**Task 2.**

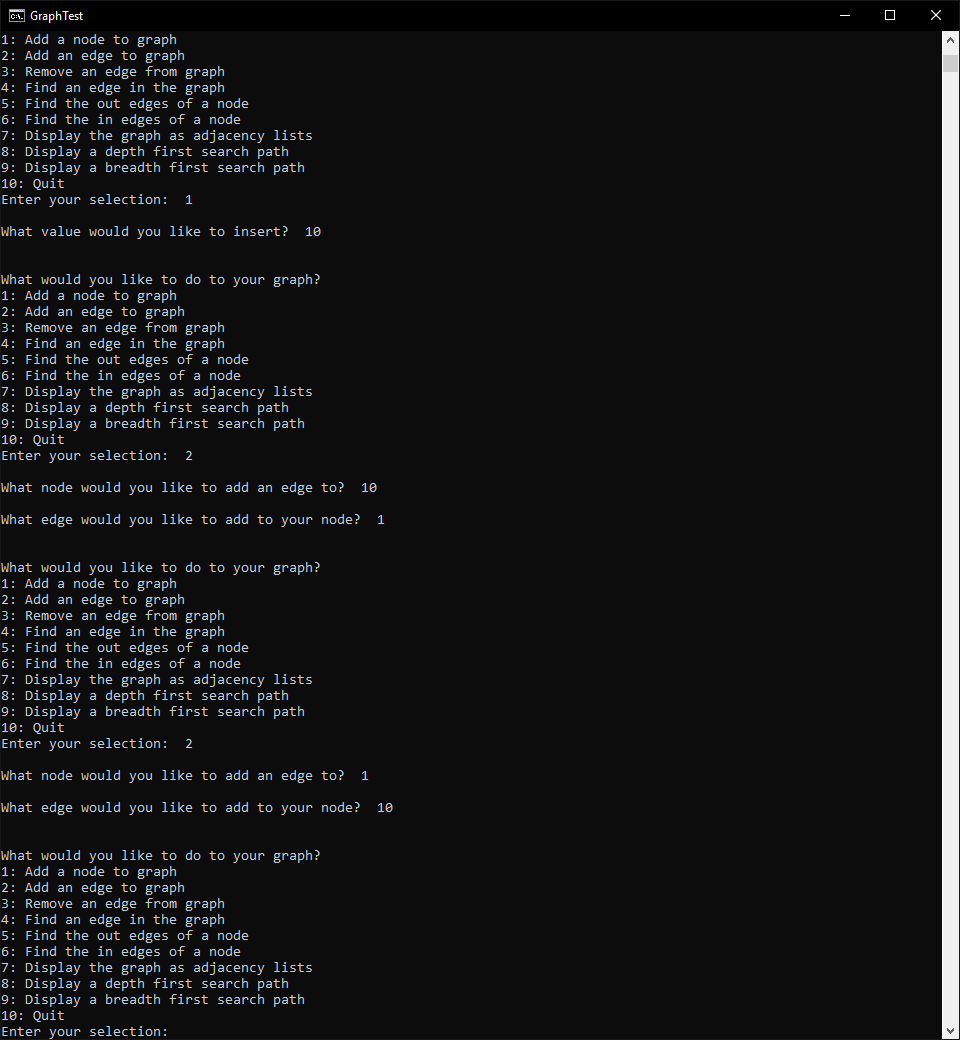


Figure – Inserting a Node and Edges

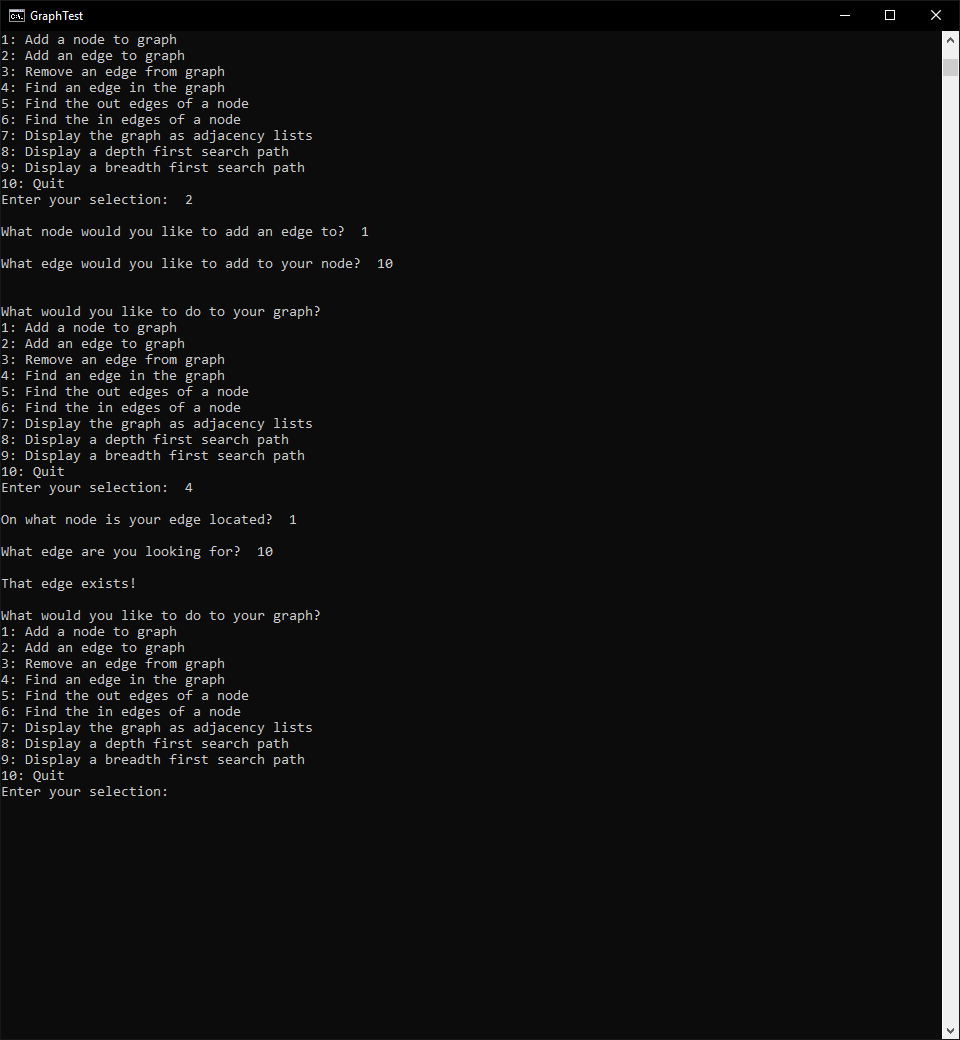


Figure – Using Find Edge to test