# Department of Computing

**CS 250: Data Structures and Algorithms**

**Class: BSCS-9AB**

**Lab 14: Implement Graphs and Traverse them in Breadth-First Order**

**Date: January 12, 2021**

**Time: 10:00 am -1:00pm, 2:00pm – 5:00pm**

# Instructor: Dr. Yasir Faheem

**Lab 14: Implement Graphs and Traverse them in Breadth-First Order**

**Introduction:**

This lab is based on graphs data structure. You should learn how to implement graphs and how to traverse one using Breadth-First Traversal algorithm.

**Objective:**

In this lab, you will implement graphs using adjacency matrix and adjacency list-based approaches. Moreover, you shall implement the Breadth-First Search traversal algorithm studied in the class.

**Description of Graphs:**

A graph data structure is a collection of nodes that have data and are connected to other nodes.

Let's try to understand this through an example. On Facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link. Note that anything that has data is a node. Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship. (source: https://www.programiz.com/dsa/graph).

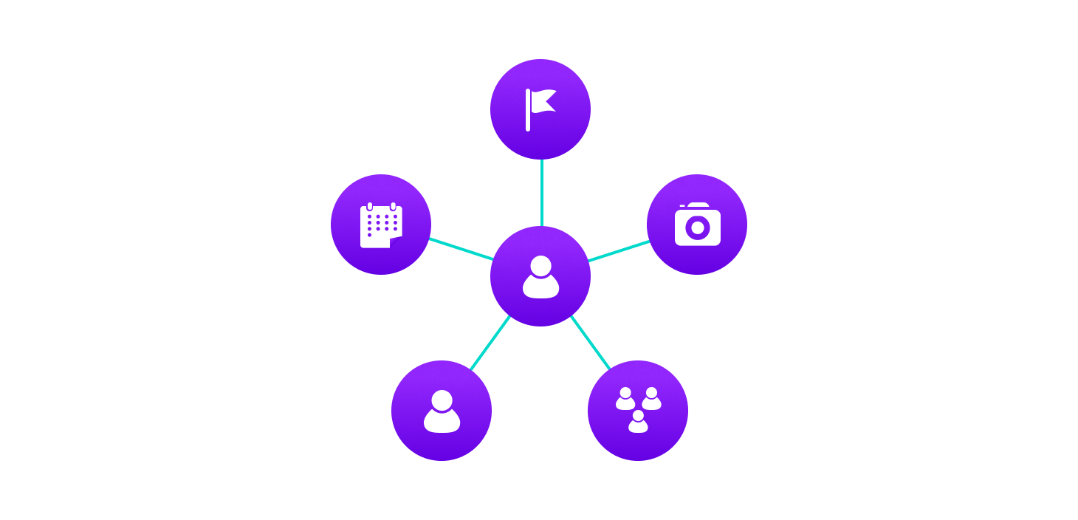


Figure 1:Example of graph data structure

All of Facebook is then a collection of these nodes and edges. This is because Facebook uses a graph data structure to store its data.

More precisely, a graph is a data structure (V, E) that consists of

* A collection of vertices V
* A collection of edges E, represented as ordered pairs of vertices (u,v)

