Summary

Time complexity: O(N log k), where N is the number of input words. We count the frequency of each word in O(N) time. Then we add N words to the heap, each in O(log k) time, where k is the number of frequent words we have to find. Finally, we pop from the heap up to k times. As k <= N this is O(N log k) in total.
Space Complexity: O(N), the space used to store N words in the hash table and the heap.

Problem Pattern

1. First, you need to find the frequency of each word. What's the best way you can think of to do this? Usually, whenever we need to find the frequency of something, using Hashmaps works most efficiently.
2. Now, the most important thing in this question is, if 2 words have the same frequency, then you need to return considering the lexicographical order, to pick your top K elements. Is a hashmap enough for this? No.
3. Is there any data structure that stores elements in a sorted way? Yes, a priority queue or a heap.
4. Who decides how the elements are sorted? Can you define this according to your need? Think about how you will need to store words in your heap.

Problem Approach

1. Count the frequency of each word, and store it in a hashmap.
2. Now we need to create a heap. What will be the size of this heap? We will store only k best candidates in the heap. This also suggests that we will store the worst candidate at the top, so that we can just compare an incoming candidate with the top element of the heap and decide if a swap is required.
3. How do you define the best candidate though? Here, 'best' is defined with our custom ordering relation or custom comparator, which puts the worst candidates at the top of the heap. At the end, we pop off the heap up to k times and reverse the result so that the best candidates are first.
4. What will the custom comparator that we use for adding elements to our heap do? It needs to do 2 things, if 2 words have the same frequency, it will return the word with lexicographical precedence. And if not, it will return the word with greater frequency.

Problem Pseudocode

```
List<String> topKFrequent(String[] words, int k) {
    new Map<String, Integer> count
    for (String word in words) {
        count.put(word, count.getOrDefault(word, 0) + 1)
    }

    PriorityQueue<String> heap = new PriorityQueue<String>(
            (w1, w2) -> count.get(w1).equals(count.get(w2)) ?
            w2.compareTo(w1) : count.get(w1) - count.get(w2) )

    for (String word: count.keySet()) {
        heap.offer(word)
        if (heap.size() > k) heap.poll()
    }

    new List<String> ans
    while (!heap.isEmpty())
        ans.add(heap.poll())
    Collections.reverse(ans)
    return ans
}
```

Alternate Approaches

1. Count the frequency of each word, and sort the words with a custom ordering relation that uses these frequencies. Then take the best k of them. The complexity of this will be slighter higher due to the sorting. Heap is more efficient.
2. We can use Trie and Min Heap to get the k most frequent words efficiently. The idea is to use Trie for searching existing words and adding new words efficiently. Trie also stores count of occurrences of words. A Min Heap of size k is used to keep track of k most frequent words at any point of time.

Trie and Min Heap are linked with each other by storing an additional field in Trie ‘indexMinHeap’ and a pointer ‘trNode’ in Min Heap. The value of ‘indexMinHeap’ is maintained as -1 for the words which are currently not in Min Heap (or currently not among the top k frequent words). For the words which are present in Min Heap, ‘indexMinHeap’ contains, index of the word in Min Heap. The pointer ‘trNode’ in Min Heap points to the leaf node corresponding to the word in Trie. This approach is ideal of the size of array is very large or you need to apply this say for all words in a book
Reference - Find the k most frequent words from a file