CM1210 Java Spring Semester Coursework – C22026075

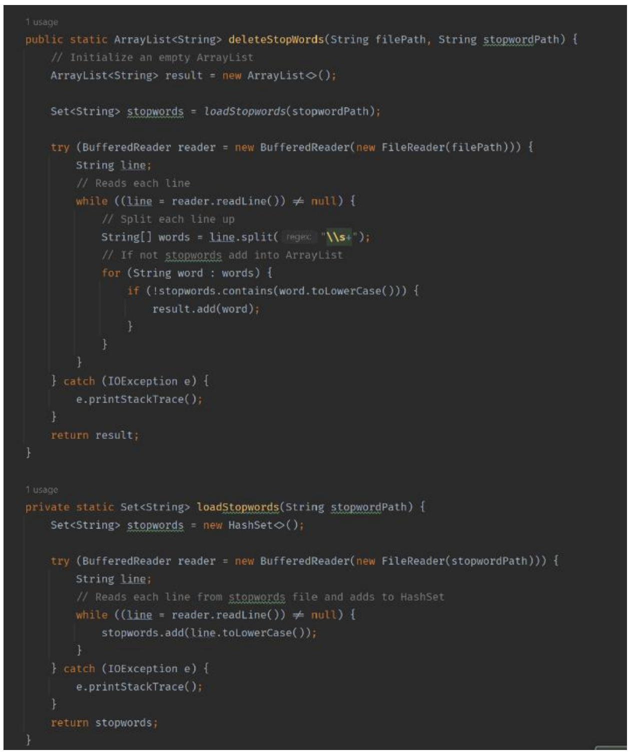
Section 1 – Reading in stopWords –

To complete the task of removing stop words from an input file and returning the remaining words as an ArrayList, I split the code into two methods - loadStopWords and deleteStopWords. This makes the code more modular and easier to read and test

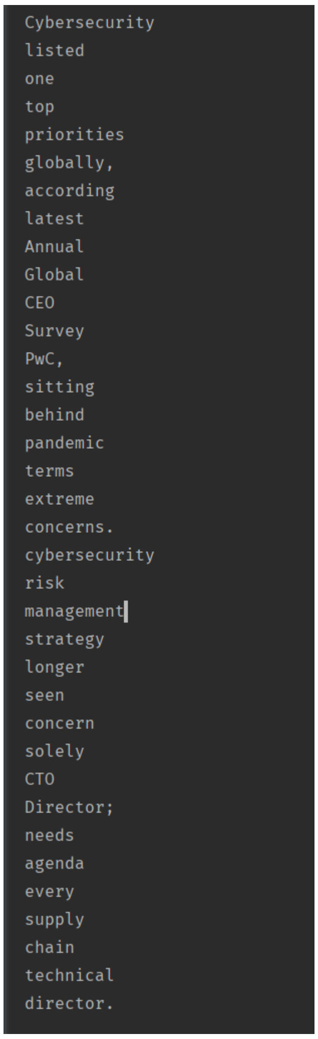
In the loadStopWords method, I used a BufferedReader to read the stop word file line by line and converted each line to lowercase. I then added each stop word to a HashSet to ensure uniqueness and case-insensitivity, making the program more robust. This approach is efficient as it avoids reading the entire file into memory at once .

The deleteStopWords method first calls the loadStopWords method to load in the stop words. I used an ArrayList to store the non-stop words and a HashSet to store the stop words. The HashSet provides fast search times and avoids the need for duplicate checks; using a HashSet means the retrieval of stop words is efficient as it has a constant time complexity of O(1) I used a try-catch statement to ensure efficient resource management.

To improve efficiency, I could have used a tree to retrieve the stop words as it has a more efficient time complexity of O(X) where X is the length of the string.