the insertion from Figure 1

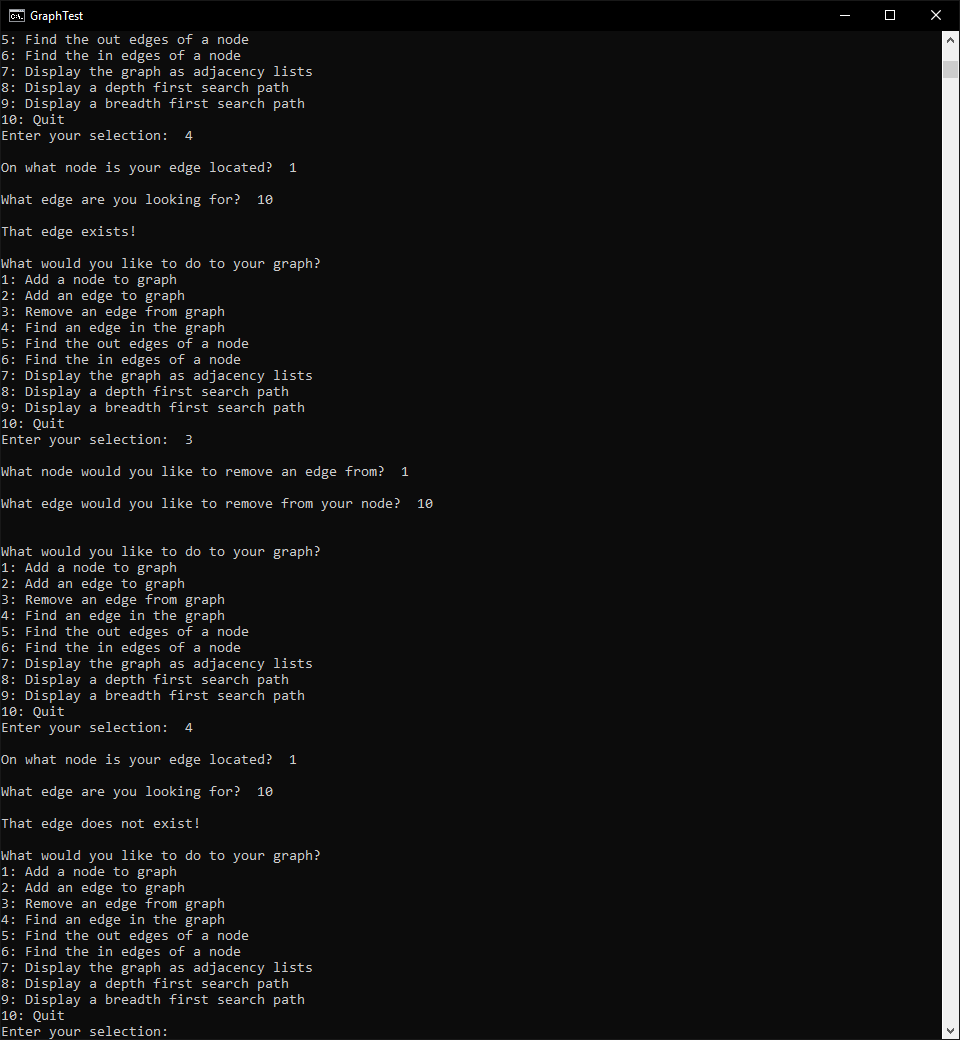


Figure 3 – Removing the Edge Created in Figure 1 and using Find Edge to verify it no longer exists

