Model Description

Introduction

To begin with, to store the buddy connections, we implemented an abstract class called BuddyLand. The BuddyLand class has the structure and features of an undirected graph network.

It is consisted of buddies, which are the graph vertices and also connections with buddies which are the graph's undirected edges. An undirected edge is simply a connection that goes both ways. To put it simply, if V1 is connected to V2 by some path, then that same path takes V2 to V1.

From the Math perspective, there are different ways of representing graphs such as edge lists, adjacency matrices and adjacency lists.

Now on the programming side,

The edge list is out of the options from the get-go, simply because it presents a lot of redundancy, especially when we are concerned about memory and we certainly would be, if we were to build a large social network. The edge list is essentially a 2 column matrix, storing the edges of the graph. It is the easiest to implement, but it falls short because we would need to fill 100 rows if a certain vertex has 100 connections. Performance-wise, we would also need to look through multiple rows to check for the number of connections.

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	1	0	1	0
1	1	0	0	0	1	0	1	0	0	1
2	0	0	0	0	1	0	1	0	0	0
3	0	0	0	0	1	1	0	0	1	0
4	0	1	1	1	0	1	0	0	0	1
5	0	0	0	1	1	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	1
8	1	0	0	1	0	0	0	1	0	0
9	0	1	0	0	1	0	0	1	0	0

The Adjacency Matrix is similar to the edge list but stores the data much more efficiently.

The principle is simple, we store a 1 in position (i, j) only if there exists an edge between vertices on i and j. While it's way better than the edge lists space wise, it's still not ideal. Even for relatively few edges, we would have to store 0-s, which still takes space, so $O(N^2)$ memory complexity.

Performance wise, if we know the position (i,j) it would take constant time O(1) to check whether or not two buddies(vertices) are connected. That's hardly the case, when checking if there exists a connection between two buddies, we will have the two buddies as arguments, without knowing their indexes in the matrix. To make this possible, we could implement a HashMap storing the Buddies and their indexes. We could retrieve their indexes in constant time from the HashMap and also retrieve whether or not they have a connection in constant time from the matrix. However, that would include a lot more memory usage, as now we are storing every Buddy(vertex) again in a HashMap. Searching for a buddy by first or last name would take O(N^2) because we have to look through each vertex in the matrix. Even if we sort it, there is no way of sorting it in such a way that we could do binary search for both first name and last name, we can only sort the list based on one parameter.

That said, it brings out to the third and our chosen option, the adjacency list. Time and memory complexities vary depending on the data structures chosen to represent the mapping and also the edges lists or sets.

Implementation of the adjacency list

To implement the adjacency list, we used the HashMap implementation of Java Map interface. The HashMap is of type <Buddy, TreeSet<Buddy> meaning <Vertex, Edges> in graph terms. We store each Buddy in our network as a HashMap key and we have a set of edges for each key. When a Buddy account is firstly created, it is added as a new key to the HashMap. When a connection is added between two buddies, we add the second buddy to the first one's edges set and the first buddy to the second buddy's edges set(that happens because the graph is undirected, same as Facebook's friends system for example). To remove a connection between two buddies is also extremely similar. We remove buddy2 from buddy1 connections set and perform the same on buddy 2. When a buddy searches for a friend of them by first or last name, their BuddyList is linearly checked for any matches. To retrieve the number of friends, we simply perform get(key).size().

The reason behind choosing a HashMap

When mapping is needed, Java offers a few options such as TreeMap, HashMap or LinkedHashMap. For this type of task, we will need to perform lots of adding and deleting, retrieving and also searching.

Let's analyze each type by their time complexities.

1. **Time complexity to insert one key in each Map:**
   - TreeMap - O(logn)
   - HashMap - O(1)
   - LinkedHashMap – O(1)
2. **Time complexity to insert m keys in each Map:**
   - TreeMap – O(m*logn)
   - HashMap – O(m)
   - LinkedHashMap – O(M)
3. **Deletion has the same time complexity as adding**
4. **Time complexity to retrieve a key from each Map:**
   - TreeMap – O(logn)
   - HashMap-O(1)
   - LinkedHashMap-O(1)

TreeMap would be the option to go if we cared about the order of the key and value sets. In our case, we don't need the key to be sorted, so we rule the TreeMap out since while still really fast, it's slower than the HashMap.

Between HashMap and LinkedHashMap, while their performance is extremely similar, we chose the HashMap simply because of less memory overhead. The LinkedHashMap uses a doubly linked list to back it up, so every node has two pointers, one pointing to the previous element and one to the next. Important thing to note, the ordering of the keys is not important, however, we want edges to be sorted by their username. While HashMap does not maintain order, it does not affect the order in the edge sets because that depends on the type of list or

set used to implement the HashMap. Everything said, it brings us to our implementation of the HashMap.

