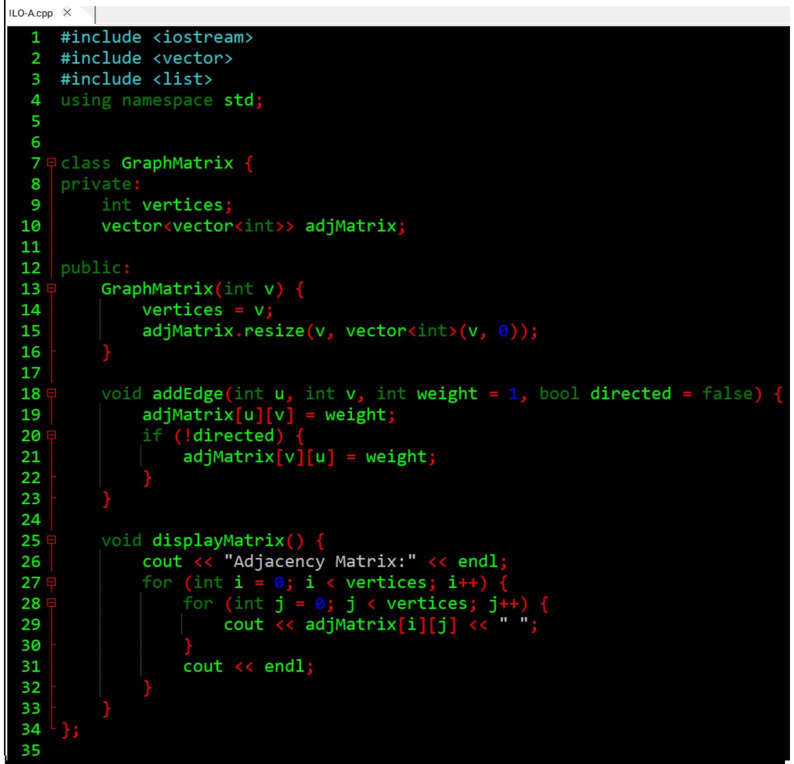
Activity No. <10>	
GRAPHS	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed: 9/30/25**
Section: CPE21S4	**Date Submitted: 9/30/25**
Name(s): ALCANTARA, JASON P.	**Instructor:** Engr. Jimlord Quejado

A. Output(s) and Observation(s):

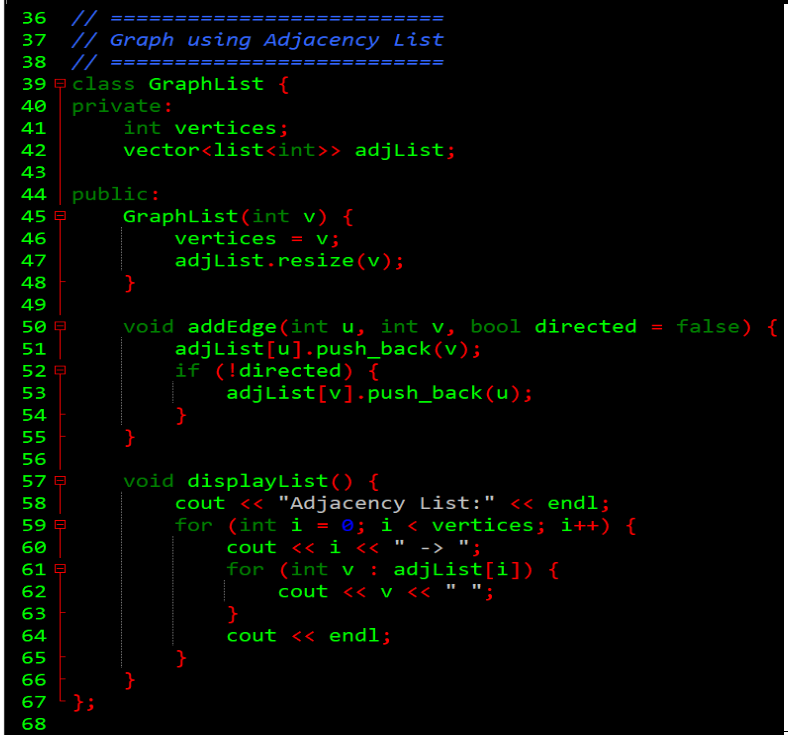
ILO_A

Main CPP:

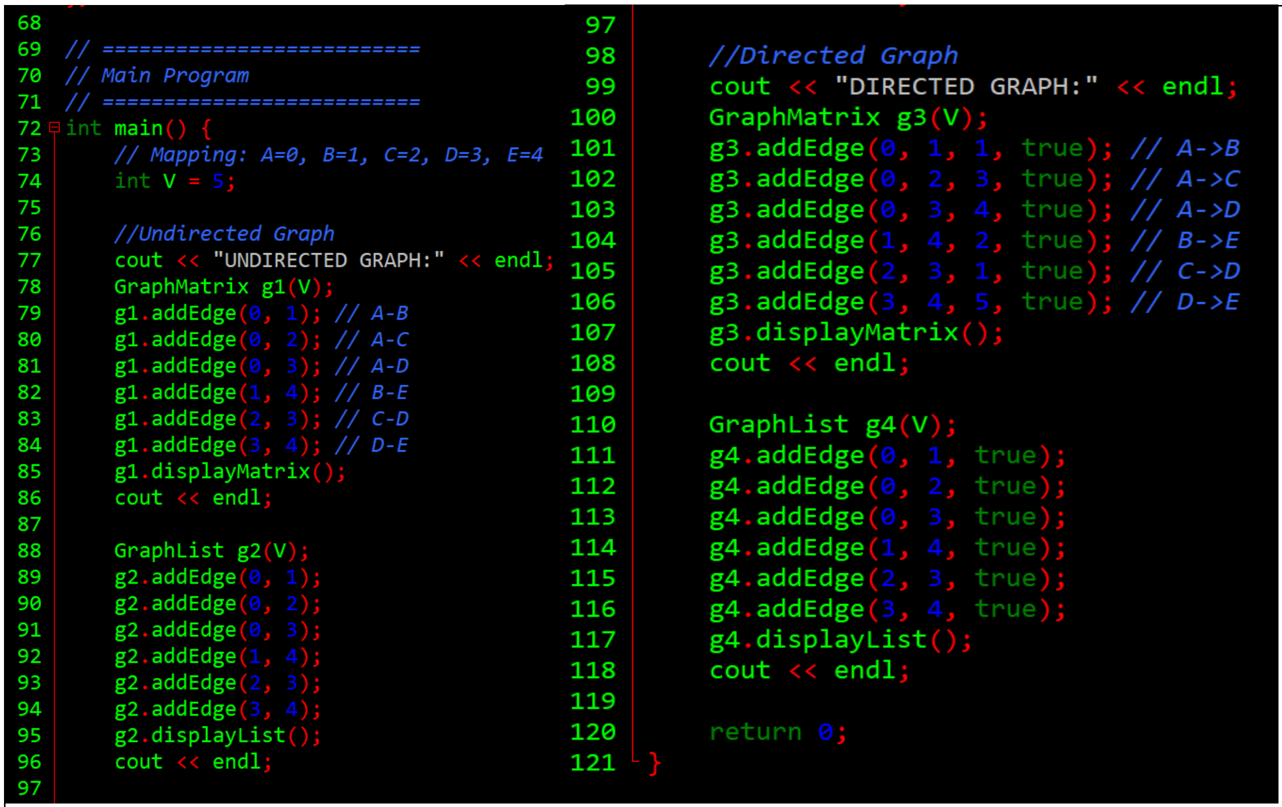


ILO-A.cpp

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


class GraphMatrix {
private:
    int vertices;
    vector<vector<int>> adjMatrix;

public:
    GraphMatrix(int v) {
        vertices = v;
        adjMatrix.resize(v, vector<int>(v, 0));
    }

    void addEdge(int u, int v, int weight = 1, bool directed = false) {
        adjMatrix[u][v] = weight;
        if (!directed) {
            adjMatrix[v][u] = weight;
        }
    }

    void displayMatrix() {
        cout << "Adjacency Matrix:" << endl;
        for (int i = 0; i < vertices; i++) {
            for (int j = 0; j < vertices; j++) {
                cout << adjMatrix[i][j] << " ";
            }
            cout << endl;
        }
    }
};

```

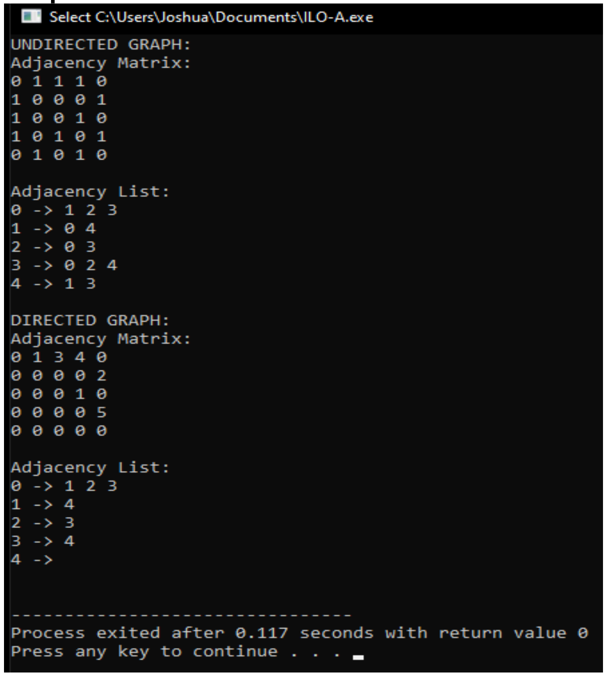