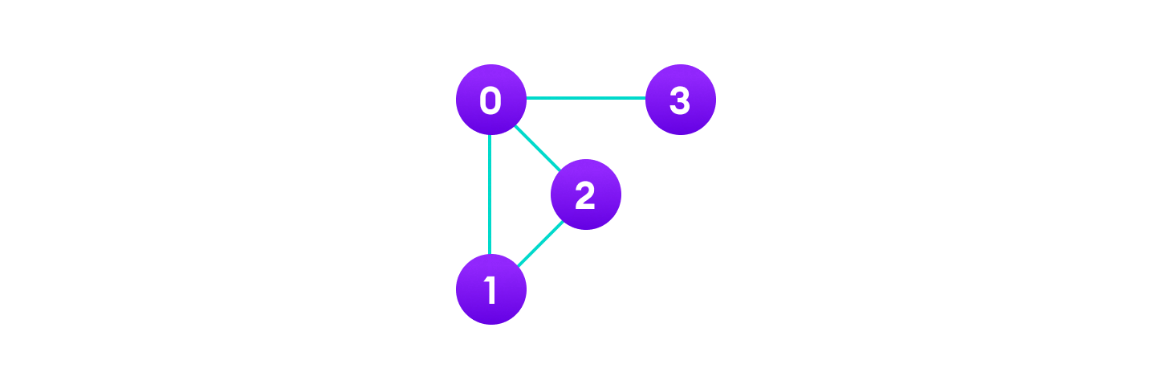


Figure 2: Vertices and edges

In the graph,

V = {0, 1, 2, 3}

E = {(0,1), (0,2), (0,3), (1,2)}

G = {V, E}

**Graph Terminology**

* **Adjacency**: A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
* **Path**: A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
* **Directed Graph**: A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

**Graph Representation**

Graphs are commonly represented in two ways:

**1. Adjacency Matrix**

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex. If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

The adjacency matrix for the graph we created above is:

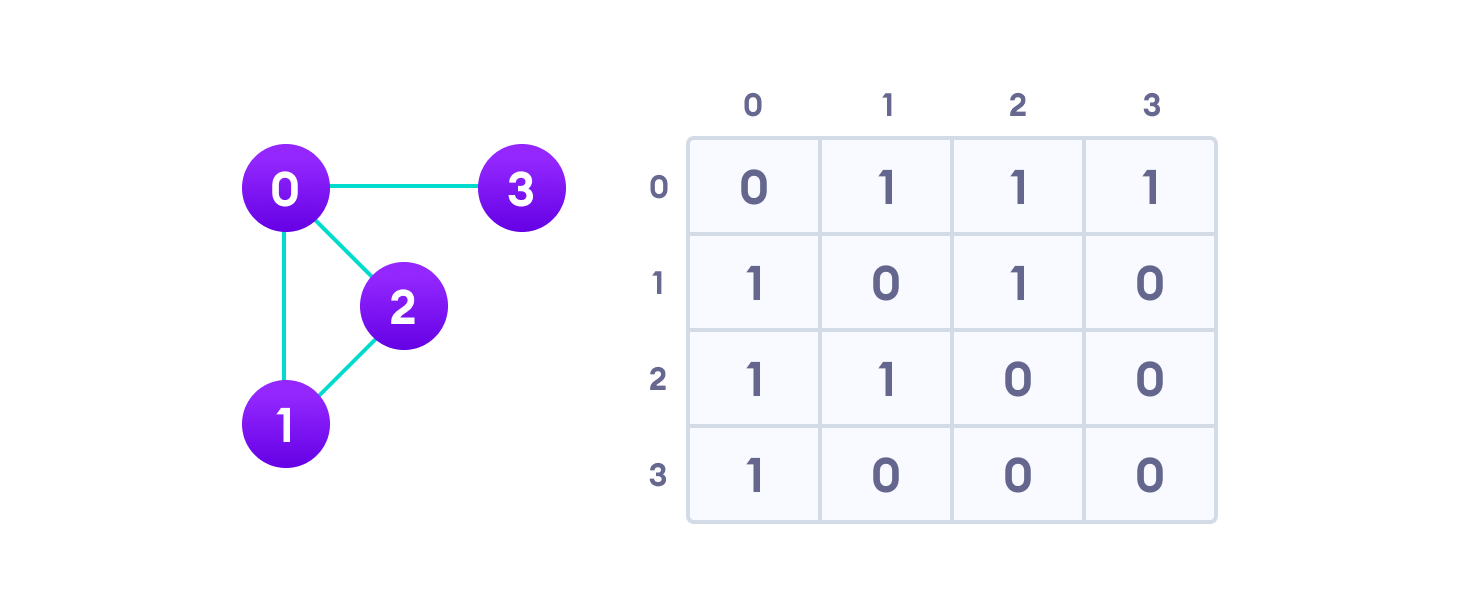


Figure 3: Graph adjacency matrix

Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

Edge lookup (checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices (V x V), so it requires more space.

