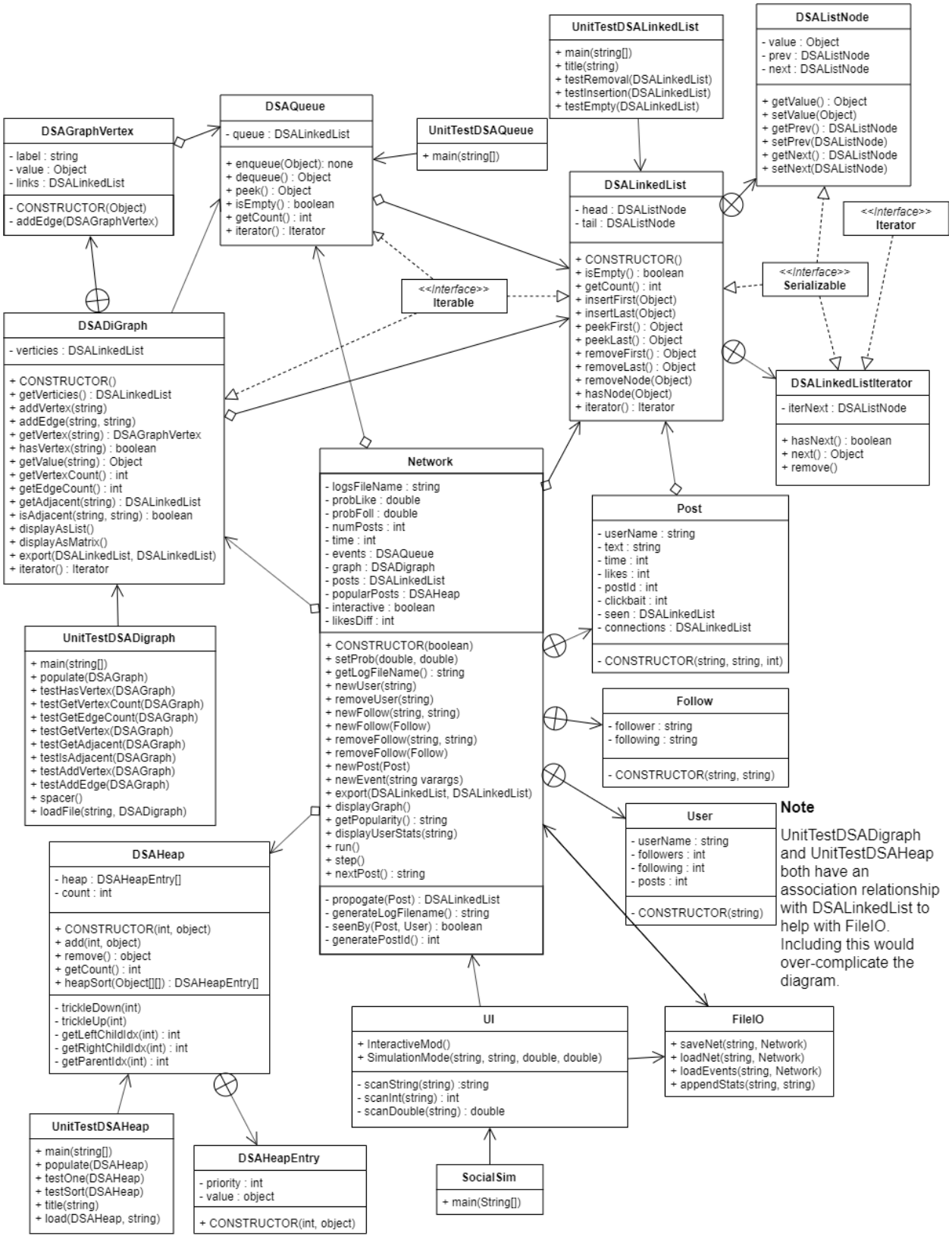
Overview

The program uses a directed graph to represent the connections between users in the social network. A link from user 1 to user 2 means that user 2 is following user 1. All events are stored in a queue and all active posts are stored in a linked list. This queue contains multiple different types of objects representing users, follows and posts. If a user/follow already exists, that event represents the removal of that user/follow from the network.

In simulation mode, each timestep, if there are no active posts, the events queue is dequeued, activating events until a post is come across and made active. If in a timestep there is an active post, the post travels one ‘level’ out in that timestep. Each post has a list of users, called ‘connections’, when a post is made active, the poster’s followers are added to this list. For each timestep, each of these connected users are iterated through and it is tested if they like the post. If so, all of that user’s followers are connected to the post for the next timestep. If they also follow, the user follows the original poster. Each timestep this connections list is worked through until it is empty, although in interactive mode, a post is made active as soon as it is manually added and timesteps are manually controlled. At the end of a post’s life, it is removed from storage and the amount of real-world time it spent propagating is saved to the log file.