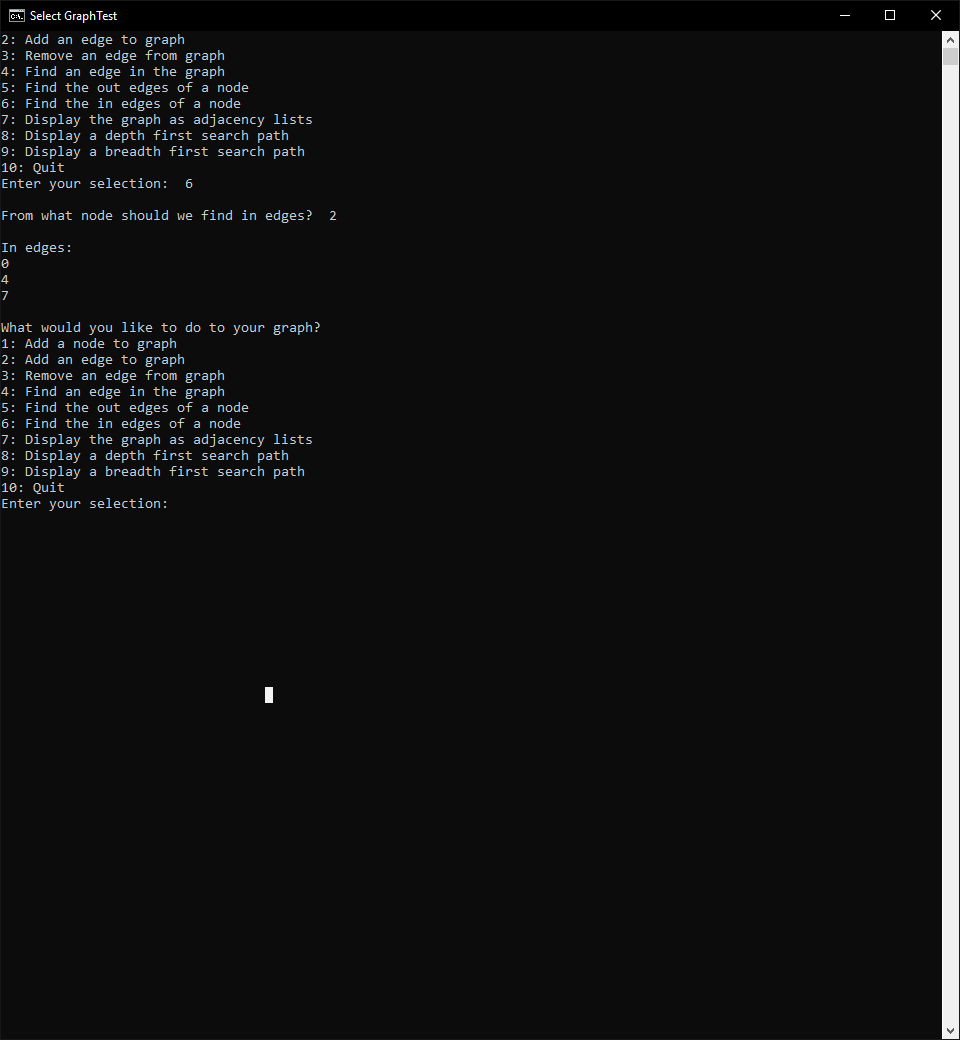


Figure 4 – Testing In Edges on Node 2