**Adjacency Matrix of a Weighted Graph:** If the graph is weighted, then you should store the weight of the edge instead of 1 between vertices vi and vj.

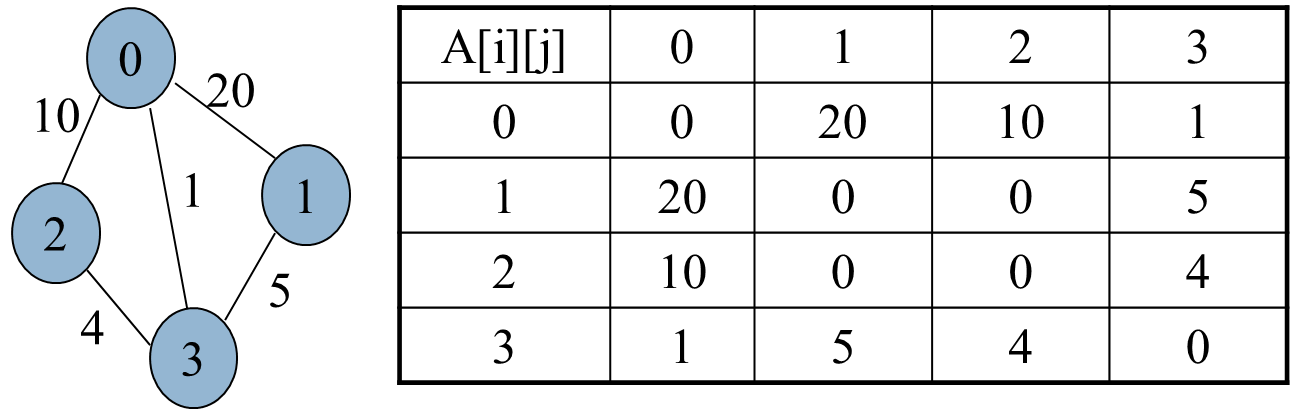


Figure 4: Adjacency Matrix of a Weighted Graph

**Adjacency Matrix of a Directed Graph:** In a directed graph, if there is an edge from vertex vi to vj, then index A[i][j] should store 1. It may store the weight of an edge if the graph is weighted.

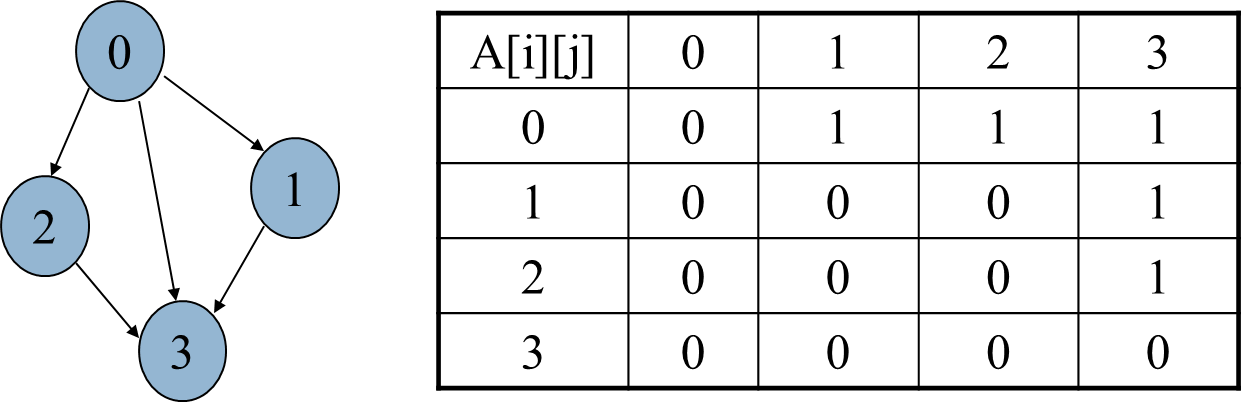
****

Figure 5: Adjacency Matrix of a Directed Graph

**2. Adjacency List**

An adjacency list represents a graph as an array of linked lists. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex. The adjacency list for the graph we made in the first example is as follows:

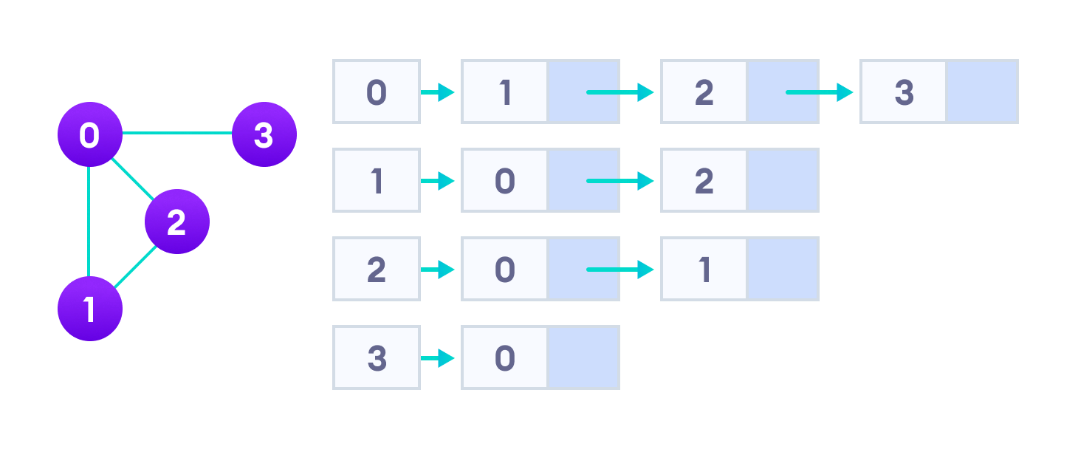


Figure 6: Adjacency list representation

An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

**Graph Operations**

The most common graph operations are:

* Check if the element is present in the graph
* Graph Traversal
* Add elements (vertex, edges) to graph
* Finding the path from one vertex to another

**Lab Tasks:**

In this lab, first you should implement the following operations of a graph using both Adjacency Matrix and Adjacency List based representations. Then, you shall implement the breadth-first traversal algorithm using adjacency-list based representation.

**Task A: Implement Adjacency Matrix Representation**

* **Initialize a graph G = (V, E) with |V| vertices in it.**
  + Ask user to enter the number of vertices in a graph.
  + Ask user to enter whether the graph is directed or undirected.
* **InsertEdge(AdjacencyMatrix[], int u, int v, int weight)**
  + Inserts an edge along with its weight from vertex u to v.
  + Set weight to 1 for all edges if the graph is unweighted.
  + If the graph is undirected, you should insert two edges: one from u to v, and another from v to u.
* **PrintGraph(AdjacencyMatrix[])**

This function should print the graph i.e. adjacency matrix.

* **Bool isConnected(int u, int v)**

Checks whether one can go from vertex u to v. Returns true if yes, false otherwise.

**CODE**

#include<iostream>

using namespace std;

int array[5][5];

class Graph

{public:

void insert(int array[5][5],int v1, int v2, int weight)

{

array[v1][v2] =weight;

array[v2][v2] = weight;

}

void insertdirected(int array[5][5], int v1, int v2, int weight)

{

array[v1][v2] = weight;

}

bool isconnected(int array[5][5], int v1, int v2)

{

bool connection = false;

if (array[v1][v2] != 0)

{

return true;

}

else

{

int save = v2;

int savev1 = v1;

//for (v1 = v1; v1 < 5; v1++)

do

{ connection = false;

for (v2 = 0; v2 < 5; v2++)

{

if (array[v1][v2] != 0)

{

connection = true;

if (v2 == save)

{

cout << "reached there" << endl;

return true;