```cpp
// ==========================
// Graph using Adjacency List
// ==========================
class GraphList {
private:
    int vertices;
    vector<list<int>> adjList;

public:
    GraphList(int v) {
        vertices = v;
        adjList.resize(v);
    }

    void addEdge(int u, int v, bool directed = false) {
        adjList[u].push_back(v);
        if (!directed) {
            adjList[v].push_back(u);
        }
    }

    void displayList() {
        cout << "Adjacency List:" << endl;
        for (int i = 0; i < vertices; i++) {
            cout << i << " -> ";
            for (int v : adjList[i]) {
                cout << v << " ";
            }
            cout << endl;
        }
    }
};

```



```cpp

// ==========================
// Main Program
// ==========================
int main() {
    // Mapping: A=0, B=1, C=2, D=3, E=4
    int V = 5;

    //Undirected Graph
    cout << "UNDIRECTED GRAPH:" << endl;
    GraphMatrix g1(V);
    g1.addEdge(0, 1); // A-B
    g1.addEdge(0, 2); // A-C
    g1.addEdge(0, 3); // A-D
    g1.addEdge(1, 4); // B-E
    g1.addEdge(2, 3); // C-D
    g1.addEdge(3, 4); // D-E
    g1.displayMatrix();
    cout << endl;

    GraphList g2(V);
    g2.addEdge(0, 1);
    g2.addEdge(0, 2);
    g2.addEdge(0, 3);
    g2.addEdge(1, 4);
    g2.addEdge(2, 3);
    g2.addEdge(3, 4);
    g2.displayList();
    cout << endl;

    //Directed Graph
    cout << "DIRECTED GRAPH:" << endl;
    GraphMatrix g3(V);
    g3.addEdge(0, 1, 1, true); // A->B
    g3.addEdge(0, 2, 3, true); // A->C
    g3.addEdge(0, 3, 4, true); // A->D
    g3.addEdge(1, 4, 2, true); // B->E
    g3.addEdge(2, 3, 1, true); // C->D
    g3.addEdge(3, 4, 5, true); // D->E
    g3.displayMatrix();
    cout << endl;

    GraphList g4(V);
    g4.addEdge(0, 1, true);
    g4.addEdge(0, 2, true);
    g4.addEdge(0, 3, true);
    g4.addEdge(1, 4, true);
    g4.addEdge(2, 3, true);
    g4.addEdge(3, 4, true);
    g4.displayList();
    cout << endl;

