2019

DSA Assignment

DOCUMENTATION & REPORT

GEORGE AZIZ - 19765453

DOCUMENTATION:

An overview of your code, explaining any questions that the marker may have. This is supplemented by the comments in your code. In general, if the marker is looking at your code and isn't sure why you have done something in that particular way, they will check your documentation to see if they can find an explanation.

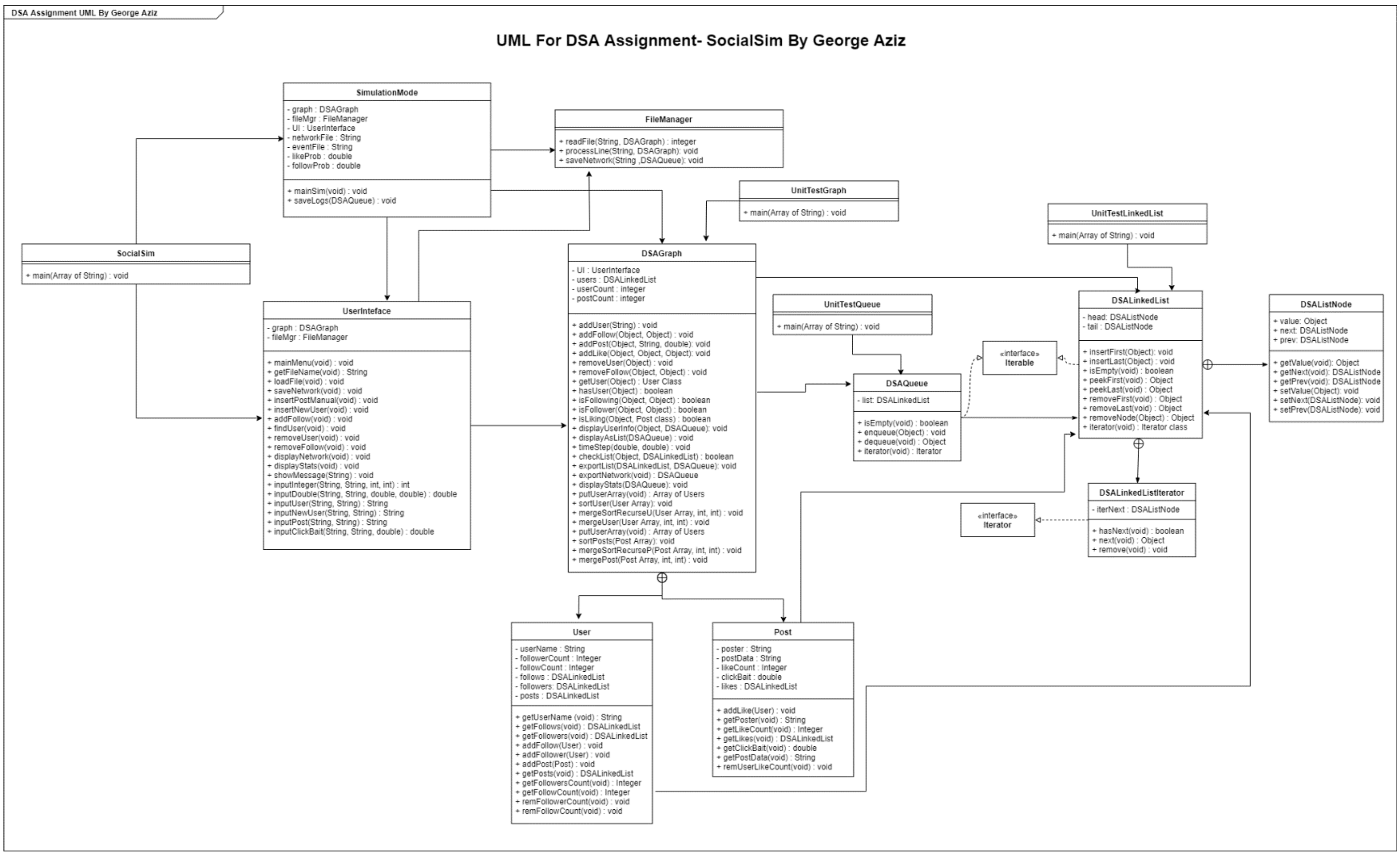
One of the classes in my program is DSAGraph. The way DSAGraph is structured is to have a linked list of users and then within each user which is a private inner class, to contain three more linked lists. The three linked lists are follows, followers, and posts. The linked list containing posts contains posts of Post type. A Post is another private inner class in my DSAGraph and contains one final linked list for likes which has the users that liked the post. There are more class fields in each of these private inner classes but in terms of the data structures that are used within them, linked lists are used.