}

v1 = v2;

cout << v1<<endl;

break;

}

}

if (!connection)

{

cout << "here it goes";

return false;

}

} while (v1 != savev1);

return false;

}

}

void PrintGraph(int array[5][5])

{

for (int i = 0; i < 5; i++)

{

for (int j = 0;j < 5; j++)

{

cout << array[i][j];

}

cout << endl;

}

}

};

int main()

{

int n;

bool directed=false;

cout << " Is the graph directed?";

cin >> directed;

int array[5][5] = { {0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0},{0,0,0,0,0} };

Graph\* g = new Graph();

if (directed)

{

g->insertdirected(array, 0, 1, 1);

g->insertdirected(array, 0, 3, 1);

g->insertdirected(array, 1, 4, 1);

g->insertdirected(array, 4, 2, 1);

}

else

{

g->insert(array, 0, 1, 1);

g->insert(array, 0, 3, 1);

g->insert(array, 1, 4, 1);

g->insert(array, 4, 2, 1);

}

g->PrintGraph(array);

cout<<g->isconnected(array, 1, 3);

}

**OUTPUT**



**Task B: Implement Adjacency List Representation**

* **Initialize a graph G = (V, E) with |V| vertices in it.**
  + Ask user to enter the number of vertices in a graph.
  + Ask user to enter whether the graph is directed or undirected.
* **InsertEdge(AdjacencyMatrix[], int u, int v, int weight)**
  + Inserts an edge along with its weight from vertex u to v.
  + Set weight to 1 for all edges if the graph is unweighted.
  + If the graph is undirected, you should insert two edges: one from u to v, and another from v to u.
* **PrintGraph(AdjacencyList[])**

This function should print the graph i.e. the adjacency list of each of the |V| vertices.

* **Bool isConnected(int u, int v)**

Checks whether one can go from vertex u to v. Returns true if yes, false otherwise.

#include<iostream>

using namespace std;

class ListNode;

struct ListNode

{

int data;

ListNode\* next;

};

class Node {

public:

int data;

ListNode\* Next;

};

class Graph

{

public:

Graph()

{

}

Graph(Node\*\* arr, int n)

{

for (int i = 0; i < n; i++)

{

arr[i]->Next = new ListNode();

}

}

void InsertVertices(Node\* a, int v1)

{

a->data = 1;

}

void InsertEdges(Node\*\* arr,int v1,int v2)

{

ListNode\* node=new ListNode();

if (v1 == 0)

{

if (arr[0]->Next->next == NULL)

{

arr[0]->Next->next = node;

arr[0]->Next->next->data = v2;

arr[0]->Next->next->next = NULL;

}

else

{

ListNode\* temp = arr[0]->Next->next;

ListNode\* ptemp = NULL;

while (temp != NULL)

{

ptemp = temp;

temp = temp->next;

}

temp = node;

ptemp->next = temp;

temp->next = NULL;

temp->data = v2;

}

}

if (v1 == 1)

{

if (arr[1]->Next == NULL)

{

arr[1]->Next = node;

arr[1]->Next->data = v2;

arr[1]->Next->next = NULL;

}

else

{

ListNode\* temp = arr[1]->Next->next;

while (temp != NULL)

{

temp = temp->next;

}

temp = node;

temp->next = NULL;

temp->data = v2;

}

}

if (v1 == 2)

{

if (arr[2]->Next == NULL)

{

arr[2]->Next = node;

arr[2]->Next->data = v2;

arr[2]->Next->next = NULL;

}

else

{

ListNode\* temp = arr[2]->Next->next;

while (temp != NULL)

{

temp = temp->next;

}

temp = node;

temp->next = NULL;

temp->data = v2;

}

}

if (v1 == 3)

{

if (arr[3]->Next == NULL)

{

arr[3]->Next = node;

arr[3]->Next->data = v2;

arr[3]->Next->next = NULL;

}

else

{

ListNode\* temp = arr[3]->Next->next;

while (temp != NULL)

{

temp = temp->next;

