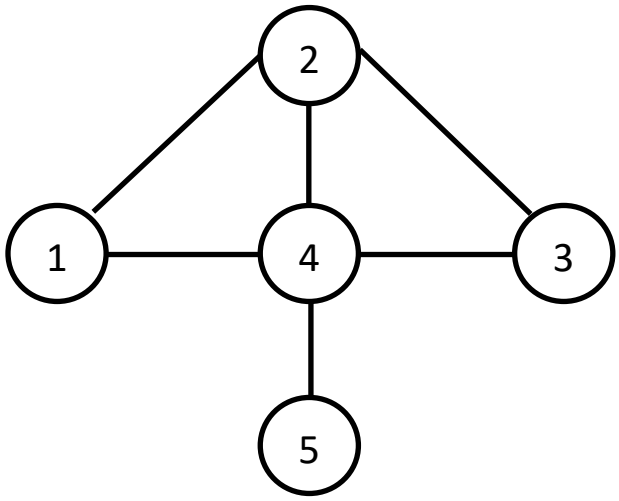
CIS 2520
Lab 07

1) Graph Overview

A graph is a data structure that consists of a finite set of vertices (nodes) and a set of edges connecting the vertices. A pair of vertices are considered adjacent if they are connected by an edge. A path represents a sequence of edges between two vertices.



There are two types of graphs: directed and undirected graphs. In directed graphs, nodes are connected by directed edges. They only go in one direction. This means if an edge connects nodes *u* and *n*, but it directed towards *n*, we can only traverse the graph from edge *u* to *n*. On the other hand, if we have an undirected graph and an edge connecting nodes *u* and *n*. We can traverse from both *u* to *n* and *n* to *u*. For this lab, we will be using an undirected graph. Graphs can also be weighted or unweighted. This means there is some value or cost associated with each edge between nodes. You can also have cyclic graphs when the graph consists of a cyclic path. A cyclic path means that the graph begins at one vertex and ends at the same root. Graphs that have no cycle are known as an acyclic graph.

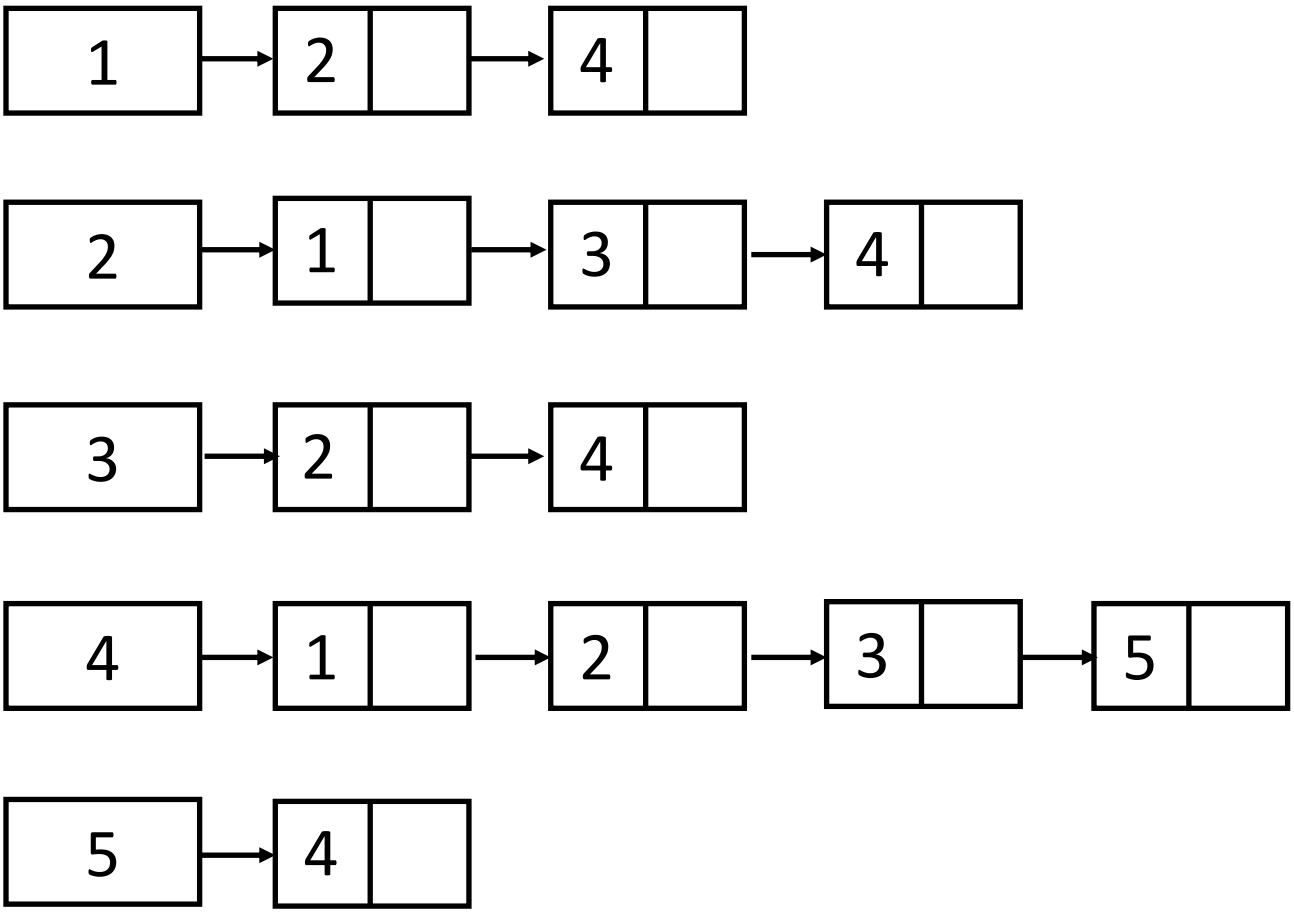
Graphs have a myriad of applications. It can be used to represent networks of communication, data organization, computational devices, flow of computation. One major application is to model social networks online. Each node represents entities such as users, groups, comments, pages, etc, and edges represent a connection or relationship between these entities. If you are familiar with APIs, the Facebook Graph API is a great application of graphs to real world problems. You can also use graphs for path optimizer, such as Google Maps. Google Maps uses a weighted graph to find the shortest path from one vertex to another.

