The data structure that I used to implement the graph is a map from stl which is a red-black tree. Inside this map, for the key-value pair the key is an integer, and the value is a vector of integers. The integers represent unique websites and are equivalent to ids. Then, I used another stl map with key-value paris of string and int to convert each website to and integer id for the graph implementation. I used this conversion method to save on the total amount of memory used by the graph. Inside the program, only one copy for each string website is stored instead of multiple copies for each connection in the graph. Since strings usually are much larger than ints, this reduces the total memory used by the program especially if there are many connections in the graph. Finally, I used a vector to store the list of nodes because if needed, it can also store parallel edges unlike using an stl set. Originally, I had used a set but in some test cases with parallel edges it produced incorrect values because duplicate connections cannot be stored unless a multiset is used. Even though parallel edges are not being tested, I decided to switch to a vector to just reduce complexity and to ensure I get similar values to others.

Method analysis:

For the method analysis, when referring an stl container or algorithm complexity, I am referring to the complexity listed on cppreference.com and bigocheatsheet.com.

* Constructor- The constructor runs in O(1) time for the best, worst, and average cases because it is initializing a single integer which takes constant time.
* checkIfExists- For the worst case this function runs in O(q\*logn+q\*logn+q) which reduces to O(q\*logn+q), where n is the number of unique websites in that map that converts strings to ints and q is the size of the string being checked as it has to be copied into the map and compared to other strings while the tree is traversed. As more unique websites are added, n increases. For the best case, the complexity is O(q). This is different because in the best case, the website being looked at is already present in the map so the string does not have to be copied and the website is at the root of the node. For the average case, the complexity is O(q\*logn) because on average, the string that is being checked will be already in the map and will not need to be copied over. Every node must be connected to at least one other node, so on average the website being checked will be in the map.
* Insert- In the worst case the execution time is O(q\*logn+q) + O(p\*logn+p) from the checkIfExists function plus O(logm+1) where m is the number of nodes that point out to other nodes and n is the number of unique websites. The variables p and q are the lengths of the strings being checked for the from and to nodes. The O(logm+1) comes from the lookup time from the adjacencyStorage which uses a map and is logm, and the constant O(1) comes from the time complexity of push\_back for the vector listed on cppreference. This then reduces down to O((q+p)\*logn+p+q+logm) which then becomes O((q+p)\*log(n)+log(m)) from factoring out the q+p and simplifying the big O notation. The finally, the q+p multiplication comes from comparing each string as the tree is traversed. In the average case, the insert function is O(q\*logn+p\*logn+logm+1) which becomes O((q+p)\*log(n)+log(m)) and makes the average case the same as the worst case in complexity. The O(q\*logn) and O(p\*logn) are the average case listed above for checkIfExists and all the variables still have the same definitions from previous methods and cases. In the average case, the from and to nodes are already in the string conversion map so they do not needed to be copied. In the best case, the function is O(q) and for this to occur, the graph connection must be a self-loop because the strings for both must be the same. Along with this, the string is already contained in the string conversion tree so no copies will be done, and the node is also the root of the adjacencyStorage map to be O(q) time. Furthermore, the push\_back method is also O(1) leading to the final O(q) best case time complexity. The O(q) comes from comparing the strings in the checkIfExists function.
* PageRank- In the worst and average case, the page rank algorithm I made is O(nlogn+(p\*(n^2+m\*(logn+r\*logn)))+nlogn) which reduces to O(nlogn+(p\*(n^2+m\*(r\*logn)))). In this, n is the number of unique nodes, p is the number of power iterations, m is the number of nodes that have outgoing connections, and r is the number of outgoing connections for each node in m nodes. In the non-reduced big O notation, the first nlogn is calculating the page rank when the number of power iterations is 1. Even though the iterations map that stores the values is not initially of size n, as it fills up it does so at a rate that the total number of operations is proportional to nlogn. The second nlogn at the end is for printing out the resulting page ranks but nlogn+nlogn just reduces to nlogn. Then, inside the loop the logn is for accessing the value from the previous power iteration and the r\*logn is for doing a partial calculating of the current power iteration, but also reduces to just r\*logn. In the best case, the number of power iterations is just one so the complexity is O(nlogn) since the second for loop to calculate p=2, … , inf never even runs and is just skipped.
* Main- In the worst case and average, main is O(L\*((q+s)\*log(n)+log(m))+ nlogn+(p\*(n^2+m\*(r\*logn)))) where L is the number of lines read, q is the length of the from string, s is the length of the to string, n is the number of unique websites, and m is the number of nodes that point to other nodes. Then, p is the number of power iterations and r is the number of outgoing connections for each m nodes. In this scenario, in the worst and average case the insert functions and page rank functions have the same complexity. In the best case when only 1 power iteration is done, the complexity becomes O(L\*((q+s)\*log(n)+log(m))+nlogn) because only the for loop that generates the initial power iteration runs.

From this project, I learned