The time step feature is very simple from the way I understood it. An example would be having three people. The first person follows the second person and the second person follows the third person. If the second person likes a post of the third person then there is a chance for the first person to follow the third person. (A -> B -> C). Time step shares a transitive relation where if person A follows person B and person B follows person C then there is eventually a chance for person A to start following person C. Each post also contains click bait factor that acts as a multiplier for the liking probability. The liking and following are based off probabilities that are provided by the user for interactive mode or as command line arguments for the simulation mode.

The DSAQueue class implements a queue which is an ADT that I use for file output and also displaying to the user. Rather than printing out with System.out.print from DSAGraph for all the methods that required displaying such as displayStats or displayAsList, or even exportList, they all rather than printing, enqueue into the queue. The reason I do this is to be able to use the same methods like displayStats for file output rather than having a different method or add extra code since all I have to do now is to import a queue to the the writeFile method in FileManager which will print out to a file just as formatted in the queue from DSAGraph.

One thing I have done for when displaying user information or when displaying statistics of the network is added for each post the people who liked the post. I have done this to resemble a truer social network platform such as Facebook for example. I understand it might look messy especially with a lot of users, but it is also useful for when testing and good to know who liked what post as well.

A UML Class diagram of your code



DSA Assignment UML By George Aziz
UML For DSA Assignment- SocialSim By George Aziz
SimulationMode
- graph : DSAGraph
- fileMgr : FileManager
- UI : UserInterface
- networkFile : String
- eventFile : String
- likeProb : double
- followProb : double
+ mainSim(void) : void
+ saveLogs(DSAQueue) : void
FileManager
+ readFile(String, DSAGraph) : integer
+ processLine(String, DSAGraph): void
+ saveNetwork(String ,DSAQueue): void
UnitTestGraph
+ main(Array of String) : void
UnitTestLinkedList
+ main(Array of String) : void
SocialSim
+ main(Array of String) : void
DSAGraph
- UI : UserInterface
- users : DSALinkedList
- userCount : integer
- postCount : integer
+ addUser(String) : void
+ addFollow(Object, Object) : void
+ addPost(Object, String, double): void
+ addLike(Object, Object, Object): void
+ removeUser(Object) : void
+ removeFollow(Object, Object) : void
+ getUser(Object) : User Class
+ hasUser(Object) : boolean
+ isFollowing(Object, Object) : boolean
+ isFollower(Object, Object) : boolean
+ isLiking(Object, Post class) : boolean
+ displayUserInfo(Object, DSAQueue): void
+ displayAsList(DSAQueue) : void
+ timeStep(double, double) : void
+ checkList(Object, DSALinkedList) : boolean
+ exportList(DSALinkedList, DSAQueue): void
+ exportNetwork(void) : DSAQueue
+ displayStats(DSAQueue): void
+ putUserArray(void) : Array of Users
+ sortUser(User Array): void
+ mergeSortRecurseU(User Array, int, int) : void
+ mergeUser(User Array, int, int) : void
+ putUserArray(void) : Array of Users
+ sortPosts(Post Array): void
+ mergeSortRecurseP(Post Array, int, int) : void
+ mergePost(Post Array, int, int) : void
UserInteface
- graph : DSAGraph
- fileMgr : FileManager
+ mainMenu(void) : void
+ getFileName(void) : String
+ loadFile(void) : void
+ saveNetwork(void) : void
+ insertPostManual(void) : void
+ insertNewUser(void) : void
+ addFollow(void) : void
+ findUser(void) : void
+ removeUser(void) : void
+ removeFollow(void) : void
+ displayNetwork(void) : void
+ displayStats(void) : void
