For our final project, we decided to use a dataset called Memetracker that tracks frequently-used phrases. For the purposes of this project, we took a sample of this data from the time frame of one day. Our goal was to collect all of the timestamps when any particular word or phrase was used and the URLs this was used in and build a network of AVL trees that comprised our graph. From this, we sought to implement a depth-first search, the Floyd-Warshall algorithm for shortest paths, and betweenness centrality.

The construction of our program begins with AVL trees. We utilized the AVL tree class provided in this course to build an AVL tree for each word or phrase. (We started the code off with a few words to demonstrate the functionality.) We implemented a searcher function that goes through the dataset to track all the instances when that particular word or phrase was used. These pairs of timestamps and URLs were then used as keys and values, respectively, for AVL tree vertices.

We had a bit of trouble collecting the timestamps, which were given to us in the form of strings. After trying several methods, we ultimately resolved this by converting them into comparable doubles. In addition, some of our data had an unproportional amount of URLs listed, making our stack space run out. We learned an important lesson about the power of pointers and references from this issue.

From there, we were able to turn our AVL tree into a graph. We connected our graph using a recursive function, which was built from the logic of the Graph class provided in this course. At first, we thought we wanted to make an undirected and unweighted graph. Part of the issue is that we couldn’t figure out a useful purpose for edge weights in the context of our program. Eventually, we decided to designate edge weights based on the difference between the timestamps of any two connected vertices,

thereby telling us how far apart in time these instances of the word or phrase were recorded. This was helpful because it allowed us to implement the Floyd-Warshall algorithm, which requires weighted graphs.

Our next step was to run algorithms on our graph. First, we implemented a depth-first search on our graph, starting with an arbitrary vertex, which was relatively straightforward. We did, however, run into a bit of trouble trying to implement Floyd-Warshall. Part of our problem was in how we were handling vertices that were unconnected in our graph. We noticed this when we started working on implementing betweenness centrality and we weren't getting reasonable values for our result. After lots of debugging and rethinking our logic, we finally arrived at what appears to be the correct answer. This allowed us to find the shortest paths between all the vertices (outputted as a map) and to accurately find the centrality of any vertex in our graph.

Ultimately, we were able to reach all of our expectations for this project. I think we all gained a lot from having to come up with our own program from scratch and work with a group of people to get there. At the very least, it makes future projects like these seem a little less intimidating.