COMP3506 – Assignment 4

s4529458

Due 27/09/2019 5:00 pm

Implementation Details

FeedAnalyser (constructor)

This makes use of a HashMap which maps users to a tree map. 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.

The TreeMap maps dates to an ArrayList of posts made on that day. This is implemented by the JCF as a red-black tree [2]. This has the property of $\mathcal{O}(\log n)$ insertion and lookup in all cases [3].

Finally, the ArrayList is an array-backed list with amortised constant-time insertions [4].

Suppose there are n FeedItems and the get methods on FeedItem are $\mathcal{O}(1)$. In the constructor, the `while` loop iterates n times, each iteration taking $\mathcal{O}(\log n)$ time because of the TreeMap inseration. ArrayList add is $\mathcal{O}(1)$ and PriorityQueue add is $\mathcal{O}(\log n)$ because it uses a heap [5]. As a whole, this loop takes $\mathcal{O}(n \log n)$ and the array sorting algorithm used by Java is bounded by $\mathcal{O}(n \log n)$ [6]. Thus, 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. Then, we index the appropriate part of the TreeMap using `subMap()`, `headMap()` or `tailMap()` to get the range of posts between the given dates. This is done using lookups which are always $\mathcal{O}(\log n)$ for a red-black tree since they are balanced [7]. Note that this TreeMap only contains the posts for this user, so will often contain less than n items if there are multiple users posting.

Then, we collect the lists of across all dates in the range into an ArrayList, which takes $\mathcal{O}(k)$ time where k is the number of posts falling within the range.

The algorithm is worst-case $\mathcal{O}(\log n + k)$ and uses $\mathcal{O}(n)$ space as every FeedItem needs to be stored once in some list of the TreeMap. There will be some overhead of using one array per date but that is unavoidable since multiple posts can be made at the same date-time, and it does not affect asymptotic space.

This algorithm is obviously better than brute-force search which would take $\mathcal{O}(n)$ time. An alternative implementation could be using a binary search on an array of posts sorted by date. This has the advantage of the data structure being simpler than a

TreeMap, but operations require manually binary searching through the array. We choose TreeMap which provides a simpler interface with equivalent $\mathcal{O}(\log n)$ performance for lookups, which is advantageous if the code needs to be easy to read and understand, e.g. for lecture examples.

A disadvantage of this is needing to collect posts from different dates into one list to return, which takes $\mathcal{O}(k)$ time. We assume that $k \ll \log n$ so this is not a problem. An list-backed implementation would be able to use Java's `subList()` to return a range of posts in constant time, reducing runtime to $\mathcal{O}(\log n)$.

getPostAfterDate

This performs one lookup on a HashMap and one lookup on a TreeMap. These are $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$. This returns an array which is indexed in $\mathcal{O}(1)$ time. Thus, this is $\mathcal{O}(\log n)$ worst-case.

This reuses the TreeMap of getPostsBetweenDates which was $\mathcal{O}(n)$ space. Similar to above, this could also be done using a sorted list and binary search. However, since we already have the TreeMap implementation, we simply use that to avoid introducing a new data structure.

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 [8].

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.

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 [9]. 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.

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) [10]. 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) [10].

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

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed. MIT Press, 2009.

[4] Oracle, "ArrayList (Java Platform SE 8)," Mar 2019. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

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

[6] 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-

[7] B. Hasti, "CS 367: Red-Black Trees," 2012. [Online]. Available: http://pages.cs.wisc.edu/~skrentny/cs367-common/readings/Red-Black-Trees/

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

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

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