+ showMessage(String) : void
+ inputInteger(String, String, int, int) : int
+ inputDouble(String, String, double, double) : double
+ inputUser(String, String) : String
+ inputNewUser(String, String) : String
+ inputPost(String, String) : String
+ inputClickBait(String, String, double) : double
UnitTestQueue
+ main(Array of String) : void
DSAQueue
- list: DSALinkedList
+ isEmpty(void) : boolean
+ enqueue(Object) : void
+ dequeue(void) : Object
+ iterator(void) : Iterator
«interface»
Iterable
DSALinkedList
- head: DSAListNode
- tail : DSAListNode
+ insertFirst(Object): void
+ insertLast(Object) : void
+ isEmpty(void) : boolean
+ peekFirst(void) : Object
+ peekLast(void) : Object
+ removeFirst(void) : Object
+ removeLast(void) : Object
+ removeNode(Object) : Object
+ iterator(void) : Iterator class
DSAListNode
+ value: Object
+ next: DSAListNode
+ prev: DSAListNode
+ getValue(void): Object
+ getNext(void): DSAListNode
+ getPrev(void): DSAListNode
+ setValue(Object): void
+ setNext(DSAListNode): void
+ setPrev(DSAListNode): void
«interface»
Iterator
DSALinkedListIterator
- iterNext : DSAListNode
+ hasNext(void) : boolean
+ next(void) : Object
+ remove(void) : void
User
- userName : String
- followerCount : Integer
- followCount : Integer
- follows : DSALinkedList
- followers: DSALinkedList
- posts : DSALinkedList
+ getUserName (void) : String
+ getFollows(void) : DSALinkedList
+ getFollowers(void) : DSALinkedList
+ addFollow(User) : void
+ addFollower(User) : void
+ addPost(Post) : void
+ getPosts(void) : DSALinkedList
+ getFollowersCount(void) : Integer
+ getFollowCount(void) : Integer
+ remFollowerCount(void) : void
+ remFollowCount(void) : void
Post
- poster : String
- postData : String
- likeCount : Integer
- clickBait : double
- likes : DSALinkedList
+ addLike(User) : void
+ getPoster(void) : String
+ getLikeCount(void) : Integer
+ getLikes(void) : DSALinkedList
+ getClickBait(void) : double
+ getPostData(void) : String
+ remUserLikeCount(void) : void

The UML is also provided in the same directory as the program and this documentation as a PNG image.

A description of any classes you have, you need to let us know not only what the purpose of that class is but why you chose to create it. As part of this, also identify and justify any places where it was possibly useful to create a new class but you chose not to, especially when it comes to inheritance.

Classes in my program including their private inner classes:

1. SocialSim (Class with my main)
2. DSAGraph
   a. User
   b. Post
3. DSALinkedList
   a. DSAListNode
4. DSAQueue
5. UserInterface
6. FileManager

7. SimulationMode

Each ADT/ Data structure contains a test harness as well to make it easier to see if all functions are working.

The **socialSim** class is the class that contains my main. It is the class that you run the program with and calls either UserInterface or SimulationMode depending on the mode the user wants to run the program with.

The **DSAGraph** class could be regarded as the main file as it contains all the main network/graph functionalities. Anything such as adding or removing a user, post, like or a follow is within the DSAGraph class. In addition to adding and removing, the DSAGraph class also contains anything related to displaying the network such as an adjacency list or as exporting as a network file to be saved later. The class itself does not actually display through System.out.print but uses DSAQueue to enqueue anything that I would want to be displayed and then the actual display happens in UserInterface by iterating over the queue. This class also contains sorting the users and posts in terms of popularity as it is required in the assignment specification sheet for the display of statistics. The reason I created a DSAGraph class is to be able to have a social network and connect users with each other.