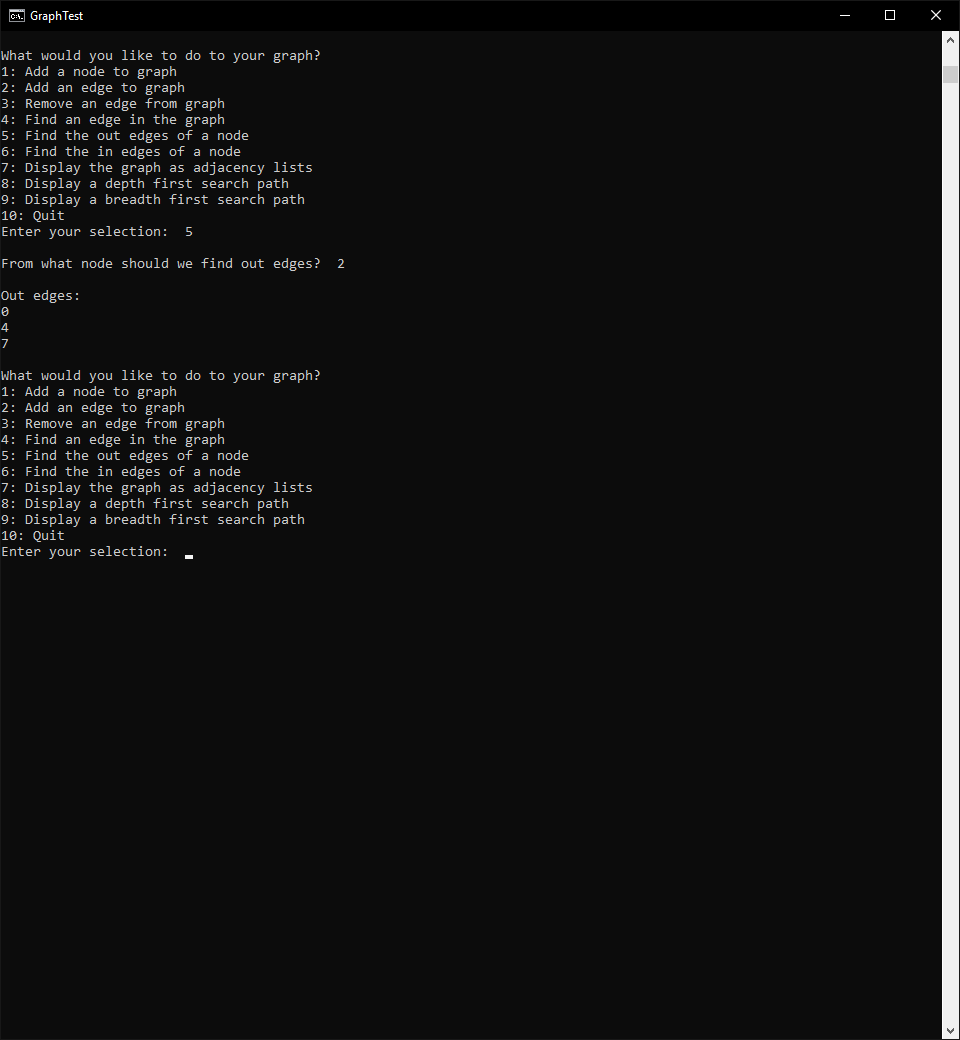


Figure 5 – Testing