COMP3506 – Assignment 4

s4529458

Due 27/09/2019 5:00 pm

Implementation Details

FeedAnalyser (constructor)

This makes use of a HashMap which maps a username to an ArrayList of their posts. HashMap is implemented by the Java Collections Framework using a hash table [1]. By default (which we use), this has a load factor of 0.75, meaning the hash-table is resized when 75% of its buckets are full. This is a compromise between time and space costs and results in (amortised) $\mathcal{O}(1)$ insertions and lookups, with linear space usage. In the worst case, it is $\mathcal{O}(n)$ if the hash table needs to be resized.

This ArrayList contains all posts made by a user and has amortised $\mathcal{O}(1)$ insertion but $\mathcal{O}(n)$ worst-case (here, n is number of items in the array) if it needs to be reallocated [2]. We also use another ArrayList to store all posts in order of their ID.

We use a PriorityQueue to store posts, in order of their upvotes. Higher upvoted posts are dequeued first from the queue.

Suppose there are n FeedItems and the get methods on FeedItem are $\mathcal{O}(1)$. In the constructor, the `while` loop iterates n times (once per item), each iteration taking $\mathcal{O}(\log n)$ time because of the PriorityQueue insertion to a heap [3]. ArrayList add and HashMap put are both amortised $\mathcal{O}(1)$ as discussed earlier. As a whole, this loop takes $\mathcal{O}(n \log n)$. After the loop, we sort each list in the HashMap by the date of the posts, and we sort the other array by post IDs. Java sorting is bounded by $\mathcal{O}(n \log n)$ [4], so the constructor is bounded by $\mathcal{O}(n \log n)$ in the worst case.

getPostsBetweenDates

This performs a lookup on the HashMap to get one user's posts, in $\mathcal{O}(1)$ time. This gets an ArrayList of this user's posts, sorted by the date of the post.

Then, we do two binary searches to find the earliest post after the start date and the latest post before the end date (as needed) and use `sublist()` to return the range of posts between these indices. The binary searches take $\mathcal{O}(\log n)$ due to it halving the search space every iteration and `subList` is $\mathcal{O}(1)$ because it only returns a view into the full list, without copying.

Note that the array we search on only contains the posts for this user so will often be smaller than n items in size if there are multiple users posting.

The algorithm is worst-case $\mathcal{O}(\log n)$ and uses $\mathcal{O}(n)$ space for the HashMap and ArrayLists; every FeedItem needs to be stored once in some user's ArrayList.

This algorithm is obviously better than brute-force search which would take $\mathcal{O}(n)$ time. An alternative efficient implementation could be using a TreeMap keyed by the date of the post. This would allow efficient indexing of ranges as well [5]. However, it has the disadvantage of only storing one post per date. To store multiple posts per date, you would need to store lists in the TreeMap. Then, you would need to aggregate all the lists into one before returning, increasing the runtime to $\mathcal{O}(\log n + k)$ where k is the number of posts between the two dates. This is clearly unsuitable when managing large amounts of posts or large ranges of times, and also has the overhead of one new list per date.

getPostAfterDate

This performs one lookup on a HashMap and a binary search on a sorted ArrayList. These are $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$ respectively. Thus, this is $\mathcal{O}(\log n)$ worst-case.

This reuses the data structure of getPostsBetweenDates which was $\mathcal{O}(n)$ space. In the implementation, we actually reuse a helper method for getPostsBetweenDates. A brute-force algorithm on a sorted array (instead of binary search) would take $\mathcal{O}(n)$ by iterating through each index and comparing it to the search date. The binary search implemented is strictly better in runtime and equivalent in space.

getHighestUpvote

This is just a heap dequeue which is $\mathcal{O}(\log n)$ time, using a precomputed max-heap of posts by their upvotes. This implementation takes $\mathcal{O}(n)$ space to store the queue, where n is the total number of posts.

An alternative could be using a sorted list to store the posts in order of upvotes. The queue implementation was chosen because it is able to build the heap as the feed items are parsed. This is advantageous if the feed items are from a slow source (e.g. network) because the heap can be sorted asynchronously while waiting for the next item. This avoids needing to wait for all feed items, then sorting the array. The sorted array would be better if, for example, we needed to find the N-th highest upvoted post without repeatedly calling this method.

getPostsWithText

This uses an implementation of the Boyer-Moore algorithm to search the text of every post. Suppose m is the length of the pattern, n is the number of posts and k is the maximum text length.

Preprocessing for the Boyer-Moore algorithm takes $\mathcal{O}(m)$ time with $\mathcal{O}(m)$ space and this is done once per call. The algorithm itself runs in $\mathcal{O}(m + k)$ if the search pattern does not appear and $\mathcal{O}(mk)$ if it does [6].

Thus, searching every post for this pattern will take $\mathcal{O}(np(mk) + n(1 - p)(m + k))$ worst-case where p is the proportion of posts which match the search pattern. This uses $\mathcal{O}(m)$ extra space.

Furthermore, because of the skip rules of the Boyer-Moore algorithm, it is well-suited to natural English text where the search pattern does not contain many characters in the text and/or does not have repeated segments. If BM encounters a letter in the text not in pattern, it can skip over the length of the pattern (bad character rule) [7]. Similarly if BM matches only a suffix of the pattern then fails, it can skip the rest of the pattern if this suffix does not reoccur in the pattern (good suffix rule) [7].

An alternative would be to use a suffix tree which precomputes all suffixes of each text. This would increase the preprocessing time (in the constructor) to $\mathcal{O}(nk)$ but decrease the runtime of `getPostsWithText` to $\mathcal{O}(nm)$ which, notably, does not depend on text length k [8]. However, this is only advantageous if many substrings will be searched. The current Boyer-Moore method will be better if few patterns are searched because it does not have the time cost of computing unneeded suffix trees.

A very naive linear search algorithm would take $\mathcal{O}(nkm)$ time to search every position of every post's text. For a small space cost, the Boyer-Moore algorithm dramatically outperforms this in runtime. Because we assume the search pattern will often be much smaller than the text, even this space cost is negligible.

References

[1] Oracle, “HashMap (Java Platform SE 8),” Mar 2019. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

[2] ——, “ArrayList (Java Platform SE 8),” Mar 2019. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/AraryList.html

[3] Oracle, “PriorityQueue (Java Platform SE 8),” Mar 2019. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html

[4] Oracle, “Arrays.sort() (Java Platform SE 8),” Mar 2019. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-java.lang.Object:A-

[5] Oracle, “TreeMap (Java Platform SE 8),” Mar 2019. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

[6] M. A. Sustik and J. S. Moore, “String searching over small alphabets,” University of Texas at Austin, Tech. Rep. TR-07-62, Dec 2007.

[7] B. Langmead. Boyer-Moore (Johns Hopkins University). [Online]. Available: http://www.cs.jhu.edu/~langmea/resources/lecture_notes/boyer_moore.pdf

[8] M. Kay, “Substring Alignment using Suffix Trees,” 2004. [Online]. Available: https://web.stanford.edu/~mjkay/CYCLING.pdf