Within the DSAGraph class, there are two private inner classes. These classes are **User** and **Post**. For User, it was originally a vertex, but I decided to change the name to make it easier to understand the code but also to make it have more sense for the class to contain the class fields that have been included. The user class is responsible for everything related to the user which is to have a user name, followers and following list, a list of posts and a count for how much follows and followers. Each of these class fields have accessors and mutators such as addFollow for example. My second private inner class is Post. I decided to create a Post class quite late into the assignment, but it has helped me a lot and made my code a lot shorter and easier to understand. This class is responsible for everything related to a post such as the post itself, a list of likes and a like count, and in addition the name of the poster. This class is very important since makes the adding of likes and removal of likes much easier but also is stored in each User's posts linked list which makes accessing a post very easy.

The **DSALinkedList** class is very important for this program as most of my private class fields in either DSAGraph, User or Post utilise this class a lot. I used it since I am most familiar with this abstract data type/structure but also since it makes the most sense to me to use a linked list within a graph. Any adding or removing requires the DSALinkedList class in my program since that's the main structure that is used.

DSAQueue was a class I added towards the end when I realised for my writeFile method in FileManager it will have to be able to write in many formats depending on my usage. For example, exportNetwork in DSAGraph is a method that takes in the whole graph as a network file format and then that gets used for when I want to save my network. But in simulation mode I also need to write to file but in a format as an adjacency list and the statistics. In order to overcome this problem, I decided to use a queue where everything that will be displayed either to the terminal for the user or to a file will be enqueued in the format that is set to and then can be used for both displaying to the user or file output.

The **UserInterface** class is responsible for displaying to the user a main menu and also all the inputs for everything such as follows or a user. This class is also responsible for validating user input whether that's a user or an integer or even a post. It is very important to have this class for my program since the mainMenu submodule inside of it connects all the functionality of DSAGraph together as well as FileManager.

The **FileManager** class is self-explanatory with its name but basically manages reading and outputting from/to a file. It is able to read both event and network files and also output a graph in the same format as a network file or output statistics of the network as well.

SimulationMode is a class that is responsible to run the sim mode of the program. It runs reads both event and networkfiles and then goes through a loop of time steps until the user stops the loop. Just like the UserInterface class is mainly for interactive mode, this class is mainly for sim mode. I decided to create this class since it made my main a lot cleaner. Rather than having everything for sim mode in main which is possible and would've worked it makes more sense to have everything for the simulation mode to be in one class and for it to be called in one line in the main.

It was possible for me to create a new class called Network and shorten the DSAGraph class by a little bit but I didn't really see a point since all the methods that are in DSAGraph currently relate to the graph's functionality or display of it but also if I did create a Network class, then I would've had to make my User and Post class public and a separate class to DSAGraph which doesn't really fit my design as I like to keep everything for a graph, inside of DSAGraph.

There was also another approach to the program that I could have taken which would have involved inheritance and making a vertex super class and that contains either a user or post which will inherit a lot of what the vertex class has. Although this seems like a good approach, I believe that my approach is much simpler and easier to understand.

<u>Justification of all major decisions made. In particular, when you choose an ADT, underlying data structure or an algorithm, you need to justify why you chose that one and not one of the alternatives. These decisions are going to be of extreme importance in this assignment.</u>

For the data structures, I used DSAGraph since that is required of us from the assignment specification, but also because it is the data structure that is used to create the social network itself. Without the graph data structure, there is no social network.