    return 0;
}
```

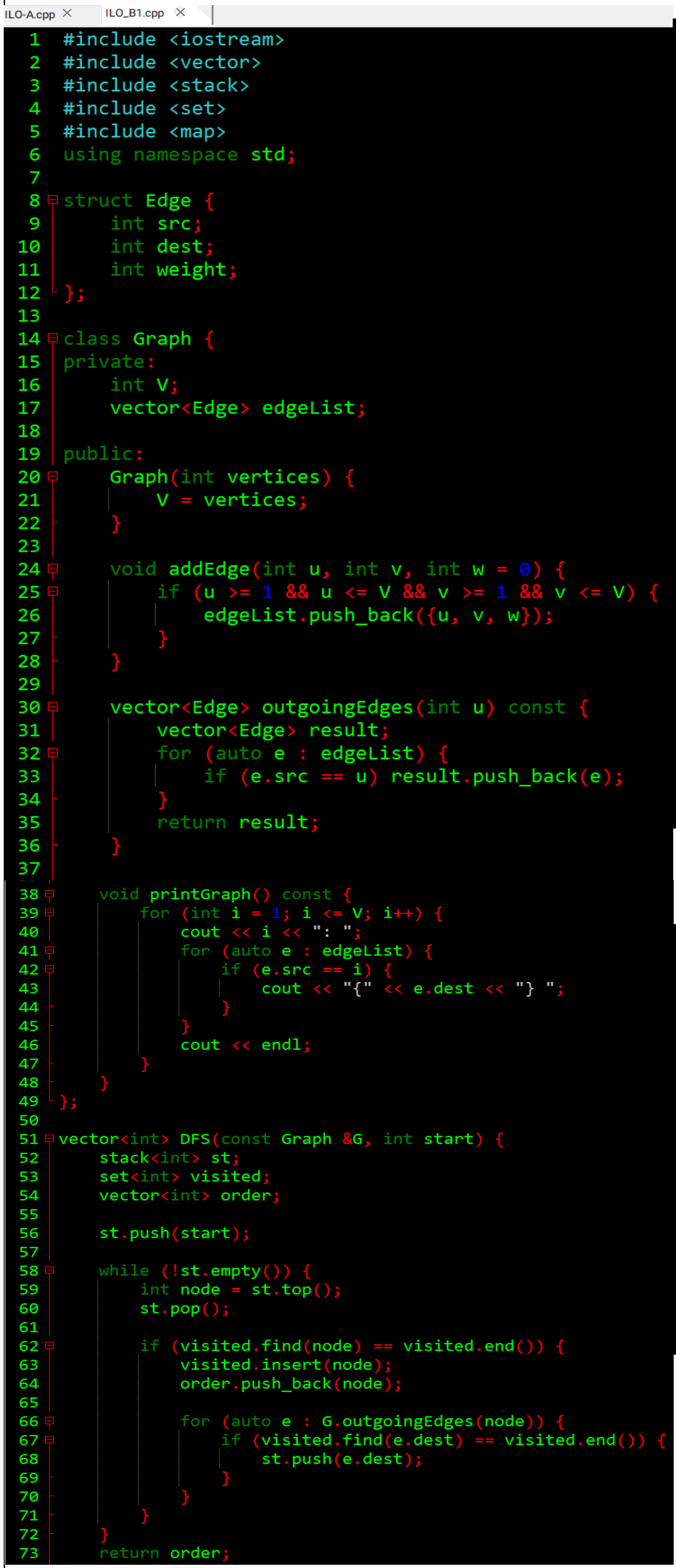
Output:



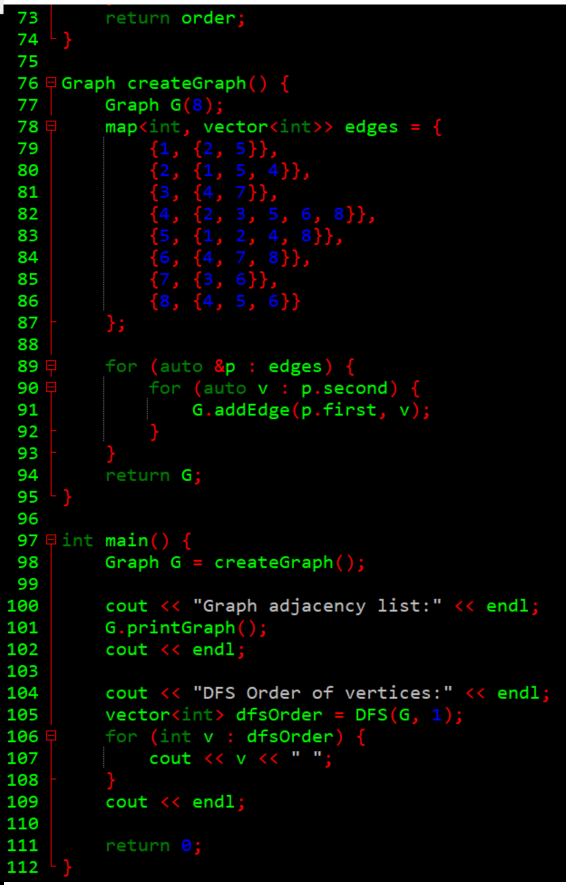
```
Select C:\Users\Joshua\Documents\ILO-A.exe
UNDIRECTED GRAPH:
Adjacency Matrix:
0 1 1 1 0
1 0 0 0 1
1 0 0 1 0
1 0 1 0 1
0 1 0 1 0

Adjacency List:
0 -> 1 2 3
1 -> 0 4
2 -> 0 3
3 -> 0 2 4
4 -> 1 3

DIRECTED GRAPH:
Adjacency Matrix:
0 1 3 4 0
0 0 0 0 2
0 0 0 1 0
0 0 0 0 5
0 0 0 0 0