Our Implementation of HashMap

Other than the type of Map to be chosen, it is also important to choose an efficient structure to store the connections of each buddy. There are a lot of options such as a modified ArrayList of our own called SortedArrayList, a LinkedList or a TreeSet. ArrayList is out of the options because we want these BuddyLists sorted by username.

Let's analyze each type by their time complexities.

1. **Time complexity to insert one key in each List(or Set):**
   - SortedArrayList - O(nlogn), O(logn) to find the right index and O(n) to copy the array over.
   - LinkedList - O(n), in order to always be sorted
   - TreeSet– O(log(n))
2. **Time complexity to insert m keys in each List(or Set):**
   - SortedArrayList – O(mnlog(n))
   - LinkedList – O(mn) in order to always be sorted
   - Treeset– O(mlog(n))
3. **Deletion has the same time complexity as adding**
4. **Time complexity to search for a value from each List(or Set) provided the buddy:**
   - SortedArrayList – O(logn)
   - LinkedList-O(n)
   - TreeSet – O(n), because it is a set so it isn't indexed

LinkedList would be the best option here if we did not care about it being sorted or not. Inserting elements at the correct places would take O(n) time, we have to iterate through all the list. We also created a SortedArrayList class which extends Java ArrayList. The way we keep it sorted is through binary search. We use it everytime we want to add an element in order to find the index where it belongs for the ArrayList to

remain sorted. But ArrayList falls short simply because of its array implementation. An array needs to be copied each time something is added in a place other than the end because of the way it is stored in memory. If we insert somewhere in the middle of the list, since an array is stored sequentially in memory, we would have to create an entirely new array with a new size and copy everything over. Thus, it would add linear time to the already logarithmic inserting time. The fastest out of the bunch to insert in sorted order is the TreeSet, backed by a Red-Black binary tree. A TreeSet also has an advantage of not containing duplicates by design, something we definitely want to be the case.

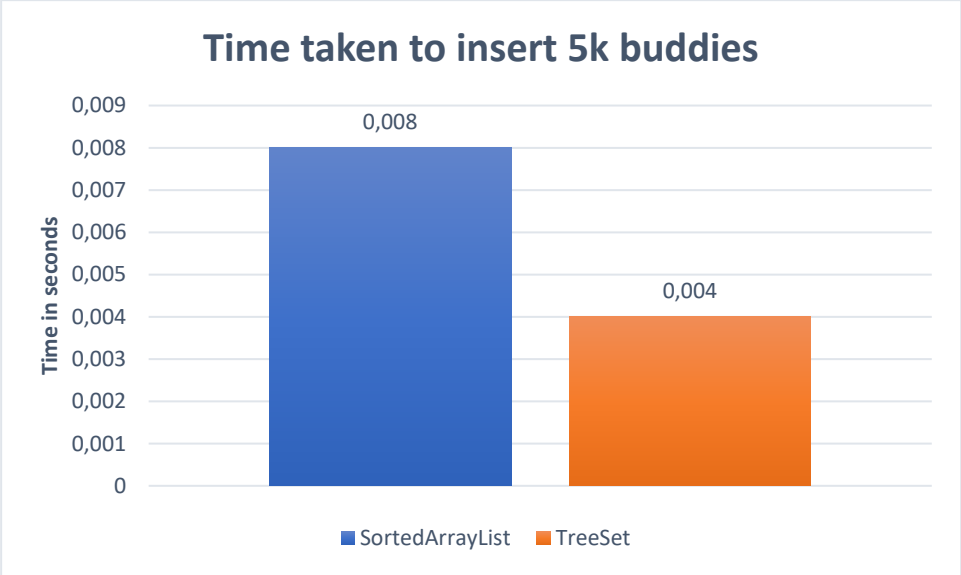
The advantage of the SortedArrayList is faster searching time. However, that's only when searching by username, because username is what we used to sort the list. We cannot sort by both first or last name because they are two different fields and the requirement was to be able to search for both of them. If we sorted by name, there's a possibility of losing results when searching for last name, because the last names would not be sorted. We could use two other **HashMap<Buddy, SortedArrayList>** specifically listing first and last names in order so we could search them both in O(logn) time and retrieve our results. That would simply consume way more space as we have to introduce four new HashMaps, two to map each buddy to lists of their buddies first or last names and two other maps to map each first and last name to a Buddy. We cannot simply retrieve a Buddy by just knowing their name. However, consuming more memory and having faster searching would definitely be the first choice when introducing the concept of pages, as pages would have lots of followers thus much larger BuddyLists, so O(n) would be impractical for let's say, lists consisting of +100K followers.