There are two main ways to represent a graph: an adjacency matrix or an adjacency list. An adjacency matrix is a two-dimensional array of size *V* x *V*, where *V* is the number of nodes in the graph. If an index in the matrix is labelled 1 (matrix[i][j] == 1), indicates there is an edge from

node *i* to *j*. If an index in the matrix is labelled 0 (matrix[i][j] == 0), indicates there is no edge from node *i* to *j*.

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	0
3	0	1	0	1	0
4	1	1	1	0	1
5	0	0	0	1	0

The advantages of an adjacency matrix are that representation is easier to implement and follow. Removing an edge takes O(1) time. Determining if there is an edge between two vertices can be done in O(1). The cons are that it consumes more space and adding a vertex is O(n^2). The adjacency matrix is most helpful in cases where the graph does not contain many nodes. On the other hand, we can use an adjacency list. An adjacency list is an array of lists, where the size of the array is equal to the number of vertices. An index in the array represents the list of vertices adjacent to the *i*th vertex. The advantages of an adjacency list is that it saves space O(|n|+|e|), and at worst case O(n^2). Determining whether there is an edge from *u* to *v* are not efficient. Adjacency lists are a great option when we need to continuously access all the neighbors of some node *n*.



2) Your task

For this lab, you task is to create a basic graph using an adjacency list. Your program should do the following:

- Initialize a new graph structure.
- Add a vertex to the graph.
- Add an edge to the graph.
- Traverse the graph.
- Check if an element is present in the graph.
- Return the number of vertices and edges.
- Delete the graph.

Note: while you can use an array to represent your vertices, we will be using a linked list instead. This gives us more flexibility with our graph, allowing us to add and remove as many vertices as we want.

Part A: Initial setup

```
typedef struct adjVertex {
     int value;
     struct adjVertex *next;
} AdjVertex;

typedef struct vertex {
     int value;
     AdjVertex *adjList;
     struct vertex *next;
} Vertex;

typedef struct graph {
     int numVertices;
     int numEdges;
     Vertex *vertexList;
} Graph;
```

1. Create a new file called graph.c and a corresponding header file called graph.h.
2. Add the three structs to your graph.h file as well as any function stubs you use in your graph.c.
3. Create a makefile to compile and run your code.