}

temp = node;

temp->next = NULL;

temp->data = v2;

}

}

if (v1 == 4)

{

if (arr[4]->Next == NULL)

{

arr[4]->Next = node;

arr[4]->Next->data = v2;

arr[4]->Next->next = NULL;

}

else

{

ListNode\* temp = arr[1]->Next->next;

while (temp != NULL)

{

temp = temp->next;

}

temp = node;

temp->next = NULL;

temp->data = v2;

}

}

}

void PrintGraph(Node\*\* arr , int v)

{

for (int i = 0; i <= v; i++)

{

ListNode\* temp = arr[i]->Next->next;

while (temp != NULL)

{

cout << temp->data;

temp = temp->next;

}

}

}

};

int main()

{

Node\*\* arr=new Node\*[5];

for (int i = 0; i < 5; i++)

{

arr[i] = new Node();

}

Graph\* g= new Graph(arr,5);

g->InsertVertices(arr[0], 0);

g->InsertVertices(arr[1], 1);

g->InsertVertices(arr[2], 2);

g->InsertVertices(arr[3], 3);

g->InsertVertices(arr[4], 4);

g->InsertEdges(arr, 0, 2);

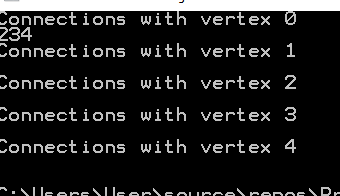
g->InsertEdges(arr, 0, 3);

g->InsertEdges(arr, 0, 4);

g->PrintGraph(arr, 1);

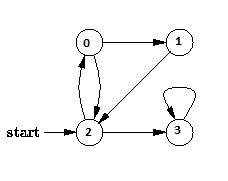
}

**Output:**



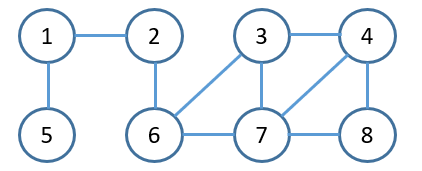
**Task C: Implement Breadth-First Traversal of a Graph Stored as Adjacency List Representation**

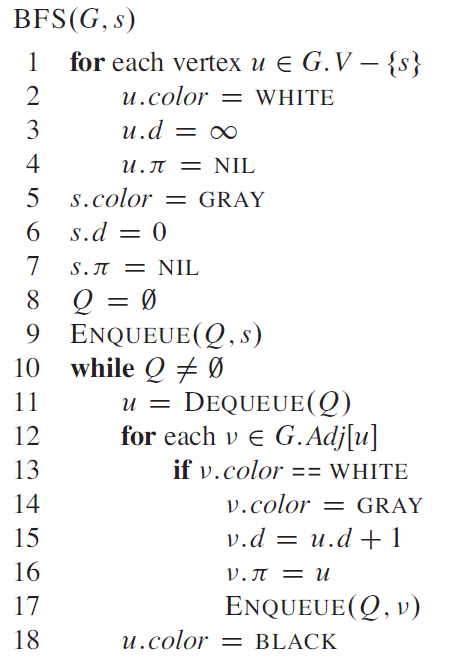
Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex. For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.

[](https://www.geeksforgeeks.org/wp-content/uploads/BFS.jpg)

**Task**

Implement BFS traversal algorithm studied in class for a graph with 8 vertices.

As an initial input we have the graph (G) with 8 nodes shown below with and a starting vertex 2.

****The Pseudocode of BFS is given below:

Your code should print the correct BFS sequence for an arbitrary starting vertex s.

**Deliverables**

Students are required to upload the lab on LMS before deadline.

**Note:**

* Take screenshots of the output of your code for all the three tasks and paste them in the word document.
* Use proper indentation and comments. Lack of comments and indentation will result in deduction of marks.
* You will submit your workingcodes in **word document** (do **NOT** take screenshot of code, just copy your code and paste it.
* The name of word document should follow this format. i.e. **YOUR\_NAME\_Lab#**