After looking at the much larger memory footprint to make searching faster, we decided to keep it at O(n) considering the BuddyList for each Buddy will be midsized, so O(n) is not bad at all. On average, a buddy might have 500-2000 friends, so linear search would seem instantaneous. Since the TreeSet is way better at adding and removing than the SortedArrayList, to implement the HashMap we used the Buddy and TreeSet<Buddy> type

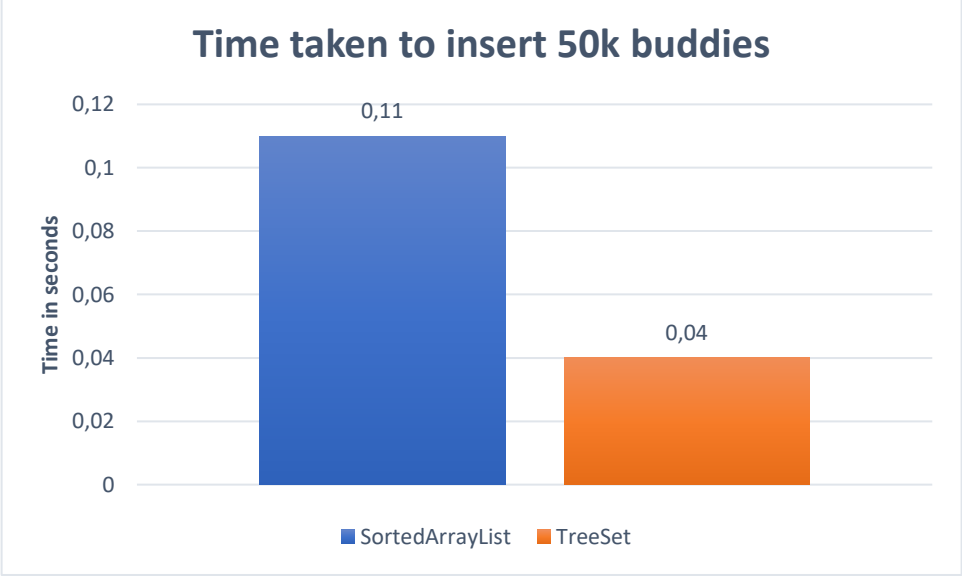
```
private static Map<Buddy, TreeSet<Buddy>> connections = new HashMap<>();
```

Comparing **SortedArrayList** and **TreeSet** performance using test data

To create each chart, we ran 10 tests and recorded the total time it took for the SortedArrayList and the TreeSet to be created. Then we took the average of the 10 tests for each, so that the data would be more accurate.

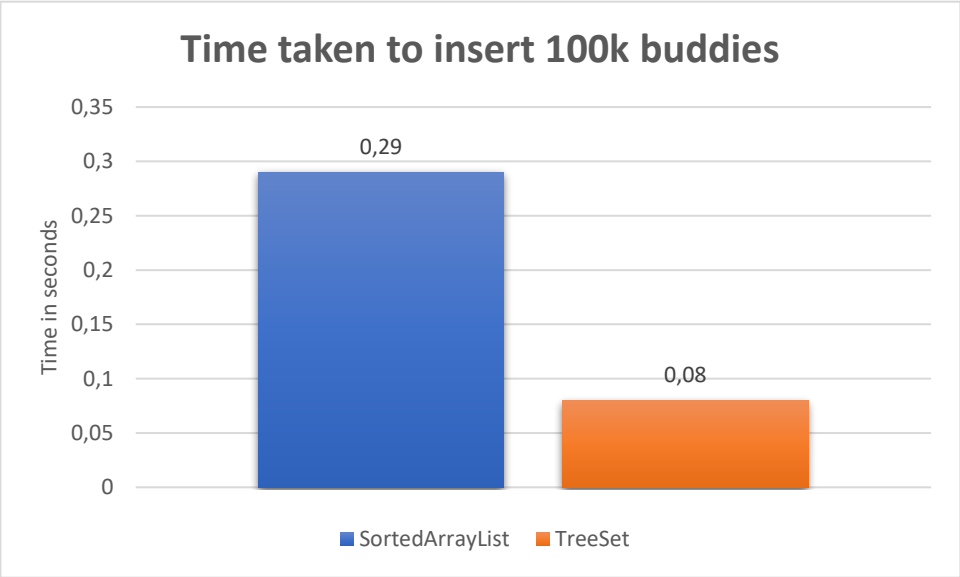


Initially, the TreeSet performs twice as fast.