out Edges on Node 2

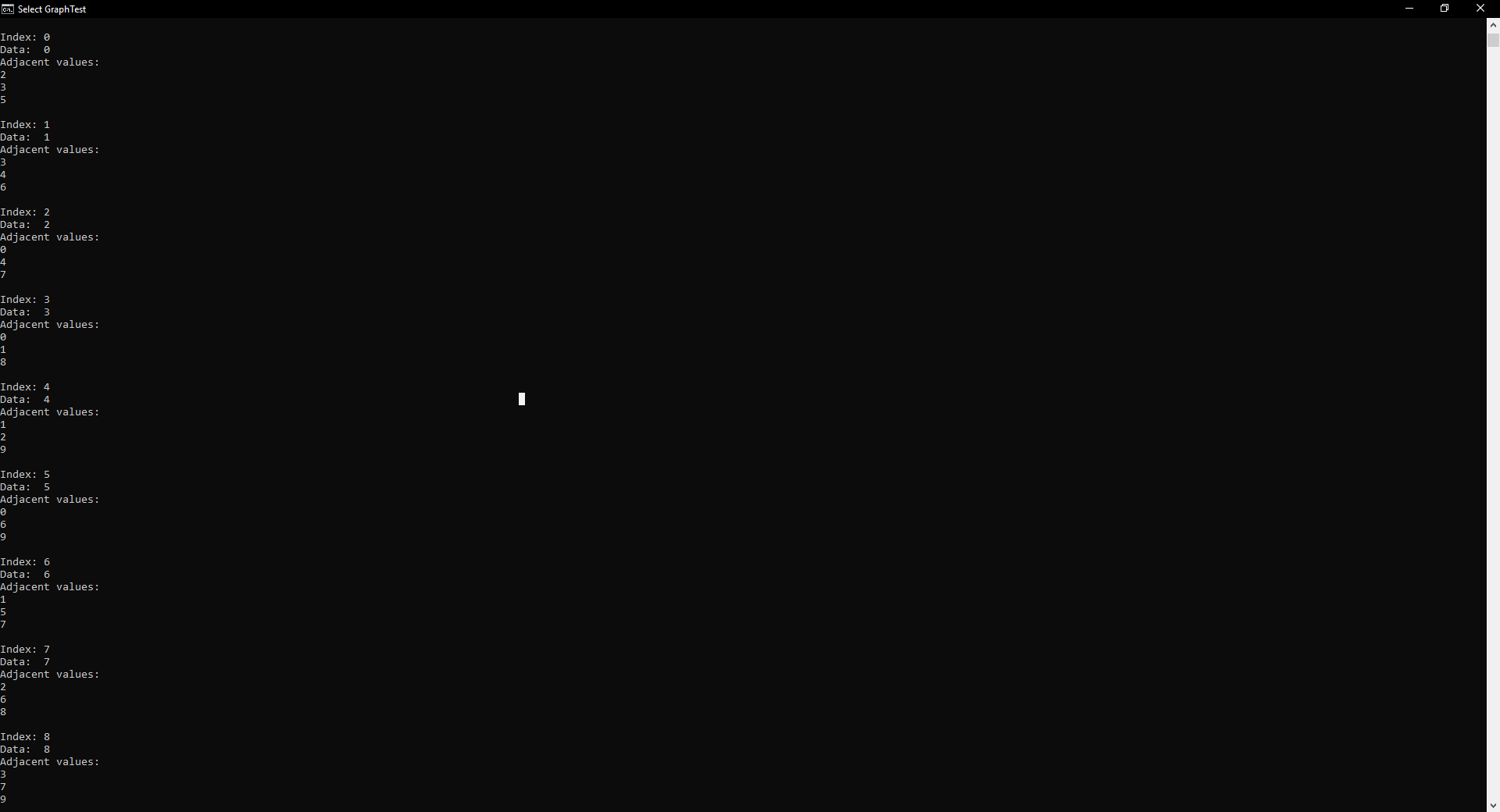