Part B: Create a function to initialize your graph.

```
Graph* createGraph(int numVertices);
```

1. Create a new graph and malloc the size of Graph.
2. Set graph->numVertices to 0.
3. Set graph->numEdges to 0.

4. Set graph->vertexList to null.
5. Return the initialized Graph struct.

Part C: Create a function to create a new vertex.

```
Vertex* createVertex(int value);
```

1. Create a new vertex and malloc the size of Vertex.
2. Set vertex->value to value passed in.
3. Set vertex->adjList to null.
4. Set vertex->next to null.
5. Return the initialized Vertex struct.

Part D: Create a function to create a new adjacency vertex.

```
AdjList* createAdjVertex(int value);
```

1. Create a new AdjVertex and malloc the size of AdjVertex.
2. Set AdjVertex ->value to value passed in.
3. Set AdjVertex ->list to null.
4. Set AdjVertex ->next to null.
5. Return the initialized Adjacency Vertex struct.

Part E: Create a function to add a new node and affiliated adjacency list.

```
void addVertex(Graph* graph, int value);
```

1. Using createVertex(), create a new Vertex (*v*) with value.
2. If graph->vertexList is null, then assign it to *v*.
3. If it is not null, you will need to move through each element in the list until you reach the final element. You will then assign the next pointer of the last element of the vertexList to *v*. This will add *v* to the end of the list.
4. Increment numVertices.

Part F: Create a function to add an adjacency vertex to a vertex in the graph.

```
void addAdjVertex(Vertex* vertex, int value);
```

1. Using createAdjVertex(), create a new AdjVertx (*v*) with value.
2. If vertex->adjList is null, then assign it to *v*.
3. If it is not null, you will need to move through each element in the list until you reach the final element. Assign the next pointer of the last element of the adjList to *v*. This will add *v* to the end of the list.

Part G: Create a function to add an edge to the graph.

```
void addEdge(Graph* graph, int vertex1, int vertex2);
```

1. For this function you can assume that there is at least one vertex, and both vertex1 and vertex2 exist.
2. You will need to loop through the vertex list.
3. If current vertex->value equals vertex1, then you will use addAdjVertex() to add a new AdjVertex to the current vertex with value vertex2.
4. If current vertex->value equals vertex2, then you will use addAdjVertex() to add a new AdjVertex to the current vertex with value vertex1.
5. Increment the number of edges.

Part H: Create a function to print your graph.

```
void printGraph(Graph* graph);
```

1. If the vertex list is null, print "No vertices in the graph".
2. If it is not null, you must loop through each vertex in the vertexList.
3. Within each vertex, loop through each AdjVertex in the AdjList.
4. Print out each the value of each vertex in the vertexList alongside the value of each adjVertex in the corresponding adjList. Add a newline between each vertex.

Part I: Create a function to check if a vertex is present in your graph.

```
int checkForVertex(Graph* graph, int src);
```

1. Loop through each vertex in the vertexList.
2. If the vertex->value equals src, then return 1.
3. If the vertex is not present in the list, return 0.

Part J: Create helper functions to get number of vertices and edges.

```
int getNumEdges(Graph* graph);
int getNumVertices(Graph* graph);
```

1. Return either numEdges or numVertices.

Part K: Free your graph.

```
void freeGraph(Graph* graph);
freeAdjVertexList(AdjVertex* list);
freeVertexList(Vertex* node);
```

1. Create a function to delete and free your graph.
2. Your freeGraph() should call your freeVertexList() which should call your freeAdjVertexList().

Part L: Put it all together in your main.

1. In your main, initialize a new graph.

2. Add 5 vertices to your graph, 1, 2, 3, 4, and 5.
3. Add edges between the following vertices:
    a. 1 and 2
    b. 2 and 3
    c. 2 and 4
    d. 3 and 4
    e. 4 and 1
    f. 4 and 5
4. Print your graph.
5. Check to see if vertex 3 is present (should return 1).
6. Check to see if vertex 6 is present (should return 0).
7. Check to see if you get the correct number of edges.
8. Check to see if you get the correct number of vertices.
9. Free your graph.
10. Run valgrind to ensure you don't have any memory leaks.
11. Your output should look something like this:

```
Vertex 1 has edges with vertices: 2 4
Vertex 2 has edges with vertices: 1 3 4
Vertex 3 has edges with vertices: 2 4
Vertex 4 has edges with vertices: 2 3 1 6
Vertex 5 has edges with vertices: 4
Checking for Vertex 5: 1
Checking for Vertex 6: 0
Number of vertices: 5
Number of edges: 6
```