During a timestep, each time a like, follow, new user, user removal, follow, unfollow, new post, or post view happens, the log file is updated accordingly. This does not include the loaded initial network state, as this is done before any timestep. Note that in comments and variable names throughout the program, ‘stats’ and ‘logs’ are used interchangeably, generally, it is most appropriate to think of them as logs.

UML


UnitTestDSALinkedList
+ main(string[])
+ title(string)
+ testRemoval(DSALinkedList)
+ testInsertion(DSALinkedList)
+ testEmpty(DSALinkedList)
DSAListNode
- value : Object
- prev : DSAListNode
- next : DSAListNode
+ getValue() : Object
+ setValue(Object)
+ getPrev() : DSAListNode
+ setPrev(DSAListNode)
+ getNext() : DSAListNode
+ setNext(DSAListNode)
DSAQueue
- queue : DSALinkedList
+ enqueue(Object): none
+ dequeue() : Object
+ peek() : Object
+ isEmpty() : boolean
+ getCount() : int
+ iterator() : Iterator
UnitTestDSAQueue
+ main(string[])
DSAGraphVertex
- label : string
- value : Object
- links : DSALinkedList
- CONSTRUCTOR(Object)
- addEdge(DSAGraphVertex)
DSALinkedList
- head : DSAListNode
- tail : DSAListNode
+ CONSTRUCTOR()
+ isEmpty() : boolean
+ getCount() : int
+ insertFirst(Object)
+ insertLast(Object)
+ peekFirst() : Object
+ peekLast() : Object
+ removeFirst() : Object
+ removeLast() : Object
+ removeNode(Object)
+ hasNode(Object)
+ iterator() : Iterator
<<Interface>>
Iterator
<<Interface>>
Serializable
<<Interface>>
Iterable
DSADiGraph
- verticies : DSALinkedList
+ CONSTRUCTOR()
+ getVerticies() : DSALinkedList
+ addVertex(string)
+ addEdge(string, string)
+ getVertex(string) : DSAGraphVertex
+ hasVertex(string) : boolean
+ getValue(string) : Object
+ getVertexCount() : int
+ getEdgeCount() : int
+ getAdjacent(string) : DSALinkedList
+ isAdjacent(string, string) : boolean
+ displayAsList()
+ displayAsMatrix()
+ export(DSALinkedList, DSALinkedList)
+ iterator() : Iterator
DSALinkedListIterator
- iterNext : DSAListNode
+ hasNext() : boolean
+ next() : Object
+ remove()
Network
- logsFileName : string
- probLike : double
- probFoll : double
- numPosts : int
- time : int
- events : DSAQueue
- graph : DSADigraph
- posts : DSALinkedList
- popularPosts : DSAHeap
- interactive : boolean
- likesDiff : int
+ CONSTRUCTOR(boolean)
+ setProb(double, double)
+ getLogFileName() : string
+ newUser(string)
+ removeUser(string)
+ newFollow(string, string)
+ newFollow(Follow)
+ removeFollow(string, string)
+ removeFollow(Follow)
+ newPost(Post)
+ newEvent(string varargs)
+ export(DSALinkedList, DSALinkedList)
+ displayGraph()
+ getPopularity() : string
+ displayUserStats(string)
+ run()
+ step()
+ nextPost() : string
- propogate(Post) : DSALinkedList
- generateLogFilename() : string
- seenBy(Post, User) : boolean
- generatePostId() : int
Post
- userName : string
- text : string
- time : int
- likes : int
- postId : int
- clickbait : int
- seen : DSALinkedList
- connections : DSALinkedList
- CONSTRUCTOR(string, string, int)
Follow
- follower : string
- following : string
- CONSTRUCTOR(string, string)
User
- userName : string
- followers : int
- following : int
- posts : int
- CONSTRUCTOR(string)
Note
UnitTestDSADigraph and UnitTestDSAHeap both have an association relationship with DSALinkedList to help with FileIO. Including this would over-complicate the diagram.
UnitTestDSADigraph
+ main(string[])
+ populate(DSAGraph)
+ testHasVertex(DSAGraph)
+ testGetVertexCount(DSAGraph)
+ testGetEdgeCount(DSAGraph)
+ testGetVertex(DSAGraph)
+ testGetAdjacent(DSAGraph)
+ testIsAdjacent(DSAGraph)
+ testAddVertex(DSAGraph)
+ testAddEdge(DSAGraph)
+ spacer()
+ loadFile(string, DSADigraph)
DSAHeap
- heap : DSAHeapEntry[]
- count : int
+ CONSTRUCTOR(int, object)
+ add(int, object)
+ remove() : object
+ getCount() : int
+ heapSort(Object[][]) : DSAHeapEntry[]
- trickleDown(int)
- trickleUp(int)
- getLeftChildIdx(int) : int
- getRightChildIdx(int) : int
- getParentIdx(int) : int
UI
+ InteractiveMod()
+ SimulationMode(string, string, double, double)
- scanString(string) :string
- scanInt(string) : int
- scanDouble(string) : double
FileIO
+ saveNet(string, Network)
+ loadNet(string, Network)
+ loadEvents(string, Network)
+ appendStats(string, string)
UnitTestDSAHeap
+ main(string[])
+ populate(DSAHeap)
+ testOne(DSAHeap)
+ testSort(DSAHeap)
+ title(string)
+ load(DSAHeap, string)
DSAHeapEntry
- priority : int
- value : object
+ CONSTRUCTOR(int, object)
SocialSim
+ main(String[])