Adjacency List:
0 -> 1 2 3
1 -> 4
2 -> 3
3 -> 4
4 ->


--------------------------------
Process exited after 0.117 seconds with return value 0
Press any key to continue . . .
```

Explanation:
In this program it illustrates two common graph representations: the adjacency matrix and adjacency list. It defines two classes, GraphMatrix and GraphList, which facilitate adding edges between vertices and displaying the graph in both formats. The graph can be either directed or undirected, with edges added based on the specified directionality. The adjacency matrix uses a 2D array to represent connections, where a non-zero value indicates an edge between vertices, while the adjacency list maintains a list of neighbors for each vertex. The main() function demonstrates these representations by creating and displaying a graph with 5 vertices, allowing comparison of both approaches.

ILO_B1:
Main CPP:

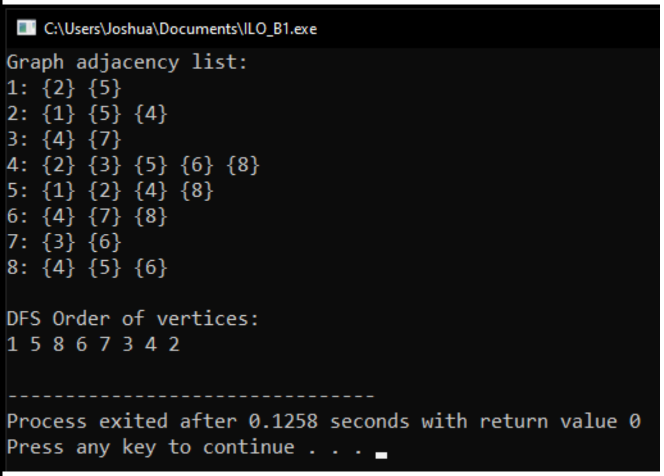


```cpp
#include <iostream>
#include <vector>
#include <stack>
#include <set>
#include <map>
using namespace std;

struct Edge {
    int src;
    int dest;
    int weight;
};

class Graph {
private:
    int V;
    vector<Edge> edgeList;

public:
    Graph(int vertices) {
        V = vertices;
    }

    void addEdge(int u, int v, int w = 0) {
        if (u >= 1 && u <= V && v >= 1 && v <= V) {
            edgeList.push_back({u, v, w});
        }
    }

    vector<Edge> outgoingEdges(int u) const {
        vector<Edge> result;
        for (auto e : edgeList) {
            if (e.src == u) result.push_back(e);
        }
        return result;
    }

    void printGraph() const {
        for (int i = 1; i <= V; i++) {
            cout << i << ": ";
            for (auto e : edgeList) {
                if (e.src == i) {
                    cout << "{" << e.dest << "} ";
                }
            }
            cout << endl;
        }
    }
};

vector<int> DFS(const Graph &G, int start) {
    stack<int> st;
    set<int> visited;
    vector<int> order;

    st.push(start);

    while (!st.empty()) {
        int node = st.top();
        st.pop();

        if (visited.find(node) == visited.end()) {
            visited.insert(node);
            order.push_back(node);

            for (auto e : G.outgoingEdges(node)) {
                if (visited.find(e.dest) == visited.end()) {
                    st.push(e.dest);
                }
            }
        }
    }
    return order;
```



```cpp
    return order;
}

Graph createGraph() {
    Graph G(8);
    map<int, vector<int>> edges = {
        {1, {2, 5}},
        {2, {1, 5, 4}},
        {3, {4, 7}},
        {4, {2, 3, 5, 6, 8}},
        {5, {1, 2, 4, 8}},
        {6, {4, 7, 8}},
        {7, {3, 6}},
        {8, {4, 5, 6}}
    };

    for (auto &p : edges) {
        for (auto v : p.second) {
            G.addEdge(p.first, v);
        }
    }
    return G;
}

int main() {
    Graph G = createGraph();

    cout << "Graph adjacency list:" << endl;
    G.printGraph();
    cout << endl;

    cout << "DFS Order of vertices:" << endl;
    vector<int> dfsOrder = DFS(G, 1);
    for (int v : dfsOrder) {
        cout << v << " ";
    }
    cout << endl;