Another data structure that I use is a linked list which is in the DSALinkedList class and I've used it since it is much easier to iterate over it in comparison to other data structures such as Hash Tables and it is much easier to use since I've used it many times before. The time complexity of a LinkedList at its worst is of O(N) whilst the best case is either when the node is in the head or tail and that is O(1) which is sufficient for a social network.

The ADT I use is a queue. The queue is in the DSAQueue class and is used mainly for outputting to a file or to the user since many outputs to files or to the user are required in different formats and the easiest way is to keep the format in a queue and then iterate over it when its time for output. I chose a queue over a stack since a stack follows the FILO structure which isn't useful for when displaying unless I push everything in reverse order which is not ideal and makes the program much more complicated than needed.

REPORT:

Abstract: Explain the purpose of the report and state what you are investigating, and the outcomes/recommendations.

The purpose of this report is to explore the time complexities of multiple data structures including a linked list and a graph since those are the main data structures that have been used for when the SocialSim program has been created which will help with the investigation of the time complexities. In addition, this report will also make comparisons to different data structures that could've been used with the graph which could possibly have been better than my current implementation for the social network and provide explanations as to why the structures that were used are better or less efficient than others. This report will also compare multiple sorting methods with each other and why the sorting algorithm I have implemented is better than others in terms of time complexity. The outcome of this report will help provide a better understanding on data structures and which data structure is better with scenarios such as a social network and help identify which sorts method to use depending on the sample size that will be used. This report will also help provide an explanation of how I came to the final design of the social network simulation program.

Background: Discuss your approach to developing the simulation code, and the aspects of the simulation that you will be investigating.

One of the main functionalities of the program that will help with this investigation is the time-step option or the update/refresh feature. The timestep is a concept of stepping through one step of time, which for me I understood it as providing the possibility of liking a post and then allowing the followers of the post liker to follow the poster. Once a user has liked a post, then their followers will be able to see the poster and have a chance to follow them and then maybe even like the post. This process will radiate out from the initial poster. The first time-step will commence after the network and event file have been processed and successfully loaded for simulation mode whilst in interactive the user loads in the network file and then manually create more users, follows and posts to resemble how a true social network would work. For most of the test files provided such as the toy story network and event files, everything is complete and all posts are propagated after one time step with a probability of 1 for both liking and following since with the way I have programmed time-step it iterates over each person and goes through their neighbours and follows and posts. The time step concept is based off a transitive relation where if person A was to follow person B and person B was to follow person C, and person B likes a post of person C, then person A will have a chance to follow person C (A->B->C). The total amount of loops and/or iterations that I use are four. Two iterators to iterate over the users and then one for the posts and within that another for the likes of each post.

Another aspect that will be explored is the sorting algorithms. The algorithm that I have picked for sorting the social network for when statistics of the network needs to be displayed is merge sort. The reason for merge sort is that it performs the best for large data sample sizes and with a social network, it must be able to handle large networks. Merge sort is not an in-place sort since there needs to be the temp array for merging and that can't be rid without slowing the algorithm down. Unfortunately this uses more memory than other sorting algorithms such as insertion sort but merge sort is a stable sort which is good since it doesn't have to sort elements/nodes that are equal. Merge sort is also very consistent with fast speeds with a time complexity of O (N log N) most of the time. While there are other types of sorting algorithms such as quick sort for example that also has a best and average case of O(N log N), the worst case is O(N^2) and nothing is as consistent as merge sort. There is also insertion sort which has a time complexity of O(N) which is faster than merge sort but that's for the best case only whilst the other cases it is O(N^2) which is slower than merge sort. Merge sort is more consistent and faster on average and worst cases in comparison to insertion sort.

The simulation code itself is mainly located in the DSAGraph class as stated before. The way the class is structured is explained above in the documentation but to summarise, the main data structure that is used in the graph are linked lists. There are multiple data structures to choose from when creating a social network such as the one I created, but I decided to go with a Linked List mainly because it is simple but also because of its time complexity is good and consistent. The linked list data structure is not the best data structure for a social network, but it performs well. One data structure for DSAGraph that I had in mind was Hash Tables.