Class Descriptions

SocialSim

This is a sort of ‘wrapper class’ to handle the command-line arguments of the program and call the UI in the correct way. It also handles any last resort exception handling. I chose to make this separate from UI so that I could have a class specific to UI for the sake of class responsibility.

UI

This handles the storage of the Network class instance for simulation or interactive mode. It displays most of the program’s user output (Apart from some things that made more sense to include in Network). This class contains a few user input methods to simplify the code for getting user input in InteractiveMode().

Network

This is the majority of the implementation of the social network simulator itself. This class contains a host of mutator methods for adding and activating events, and accessors for interactive mode features such as graph displaying and user-specific statistics. This class also contains the 4 methods used for post propagation and event handling, and some extra private helper methods to help with data storage.

Post

One of the 3 private inner classes of Network, Post is responsible for storing the information for each post in the network. Similar to the other Network inner classes, it does not contain any useful methods, but Post specifically stores a lot of information in it’s classfields. This includes two linked lists, firstly ‘seen’, which lists all users that have seen the post, and secondly connections, listing the post’s connected users, which is the basis of post propagation. Posts are uniquely identified by a post ID, which are generated sequentially using a method in Network. Inside Network, instances of this class are stored in the events queue to detail when a new post is to be set to active, and the posts linked list which represents all active posts.

User

Another Network inner class, User stores the username and statistics of a user. Similar to the other Network inner classes, it doesn’t contain any useful methods apart from a toString() and equals(). Inside Network, instances of this class are stored in the events queue to detail a new user or a removal of a user, and they are used as the values for the vertices in the graph.

Follow

The final Network inner class, this one stores two usernames, describing a one-way follow in the network. Inside Network, instances of this class are stored in the events queue to detail a follow or unfollow.

FileIO

This program requires a lot of File IO, and because File IO is so error-prone, I thought it would be a good idea to put this functionality in a separate class. This class handles saving and loading network files, loading event files and appending to log files.

Justification

I use a number of ADTs in the Network class, all with varying justifications.

Using a directed graph to store users (vertices) and follow relationships (edges) allows me to think graphically about the operations being done in the program and overall helped me to get everything working. I ended up having edges point in the opposite direction from what I find instinctive, having the user being followed be the edge start, and the user doing the following being at the point of the arrow. This better represents the flow of information through the network and simplified my post propagation implementation significantly. In terms of overheads and time complexity, the graph is quite costly with it's many linked lists, but it's such an integral part of the program that it is worth the sacrifice.

I chose to use a queue to store all upcoming events. As the behaviour of entering events on one side and taking them out on the other fits with a queue's FILO behaviour. My queue uses a double ended, doubly-linked list, and in some cases, the queue can get many elements in it. While this does create a significant memory overhead, while the program runs it is constantly dequeuing, so the problem quickly goes away.

I used a linked list to store all active posts that were currently being propagated. While in simulation mode only one post propagates at a time, interactive mode allows you to break out of this and propagate as many posts at a time as desired, so this needed to be an ADT rather than a single post variable. A linked list worked well as it's large memory overhead was insignificant at this small of a scale and my specific linked list implementation allows for the removal of a node from any point, without the need for a key. These are the same reasons I used linked lists in the Post classfields to track user activity.

To be able to implement the popularity view, showing all posts and users in the order of popularity, the obvious choice was a max priority heap. I used it in two different ways, for tracking posts, I used a heap classfield that I added to whenever a post finished propagating. For tracking users, I decided to insert all of the data when the popularity view was called. This means I don't have to update a heap every time a post is liked, or a user is followed. This is advantageous because of the computationally expensive trickle operations the heap must perform on every insert. One downside to the post tracking heap classfield is that I needed to hard-code a maximum capacity. This problem would be negated if my implementation of a heap was self-resizing.

All of these ADTs were previously submitted in practical sessions. Linked list and queue in Prac 3, Heap in Prac 7, and Graph is Prac 5.