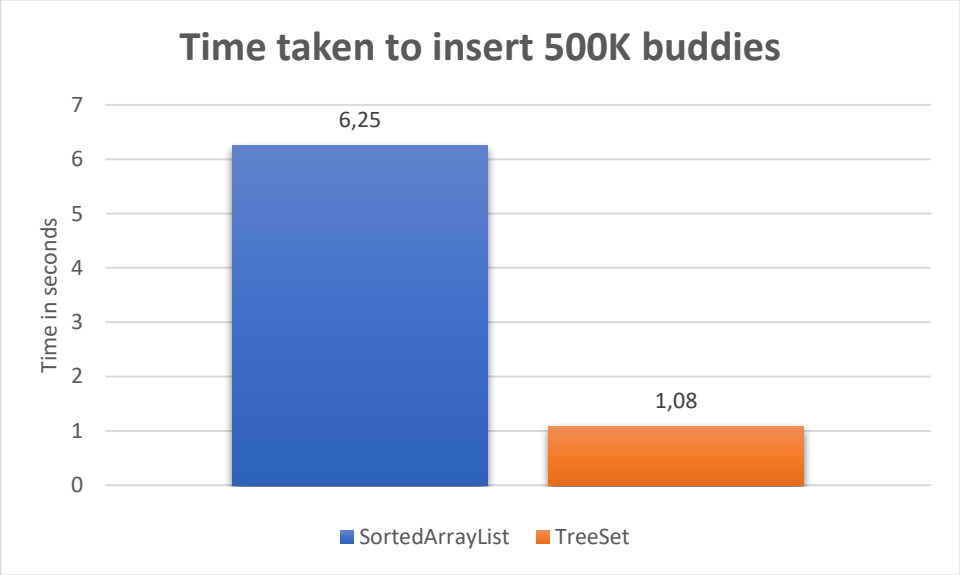


Now we can see the difference getting larger. SortedArrayList time when the lists get bigger grows much faster. Now the

TreeSet is 2.75X faster.



TreeSet 3.6 times faster with a difference of 0.21 seconds.



TreeSet a whopping 5.7 times faster, with a huge difference of 5.17 seconds.

As we can see, the performance of the TreeSet is much better when adding. Up to 100k, the difference is negligible, however, in a real world scenario, that data would be added in a server. Imagine how many requests there will be in a second.

When thinking at scale, even a minimal time difference makes a huge difference when handling multiple requests per second.

Deleting is also almost identical.

For searching by first or last name, it's also practically the same, we have to iterate through all the list to get all the matches and then return a List<Buddy>.

For our use case, TreeSet is the superior data structure to store the edges of the graph.

Fulfilling design requirements

We created a Buddy class with all the required fields. The only thing worth mentioning is the way we maintain a unique username for each Buddy. Initially, in the class's constructor, when adding the username field we wrote:

this.username = setUsername(username) instead of this.username = username

We created a setter in order to check if the username existed in BuddyList of that buddy(initially we were using Java ArrayList). If the username existed, the method would throw an exception indicating that the username existed. The constructor would then catch it and proceed to print on the screen that it already existed. While it achieved what we needed, it was a very bad choice from a design and object-oriented perspective. We want classes to just be blueprints for creating objects.

After deciding that was a bad idea, we changed the Buddy's constructor to a standard one and found another way. The fact that another User object with that same username is not important, it becomes important at the moment we decide to add it to our BuddyLand graph structure.

So, instead of checking if the username exists at object creation, we check when we attempt to add it to the HashMap and HashMap is a great choice to achieve our task. Why is it? Simply because it stores elements using their hash values and a HashMap won't add two values of the same hash. How can we make two different objects have the same hash when they have the same username? Simple, overriding the hashCode() method in Buddy does the trick. We set it to

return the username string hashCode, instead of a unique hash for each object. That way, when we attempt to add an object with the same username in the HashMap key set as a new key, it simply won't add it if it already exists and the extremely good thing about that is that all the checking can be done in O(1) time. Problem solved. However, while the HashMap does not allow duplicate keys, it doesn't check the key values. To ensure are the BuddyLists contain unique usernames, the collection where we store the values also shouldn't allow duplicates. Initially, we checked if the SortedArrayList contained the element before adding it, but when we switched to TreeSet, it does the job for us. A TreeSet does not allow duplicates by design, binary search trees do not allow duplicates and also sets do not allow duplicates. Since the TreeSet compares the objects instead of the hashes, we also had to override the equals() method to return true when two buddy's usernames match. That way, when the TreeSet searches if it contains a Buddy, it will return true if it finds that Buddy's username.

The HashMap ensures that there are no duplicate keys and the TreeSet ensures there are no duplicate values for these keys.