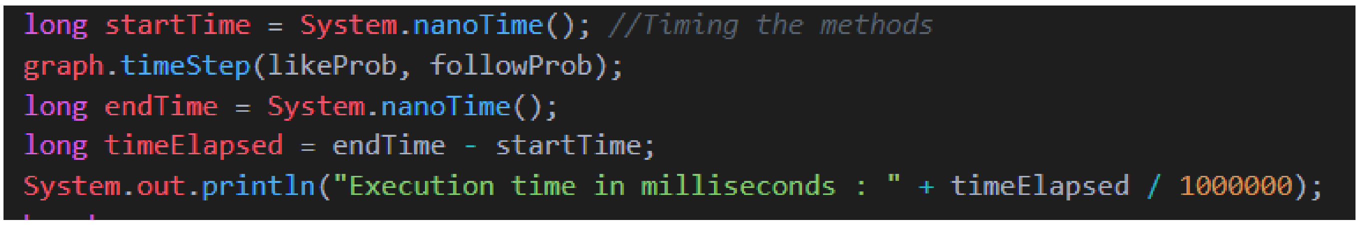
Hash Tables are great because of their time complexity of O(1) or O(K) for almost everything but when implementing a Hash Table I ran into issues with iterating over my users for example and also with the removal of likes from posts since a post is within a user and then a like is inside a post which was too complicated. I also ran into issues where the only way to work with a Hash Table was to have my User and Post class outside and as public which didn't fit the design I wanted but also would have went against the graph data structure since a user in my DSAGraph is just a vertex with added class fields and a different name and it isn't good to have the vertex an outer public class since its only meant to be used by DSAGraph. One disadvantage of a hash table that a linked list performs better in is memory usage with the hash table always requiring an array to be under a load factor of around 60% which takes up more space and also resizing which will happen frequently with a large social network having users getting added constantly.

With the linked lists in DSAGraph, a method that I had trouble with was mainly the removeNode submodule since it was very different to the other remove methods such as removeFirst or removeLast and that to get to a node, a traversal of the linked list was needed until the node is found, all this while maintaining a good time complexity. The way the method works is it checks for both tail and head first before traversing to find the node to ensure that there are two best cases rather than just one.

Another data structure that I also thought about was Binary Search Trees as they are self-sorting, but the only issue is with a binary search tree, it is very difficult to maintain its balance, and so degenerate trees and un-balanced trees will be a possibility and in that sense they will become the same as linked lists in terms of time complexity or just a little bit better in some cases.

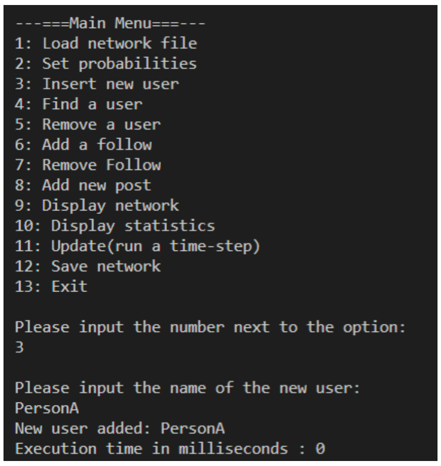
Methodology: Describe how you have chosen to profile and compare multiple runs of the simulation, and why. You should give some prediction of the expected "performance" of the aspects of your code you are investigating – this includes time complexity and/or memory usage. Include the commands, input files, outputs – anything needed to reproduce your results.