    return 0;
}
```

Output:



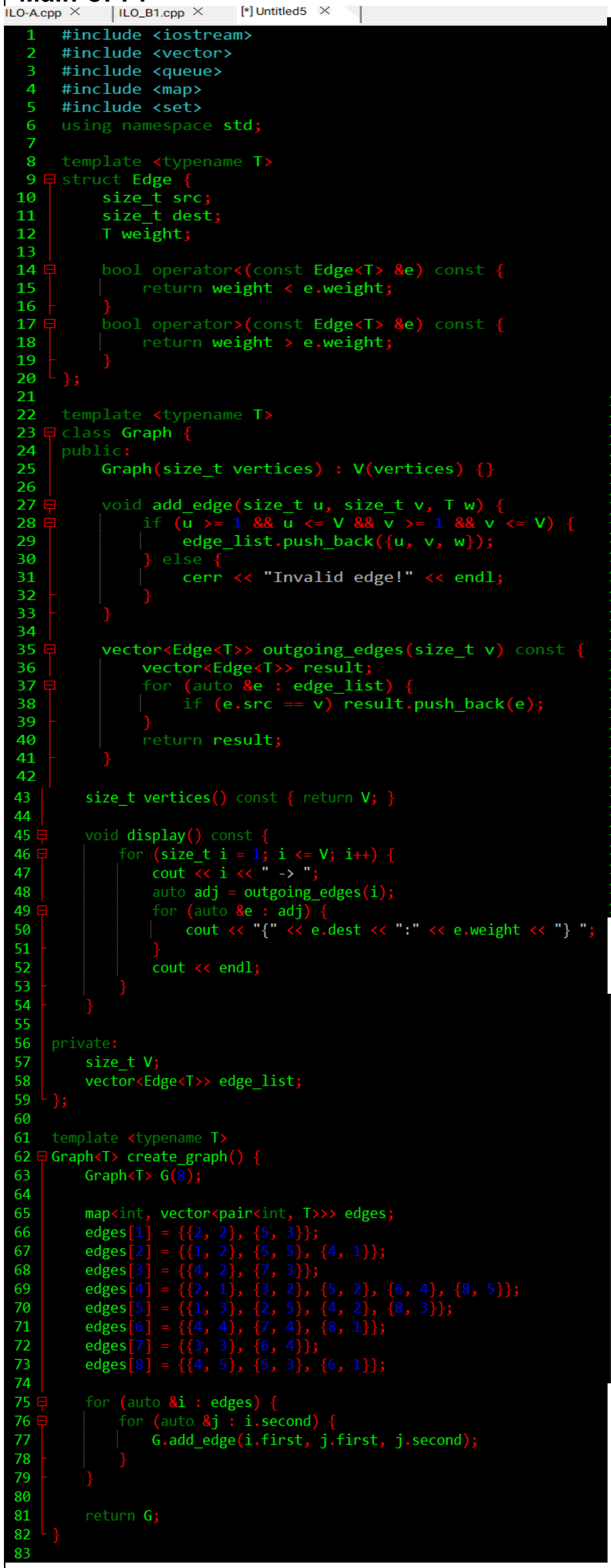
```
Graph adjacency list:
1: {2} {5}
2: {1} {5} {4}
3: {4} {7}
4: {2} {3} {5} {6} {8}
5: {1} {2} {4} {8}
6: {4} {7} {8}
7: {3} {6}
8: {4} {5} {6}

DFS Order of vertices:
1 5 8 6 7 3 4 2

--------------------------------
Process exited after 0.1258 seconds with return value 0
Press any key to continue . . .
```

Explanation:
This program implements a directed graph using an edge list and performs a Depth-First Search (DFS) traversal on it. The Graph class stores vertices and edges, allowing edges to be added between vertices with an optional weight. The DFS function uses a stack and a set to explore vertices starting from a given node, ensuring each vertex is visited once, and returns the order of traversal. The createGraph function builds a specific graph of 8 vertices and their edges based on a predefined adjacency list. Finally, the main function

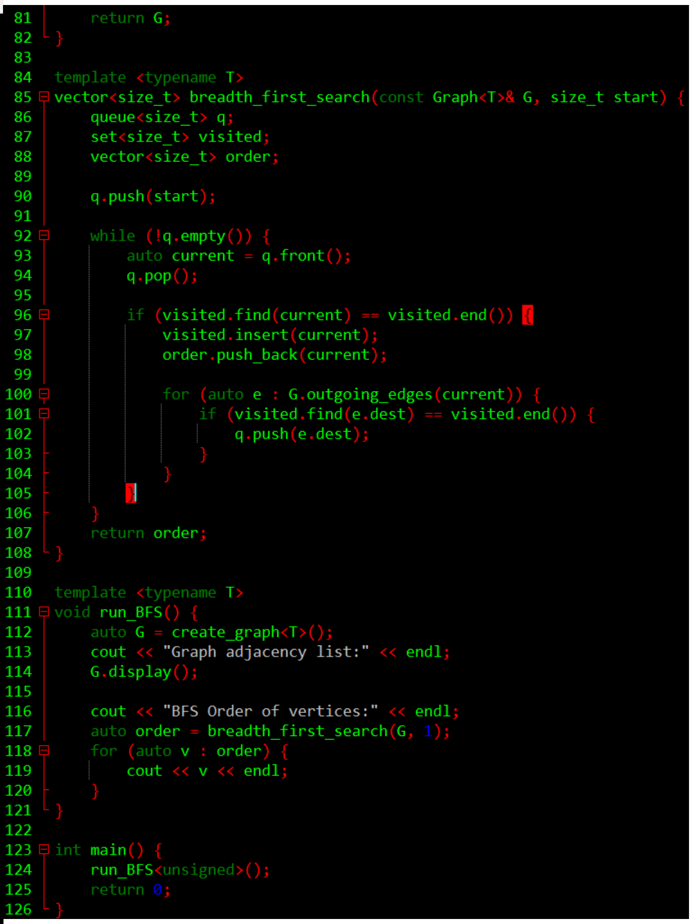
prints the adjacency list representation of the graph and displays the order in which vertices are visited during the DFS starting from vertex 1.

ILO_B2:

Main CPP:



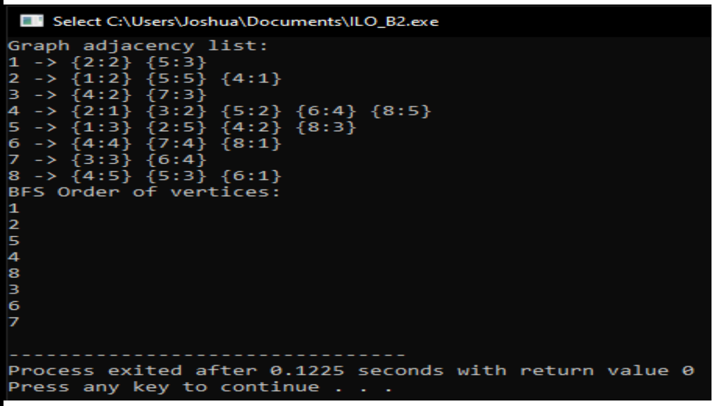
```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <set>
using namespace std;