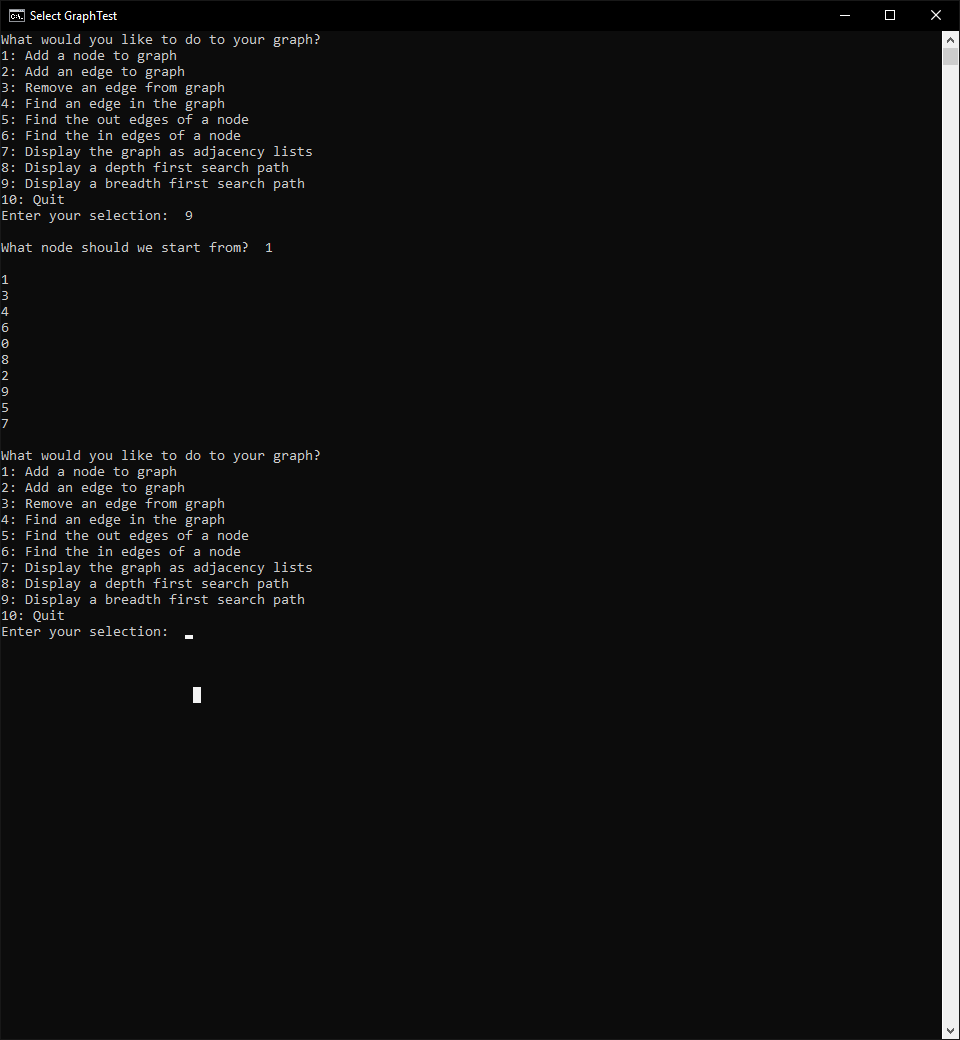
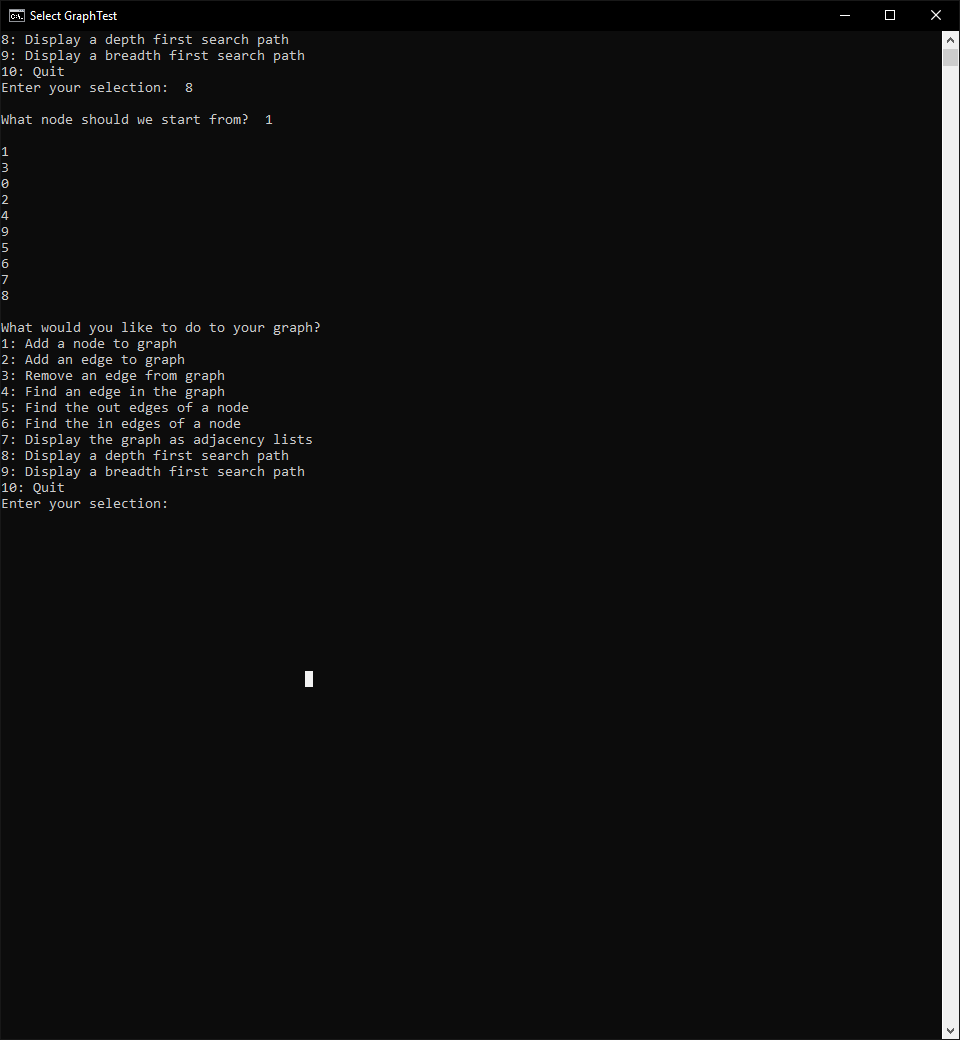