The way of testing will be by using the concurrent library in Java as the following screenshot and also will be testing each test case three times and taking the average time of those and putting them into a table format in a .txt.



```
long startTime = System.nanoTime(); //Timing the methods
graph.timeStep(likeProb, followProb);
long endTime = System.nanoTime();
long timeElapsed = endTime - startTime;
System.out.println("Execution time in milliseconds : " + timeElapsed / 1000000);
```

All tests will happen in the interactive mode of the SocialSim program and the times will be in the UserInterface class. To start the interactive mode you must use the command: ./SocialSim -i and to be able to use the same tests as I have done if needed, you will need to go into UserInterface and uncomment the time tests for each method that gets called in the main menu. To test the times run one of the menu options and the time elapsed will be outputted to the terminal.



```
---===Main Menu===---
1: Load network file
2: Set probabilities
3: Insert new user
4: Find a user
5: Remove a user
6: Add a follow
7: Remove Follow
8: Add new post
9: Display network
10: Display statistics
11: Update(run a time-step)
12: Save network
13: Exit

Please input the number next to the option:
3

Please input the name of the new user:
PersonA
New user added: PersonA
Execution time in milliseconds : 0
```

The actual timing of the test will commence before the call to the DSAGraph method just as shown above in the first screenshot. This includes all displays, adding, removing and time step. I won't be testing the save network method as it doesn't help with the understanding of the performance of the program. All the methods that are tested are from the menu selection in the interactive mode of the SocialSim program as shown to the right. The number of tests that I will be doing will be a total of three with the first having a network file of 100 users, second having 1000 users and third having 10,000 users. The follows will be random, and the network files are created in Microsoft Excel and then exported to a .txt file. The test samples will be in the same directory as the program and this report.

I am expecting the times for my time step to take longer as the size of the sample increases since the time complexity of my time-step for each iteration would be O(N) which is just the traversal of the lists. But for the overall method it would be O(N^2) + O(N)*2 since 2 of the iterators are iterating over the number of users but then there are 2 more for the posts and likes of each post. The best case scenario would be O(1) for each iterator if there is only 1 iteration by having just one user but that is highly unlikely in a social network especially with my tests where I will be testing with 100 users at the minimum.

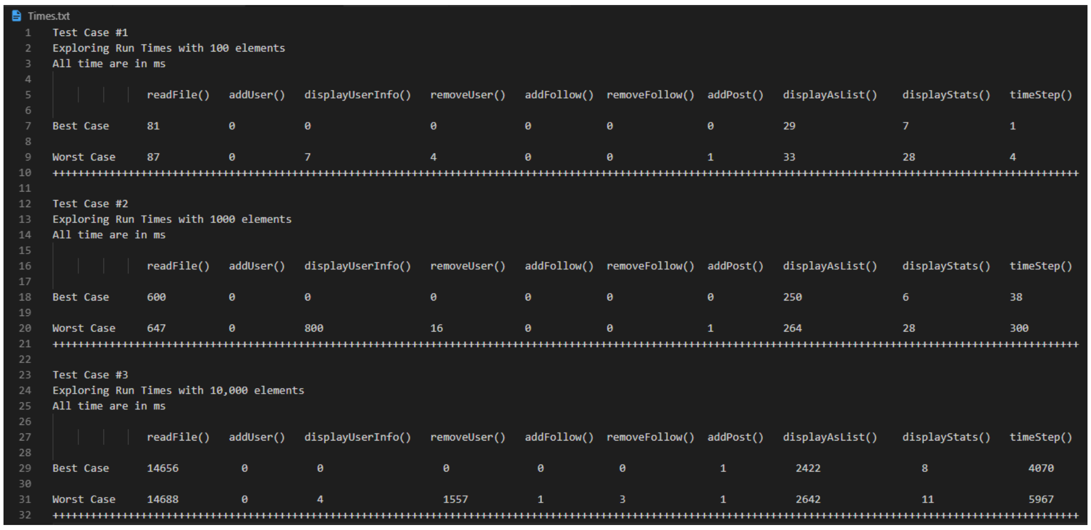
As stated above the DSAGraph class uses mostly linked lists. Note that all my linked lists are double ended, doubly linked which helps with better time complexities for accessing the tail node and more. At the best case the time complexity of a double ended, doubly linked list with the Big O Notation would be of O(1) where the node or object is at the head or tail whilst the worst case would be O(N) where it will have to traverse / iterate over the list to find the node. For methods such as adding I am expecting instant times like 0ms for example since every new user or follow or like will be added to the end of the list with the insert last method from the linked list class. The time complexity of any add will be of O(1) since it accesses the tail and adds a new node after the tail. I am also expecting the display methods to be on average speeds since they will follow the O(N) time complexity and so the times should scale depending on how big the social network is. Another method I am expecting to perform well is removeUser, since there are two best cases of O(1) time complexity and the average and worst being of O(N).