template <typename T>
struct Edge {
    size_t src;
    size_t dest;
    T weight;

    bool operator<(const Edge<T> &e) const {
        return weight < e.weight;
    }
    bool operator>(const Edge<T> &e) const {
        return weight > e.weight;
    }
};

template <typename T>
class Graph {
public:
    Graph(size_t vertices) : V(vertices) {}

    void add_edge(size_t u, size_t v, T w) {
        if (u >= 1 && u <= V && v >= 1 && v <= V) {
            edge_list.push_back({u, v, w});
        } else {
            cerr << "Invalid edge!" << endl;
        }
    }

    vector<Edge<T>> outgoing_edges(size_t v) const {
        vector<Edge<T>> result;
        for (auto &e : edge_list) {
            if (e.src == v) result.push_back(e);
        }
        return result;
    }

   size_t vertices() const { return V; }

   void display() const {
       for (size_t i = 1; i <= V; i++) {
           cout << i << " -> ";
           auto adj = outgoing_edges(i);
           for (auto &e : adj) {
               cout << "{" << e.dest << ":" << e.weight << "} ";
           }
           cout << endl;
       }
   }

private:
   size_t V;
   vector<Edge<T>> edge_list;
};

template <typename T>
Graph<T> create_graph() {
   Graph<T> G(8);

   map<int, vector<pair<int, T>>> edges;
   edges[1] = {{2, 2}, {5, 3}};
   edges[2] = {{1, 2}, {5, 5}, {4, 1}};
   edges[3] = {{4, 2}, {7, 3}};
   edges[4] = {{2, 1}, {3, 2}, {5, 2}, {6, 4}, {8, 5}};
   edges[5] = {{1, 3}, {2, 5}, {4, 2}, {8, 3}};
   edges[6] = {{4, 4}, {7, 4}, {8, 1}};
   edges[7] = {{3, 3}, {6, 4}};
   edges[8] = {{4, 5}, {5, 3}, {6, 1}};

   for (auto &i : edges) {
       for (auto &j : i.second) {
           G.add_edge(i.first, j.first, j.second);
       }
   }

   return G;
}

```



```cpp
    return G;
}

```

```cpp
template <typename T>
vector<size_t> breadth_first_search(const Graph<T>& G, size_t start) {
    queue<size_t> q;
    set<size_t> visited;
    vector<size_t> order;

    q.push(start);

    while (!q.empty()) {
        auto current = q.front();
        q.pop();

        if (visited.find(current) == visited.end()) {
            visited.insert(current);
            order.push_back(current);

            for (auto e : G.outgoing_edges(current)) {
                if (visited.find(e.dest) == visited.end()) {
                    q.push(e.dest);
                }
            }
        }
    }
    return order;
}

template <typename T>
void run_BFS() {
    auto G = create_graph<T>();
    cout << "Graph adjacency list:" << endl;
    G.display();

    cout << "BFS Order of vertices:" << endl;
    auto order = breadth_first_search(G, 1);
    for (auto v : order) {
        cout << v << endl;
    }
}

int main() {
    run_BFS<unsigned>();
    return 0;
}
```

Output:



```
Graph adjacency list:
1 -> {2:2} {5:3}
2 -> {1:2} {5:5} {4:1}
3 -> {4:2} {7:3}
4 -> {2:1} {3:2} {5:2} {6:4} {8:5}
5 -> {1:3} {2:5} {4:2} {8:3}
6 -> {4:4} {7:4} {8:1}
7 -> {3:3} {6:4}
8 -> {4:5} {5:3} {6:1}
BFS Order of vertices:
1
2
5
4
8
3
6
7

--------------------------------
Process exited after 0.1225 seconds with return value 0
Press any key to continue . . .
```

Explanation: This program implements a weighted directed graph using templates to support generic edge weights. The Graph class stores vertices and edges in an edge list and provides functions to add edges and retrieve outgoing edges for a given vertex. The graph is built with a predefined set of weighted edges connecting 8 vertices, and its adjacency list representation is displayed. The Breadth-First Search (BFS) algorithm is

implemented using a queue and a set to traverse the graph level by level, starting from vertex 1. Finally, the program outputs the order in which vertices are visited during the BFS traversal.

B. Answers to Supplementary Activity:

1. **A person wants to visit different locations indicated on a map. He starts from one location (vertex) and wants to visit every vertex until it finishes from one vertex, backtracks, and then explore other vertex from same vertex. Discuss which algorithm would be most helpful to accomplish this task.**

- **In my opinion the most suitable for this task is Depth-First Search (DFS). DFS starts at one vertex and explores as far as possible along each path before backtracking. This helps cover all vertices deeply, one path at a time. It is especially useful when you want to explore all possible routes from any location. After finishing one path, it returns to explore the next unvisited path from the same point.**

2. **Identify the equivalent of DFS in traversal strategies for trees. To efficiently answer this question, provide a graphical comparison, examine pseudocode and code implementation.**

Graphical Comparison:

```
     A
    / \
   B   C
  / \
 D   E
```

DFS/Preorder: A → B → D → E → C

Pseudocode:

```
void preorderTraversal(Node* node) {
    if (node != NULL) {
        visit(node);              // Process current node
        preorderTraversal(node->left);   // Traverse left subtree
        preorderTraversal(node->right);  // Traverse right subtree
    }
}
```

Code Implementation:

```
#include <iostream>
using namespace std;

struct Node {
    char data;
    Node* left;
    Node* right;

    Node(char value) {
        data = value;
        left = right = NULL;
    }
};

void preorderTraversal(Node* node) {
    if (node != NULL) {
        cout << node->data << " ";
        preorderTraversal(node->left);
        preorderTraversal(node->right);
    }
}

int main() {
    // Construct the tree
    //       A
    //      / \
    //     B   C
    //    / \
    //   D   E

    Node* root = new Node('A');
    root->left = new Node('B');
    root->right = new Node('C');
    root->left->left = new Node('D');
    root->left->right = new Node('E');

    cout << "Preorder Traversal: ";
    preorderTraversal(root);  // Output: A B D E C

    return 0;
}
```

Explanation:
In these trees the traversal technique is equivalent to Depth First Search (DFS) is pre order traversal. Both DFS and Pre order traversal follows a deep first approach, where a user explores a node before its subtrees. Pre order traversal specifically follows the Root Left → Right pattern: the root node is processed first, followed by the left manner making sure that the user is examine each node before moving into deeper branches. Pre order traversal like DFS can be made using either recursion or an explicit stack and is a key technique in understanding tree structures.

3. **In the performed code, what data structure is used to implement the Breadth First Search?**
- **To implement the Breadth First Search (BFS) the queue is used. A queue follows the First in First out (FIFO) principle. The nodes are added to the queue as they are discovered. Each node is then processed in the order it was added. This ensures that BFS visits all nodes level by level.**

4. **How many times can a node be visited in the BFS?**
- **In the BFS the node is visited only once. Because once the node is marked as visited it is not added to the queue again. This prevents unnecessary repetition and ensures efficiency. Visiting each node once guarantees that the smallest path is found in unweighted graphs. Using a visited list or set helps to the track which nodes have already been explored.**

C. Conclusion & Lessons Learned: • **In this laboratory activity, I understand the differences between DFS and BFS and when to use each one. DFS is good for going deep into this paths while BFS is better for exploring level by level. I also understood how important the data apply what I learned in code and see how traversal works step by step. I think I need to practice and keep learning with complex graphs and edge cases to improve my practical skills more.**
D. Assessment Rubric
E. External References