Figure 6 – Display Adjacency Lists

**Task 3.**

**** Figure 7 – Breadth First Search Starting at Node 1

**Task 4.**

******

*Figure 8 – Depth First Search Starting at Node 1*

**Task 5.**

The Breadth First Search was implemented using a vector and a queue. The queue was made to store the nodes that we have visited in the order they were visited in, and the vector was just used to mark if nodes have been visited. A queue was used to maintain the order of the visited nodes, and a vector was used for easy insertion and item detection. The search starts on the user given node. From there, in a while loop, it immediately pops off the first node, stores it in a temp variable and displays it. The search then looks at all edges from this temp node (equal to the user given node) and inserts it in the queue. The loop then repeats the while loop taking one node out of the queue at a time and displaying it while also branching off.

The depth First Search was implemented using 2 vectors. One vector was used to track nodes that have been visited and in what order, and the other was used to store the nodes we still have to search. In a while loop, the current node we are using to search is set to the back of the vector used for tracking which nodes to visit. The first iteration, this is the user provided node. In the loop, the search then finds the first node in the current nodes adjacency list that is not visited. That node is added to the back of the vector used for tracking nodes, and the loop repeats. If there was not a new node found in the adjacency list, the node is popped off the back of the vector, and the current node is now the new back. This search continues until the vector is empty.

There are different scenarios where one might use breadth first vs depth search. Breadth search will find the shortest path through a graph. However, it has high memory usage in doing this. Depth search will find a path through the graph faster than breadth search. It does this while using less RAM. Thus, if you don’t care about finding the shortest path, depth first searches should be used. For instance, you would usually want to do breadth search if the data you are using will be used multiple times. The extra resource usage is worth the resulting speed increase of further operations from knowing the shortest path. However, if you are parsing one-time-use data, it would be faster and smarter to just use depth search.

**Discussion.**

The primary concepts explored in this lab were graphs and their searching techniques. Graphs are a useful concept for mapping out data and their relations. This is very important for any data science application to further understand the connection between data. The searches are also important for figuring out the most efficient pathways between said data. If a company has sets of data that need to connect with other sets in some way, building a graph data structure could be wholly crucial to the data’s success, and having knowledge of how to use and modify this structure could be essential in the workplace.

**Compiling**

Lab11.cpp contains the main. It should be compiled with standard gpp settings for windows.

**Contributions**

Zach In edge, out edge, searches  
Dustin: Main and rest of Graph.cpp  
The work is equitable.