CSS 343 Assignment #3

## **Part 2, Programming** Graph ADT (emphasis on depth-first search)

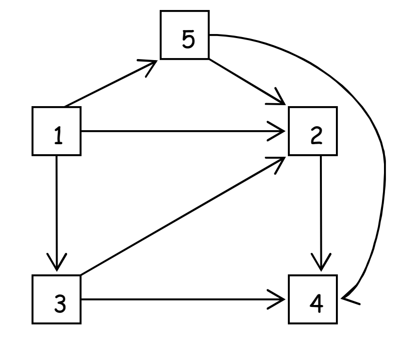
Display the graph information and implement depth-first search (using the ordering as given by the data, i.e., start at one). Display the node numbers in depth-first order. Implement the specified graph ADT functions.

In the data, the first line tells the number of nodes, say n (assume a nonnegative integer). Following is a text description of each of the 1 through n nodes, one description per line (max length of 50 characters). After that, each line consists of 2 ints representing an edge. (Assume int data.) For an edge from node 1 to node 2, the data is: `1 2`. A zero for the first integer signifies the end of the data for that one graph. All the edges for the first node will be together first, then all the edges for the second node, etc. Take them as they come, no sorting. (They appear sorted because the data is in order. Don’t assume that happens.) There are many graphs, each having at most 100 nodes. As in part1, assume the input data file has correctly formatted data. You may assume the number of nodes is a valid int, but you must error check for valid ints for the rest. Ignore invalid data, meaning don't use in the graph. E.g.,

Sample Input:

```
5
Aurora and 85th
Green Lake Starbucks
Woodland Park Zoo
Troll under bridge
PCC
1 2
1 3
1 5
2 4
3 2
3 4
5 2
5 4
0 0
```

picture (not part of data):



Sample Output

```
Graph:
Node 1       Aurora and 85th
  edge 1 2
  edge 1 3
  edge 1 5
Node 2       Green Lake Starbucks
  edge 2 4
Node 3       Woodland Park Zoo
  edge 3 2
  edge 3 4
Node 4       Troll under bridge
Node 5       PCC
  edge 5 2
  edge 5 4

Depth-first ordering: 1 2 4 3 5
```

### Part 2 Notes

-- Implement using an adjacency list (array of lists). Add any needed data, e.g., a field in the graph node can be used to mark visiting a node (used in depthFirstSearch() ). The object with the information on a graph node can either be directly stored in GraphNode as a NodeData object or as a pointer to an object (NodData*). As with part 1, start in array element one.

```
struct GraphNode {           // structs for simplicity (no fns), use class if desired
   STL list of ints          // list of edges (edge nodes), the adjacent graph node
   NodeData or NodeData*     // information on the graph node
   etc.
};

class GraphL {
public:
   etc.
private:
   // array of GraphNodes, static array is fine, assume at most 100 nodes
};
```

-- You do not need to implement a complete Graph class. Normally you would have a copy constructor, etc., but you don't need to implement all of them if you promise not to do that in real life. Implement the constructor, destructor, buildGraph(), displayGraph(), and depthFirstSearch(). A driver that (partially) tests your code is given.

-- To simplify things, you should always insert edges (edge nodes) at the end of the adjacency list of edges for the graph node (see the example above). An edge list may not be sorted. (They are sorted in the example output for readability.) Make sure to follow this coding simplification (and process the edges in the order they are in the list,) since it affects the depth-first ordering that you will get.

-- Note that you must handle a graph with disconnected components, e.g., add graph nodes 6 and7 to the above example with an edge between them, (6,7), with no other edges added. With these added nodes and edge, the DFS ordering is `1 2 4 3 5 6 7 .`