For all parts of this document, let n denote the number of FeedItems.

Part 1: getPostsBetweenDates(String username, Date startDate, Date endDate)

For this task, a HashMap was used to organise posts by username (mapping from username to list of posts). To list each user's posts, an ArrayList was used. The total memory used is that of the FeedItems, and the pointers to the usernames and lists of posts in the hashmap. Because the pointers to usernames and lists use a constant amount memory, and there will be at most n of them, they have $O(n)$ memory complexity (in the worst case - the best case where all posts are by the same user will have $O(1)$ memory complexity). There are n FeedItems, each using constant memory, meaning the items themselves have $O(n)$ memory complexity in all cases. The lists are sorted using List.Sort, which uses merge sort. This uses an additional $O(m_{max})$ memory, where m_{max} is the maximum size of one of the lists ($O(n)$ in the worst case). Therefore, the data structure used for this has overall $O(n)$ memory complexity in all cases.

In preprocessing, all FeedItems are looped through. For each, it checks whether or not the user has an entry in the map yet (an $O(1)$ time complexity operation for a HashMap). If they don't, a new ArrayList is created and placed into the HashMap, otherwise the existing list of that user is retrieved and updated. Adding a post to an existing ArrayList has $O(1)$ amortized time. Inserting an entry into the HashMap has an $O(1)$ best case and expected time complexity, however in the worst case where it collides with every other existing element, it has $O(m)$ time complexity, where m is the number of entries already in the HashMap. The overall worst case is where there are n lists, each colliding with every other list that has already been added. In this case, m will take on every value from 0 to $n-1$ exactly once, making it on average approximatley $n/2$ (resulting in an $O(n)$ amortized time complexity). Repeating this for all n FeedItems would result in this taking $O(n^2)$ running time. In the expected case, inserting into the HashMap will have $O(1)$ amortized time, resulting in an expected time complexity for this step of $O(n)$.

After this is done, the entries are looped through, and ArrayLists sorted using List.Sort (which uses merge sort). Merge sort has $O(n \log n)$ worst case and average case time complexity. In the best case, there will be n lists with 1 element each, resulting in each sort running in constant time (as sorting a single element takes constant time), and this step running in $O(n)$ time. The worst case is when all FeedItems are in one list, meaning the sorting will take $O(n \log n)$. The average case would be between these two cases, and depends on how FeedItmes are typically distributed between users. If there are typically many users with few posts, the average case would be $O(n)$, but if there are typically few users with many posts, the average case would be $O(n \log n)$ (intuitively, I don't know how to prove this. It may actually be something in the middle like $O(n \log(\log n))$). $O(n \log n)$ for the average case will be assumed for the rest of this question.

This means that the overall time complexity for pre-processing is $O(n^2)$ in the worst case, and $O(n \log n)$ in the average case.

When running the actual method, first the list for the given username is found ($O(n)$ worst case, $O(1)$ average case). Next, a binary search algorithm is used to find the index of the next post after the start date, and again for the end date ($O(\log n)$ worst case and expected time complexity). The end index is then incremented if necessary (due to how the search algorithm was implemented), using all constant time operations. List.subList is then used ($O(1)$ running time for an ArrayList) with the starting and ending indexes to get all relevant posts, and the sub list is returned. This means the time complexity of a call to this method is $O(n)$ in the worst case, $O(\log n)$ in the expected case, and $O(1)$ in the best case (the binary search finds the dates on its first try).

A HashMap was used so that filtering FeedItems by username runs is constant expected time. ArrayLists were used within the HashMap so that subList runs in constant time. An alternative implementation could be to use some sort of search tree to order the posts, such as an AVL tree or 2-4 tree (ordered by date). For these, insertion runs in $O(\log m)$ worst case time (m being the number of elements already in the tree - $n/2$ on average), resulting in a $O(n \log n)$ worst case time for pre-processing (better than the hashmap

implementation). The drawback to this, is that to get all elements between two dates, an inorder traversal would need to be run between the two nodes corresponding to the dates. This would have an expected time complexity of $O(l)$, where l is the number of posts between the two dates. For dates which are on the scale of the age of the system, this will become $O(n)$, a worse expected running time than the current implementation.

Part 2: getPostAfterDate(String username, Date searchDate)

This method has the same pre-processing (including memory complexity) as getPostsBetweenDates. The method itself is also similar, and has the same time complexity as getPostsBetweenDates ($O(n)$ in the worst case from accessing an entry of a HashMap, $O(\log n)$ expected case for a binary search).

Again, an AVL or 2-4 tree could be used, for similar benefits, but has similar drawbacks. The use of a HashMap makes filtering by username take constant time in the average case, however a search tree does not have an effective way of filtering by username other than by checking each node. Using a search tree, the algorithm would have to check the username of the first node found, and if it doesn't match, inorder traverse to the next node and try again, repeating until the right user is found, or there are no more nodes. If the user has not made a post for a long time (or at all) after the specified date, this will again have an $O(l)$ time complexity (where l is the number of posts between the date, and the user's next post), resulting in $O(n)$ expected running time

Part 3: getHighestUpvote()

For this task, an ArrayList of FeedItems was stored ($O(n)$ memory complexity), along with an int storing an index ($O(1)$ memory complexity). The array is also sorted using the sort method in java.util.list (which uses merge sort), which requires an additional $O(n)$ memory. This results in an overall memory complexity of $O(n)$.