Within the DSAGraph class, there is a method that displays the statistics of the network at its current state. For this method I am expecting it to perform quite well since I am using merge sort which should be able to be consistent with its times depending on the scale of the sample size. I am expecting the merge sort to maybe not perform as well on the third test case as it must sort 10,000 users which should be a lengthy process in comparison to the other two test cases.

Results: Present the results of your simulations, and what you discovered.

The results can be found in the same directory as the program and this report which is named as Times.txt. All these results were tested on my home Windows PC which has an i7 processor with 16GB Ram. The times on your machine may differ depending on the hardware of your machine.

An image of the results can be seen below:



Times.txt

```
Test Case #1
Exploring Run Times with 100 elements
All time are in ms

                readFile()   addUser()   displayUserInfo()   removeUser()   addFollow()  removeFollow()  addPost()   displayAsList()    displayStats()   timeStep()

Best Case       81           0           0                   0              0            0               0           29                 7                1

Worst Case      87           0           7                   4              0            0               1           33                 28               4
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Test Case #2
Exploring Run Times with 1000 elements
All time are in ms

                readFile()   addUser()   displayUserInfo()   removeUser()   addFollow()  removeFollow()  addPost()   displayAsList()    displayStats()   timeStep()

Best Case       600          0           0                   0              0            0               0           250                6                38

Worst Case      647          0           800                 16             0            0               1           264                28               300
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Test Case #3
Exploring Run Times with 10,000 elements
All time are in ms

                readFile()   addUser()   displayUserInfo()   removeUser()   addFollow()  removeFollow()  addPost()   displayAsList()    displayStats()   timeStep()

Best Case       14656          0           0                   0              0            0               1           2422               8                4070

Worst Case      14688          0           4                   1557           1            3               1           2642               11               5967
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

The results are as expected for most of the method with the more users or elements in the social network, the longer most of the methods will take. However, there were some unexpected results such as displaying the stats on the third test case. Surprisingly it was almost as fast as the first and second test cases but what was even more surprising was the worst case was faster on the third than the rest. I was surprised by this since there are 10,000 users in the network. Most of the work for display stats is with merge sort which performs well as it seems from these tests.

Most of the methods followed the time complexities such as addUser since that method adds the new user to the end of the linked list every time which is of O(1) time complexity. The removeUser method also performed quite well with the best case

The time step method performed poorly with the third test case since there are a lot more users in the third case than the other two. The difference is quite significant but it should be expected when the data structure being used is a linked list and with the way my time step functions, it uses four iterators which will be of a time complexity similar to O(N^4) even though two of the iterators iterate over different lists.

<u>Conclusion and Future Work: Give your conclusions and what further investigations could follow. Do the actual results match your prediction?</u>

After testing the performance of my program it is clear that linked lists are good for social networks but when comparing it to something such as a hash table, it will most likely perform worse since the time complexity of a hash table is O(1) whilst a linked list has O(N) unless the node is at the head or the tail. Linked lists are simple to understand but also consistent and so is always a sufficient choice. The results that have been shown above, were as expected for most of the methods. One method that I expected to perform poorly was time step but in the test results it is clear that the time step method showed a even more poor performance than expected but is still understandable as it uses multiple iterators. The merge sort that is used within the statistics display method used merge sort which is clearly a very good option since it is consistent but showed even better performance than expected and for the final test case it performed better than the other two as well.