The first step in preprocessing is iterating through all FeedItems, and adding them to the list. All operations involved in this have $O(1)$ time complexity, and are repeated n times, resulting in a time complexity of $O(n)$ for this step. Next, the list is sorted using List.Sort (merge sort), which has a worst and average case time complexity of $O(n \log n)$. The index is also set to last element of the list. This requires some primitive operations, and an ArrayList.size() call, all of which have $O(1)$ time complexity. This results in an overall pre-processing time complexity of $O(n \log n)$.

The reason an ArrayList was used, is that retrival given an index is has $O(1)$ time complexity, as does decrementing the index, and checking if the index is less than 0. This means the getHighestUpvote() method has $O(1)$ time complexity. This is not the case with a LinkedList.

An alternative would be to use a heap/priority queue. This way pre-processing would only required adding, which would be $O(n)$. However, accessing all n elements would be equivalent to a heap sort ($O(n \log n)$), resulting in the method itself having an amortized time complexity of $O(\log n)$. This makes a sorted list better if having fast individual calls is more important than pre-processing time, however if this method is only generally going to be called a constant number of times compared to n, the total time complexity for pre-processing and method calls will be $O(n)$ with a priority queue, which is asymptotically faster than the pre-processing a sorted list ($O(n \log n)$). For both implementations, pre-processing, then calling the method n times will be $O(n \log n)$.

Part 4: getPostsWithText()

Let p be the size of the pattern to match, s be the size of the alphabet (the ASCII values from 32 to

126), and m be the size of the content of the FeedItem. For this task the pre-processing is identical to getHighestUpvote, except with the list being sorted by ID rather than Upvote count, and without the index being assigned. This means memory complexity is $O(n)$ and time complexity is $O(n \log n)$.

The algorithm used to search for text is the Boyer-Moore algorithm. An array for the last occurence is calculated first, which has $O(s)$ memory complexity, and $O(p + s)$ time complexity. In the worst case, the search part of the Boyer-Moore algorithm has $O(pm)$ time complexity, however the average case would be closer to $O(p + m)$, as the heuristics used in the algorithm result in most characters being checked far less than p times. This results in an overall worst case time complexity for each Item of $O(pm + s)$, and average case of $O(p + m + s)$. The last occurence function only needs to be node once, meaning the overall time complexity of a method call $O(npm_{max} + s)$ in the worst case, and $O(n(p + m_{max}) + s)$ in the average case, where m_{max} is the size of the longest comment.

The alternatives to Boyer-Moore are the KMP algorithm, or a trie. The use of a compressed trie would reduce the time complexity of the string searching to $O(p)$, which is better than the average case for the Boyer-Moore algorithm. However, in the pre-proecssing, a trie would have to be created for every FeedItem, and each trie would have to contain every substring of any length. ($O(m^2)$ for each trie - traversing through the item content, after x characters, there will be x substrings ending with the previous character. Each of these leaves will get a child when the next character is added. Summing this over the entire string will be $O(m^2)$). This means pre-processing will be $O(nm^2_{max})$ in the worst case (largest comment is much larger than any other), where m_{max} is the largest size of comments in feed items. This may be better than $O(n \log n)$ when the strings are short, but this is not guaranteed in this scenario. Memory complexity will be also be $O(nm^2_{max})$ (storing every substring of m length will take $O(m^2)$ memory).

The KMP algorithm has a better worst case time complexity of $O(m + p)$, however, the alphabet being searched through is fairly large, meaning KMP algorithm is not likely to have high values for the failure function. This means times closer to the worst case are more likely, whereas for Boyer-Moore, a larger alphabet will generally mean larger jumps, meaning times close to the worst case are less likely. Conversly, the worst case for Boyer-Moore's algorithm is when there are many repeated characters (e.g. searching for "baaa" in the comment "aaaaaaaaaa"). This is less likely with a large alphabet.

If the comments are short compared to the alphabet size, a brute force algorithm might also be better than Boyer-Moore, as there isn't required preprocessing, meaning $O(mp)$ could be better than $O(p+m+s)$, although this is specified to not necessarily be the case.

In terms of the data structure to store the FeedItems for this method, any iterable, sortable (in $O(n \log n)$ time) data structure would suffice, as no particular indexes are needed, and all elements are being iterated through. This could include a LinkedList or a Search Tree.

Overall:

Overall, the FeedAnalyser class has $O(n)$ memory complexity. One downside to the way it was implemented is that the same data (the FeedItems) is being stored three times. This was done as it does not increase the memory complexity, or the pre-processing time complexity asymptotically. An alternative would be to only store the FeedItems once, however seeing as some of the methods' efficiency rely on the FeedItems being stored in a certain order, and the field being sorted by is different for different methods, at least one of the methods would have to be asymptotically less efficient to allow for this. For example, if only the list sorted by upvote count is stored, the getPostAfterDate algorithm would likely have to use a brute-force approach, which would be $O(n)$ in the worst case (rather than $O(\log n)$).

The overall time complexity of the structure is $O(n \log n)$ in the expected case, and $O(n^2)$ in the worst case (from taking the max time complexity of the pre-processing of each method).

A way of reducing memory used in pre-processing would be to implement an in place quicksort algorithm for pre-processing, instead of using a merge sort in List.Sort, which requires an extra $O(n)$ memory. However, this is not an asymptotic difference, and Quicksort does have the drawback of having an $O(n^2)$ worst case time complexity, compared to merge sort being $O(n \log n)$ in all cases.