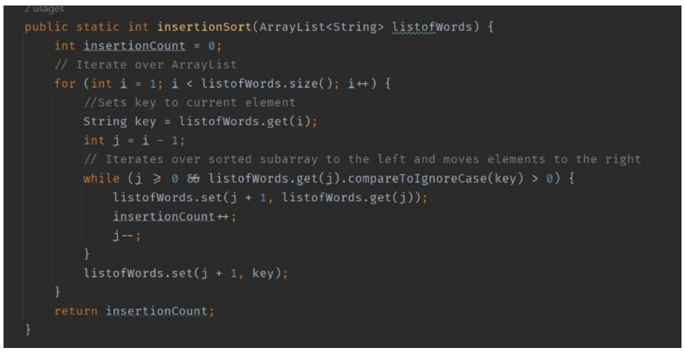
```java
1 usage
public static ArrayList<String> deleteStopWords(String filePath, String stopwordPath) {
    // Initialize an empty ArrayList
    ArrayList<String> result = new ArrayList<>();

    Set<String> stopwords = loadStopwords(stopwordPath);

    try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
        String line;
        // Reads each line
        while ((line = reader.readLine()) ≠ null) {
            // Split each line up
            String[] words = line.split( regex: "\\s+");
            // If not stopwords add into ArrayList
            for (String word : words) {
                if (!stopwords.contains(word.toLowerCase())) {
                    result.add(word);
                }
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return result;
}

1 usage
private static Set<String> loadStopwords(String stopwordPath) {
    Set<String> stopwords = new HashSet<>();

    try (BufferedReader reader = new BufferedReader(new FileReader(stopwordPath))) {
        String line;
        // Reads each line from stopwords file and adds to HashSet
        while ((line = reader.readLine()) ≠ null) {
            stopwords.add(line.toLowerCase());
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return stopwords;
}
```



```
Cybersecurity
listed
one
top
priorities
globally,
according
latest
Annual
Global
CEO
Survey
PwC,
sitting
behind
pandemic
terms
extreme
concerns.
cybersecurity
risk
management
strategy
longer
seen
concern
solely
CTO
Director;
needs
agenda
every
supply
chain
technical
director.
```

Section 2 – Insertion Sort –

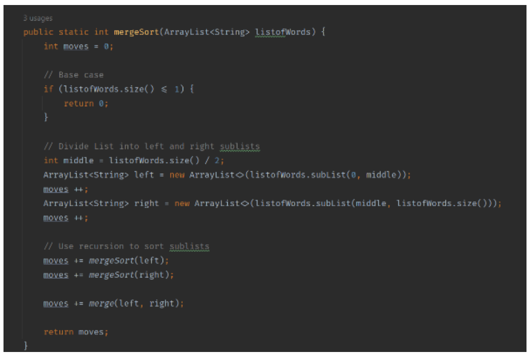
I was tasked with implementing an insertion sort to sort the words obtained from Q1. The implementation of my insertion sort uses a loop to iterate over the ArrayList, a nested loop to compare the current element to the sorted subarray and the ‘set’ method to move elements to the right. I also decided to keep track of the number of swaps made which is useful for analyzing the efficiency of the sorting algorithm. An insertion sort is an effective choice for small input sizes as it has a worst-case time complexity of O(n^2) meaning it grows in proportion to the square of the size of the input. For large input sizes, a more efficient sorting algorithm such as merge sort or quicksort may be better.



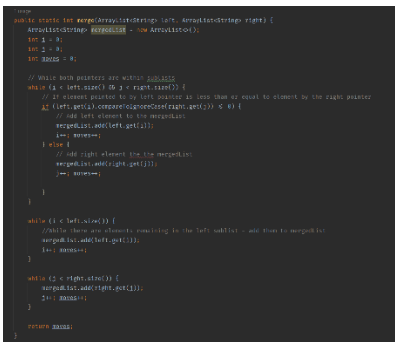
```
2 usages
public static int insertionSort(ArrayList<String> listofWords) {
    int insertionCount = 0;
    // Iterate over ArrayList
    for (int i = 1; i < listofWords.size(); i++) {
        //Sets key to current element
        String key = listofWords.get(i);
        int j = i - 1;
        // Iterates over sorted subarray to the left and moves elements to the right
        while (j >= 0 && listofWords.get(j).compareToIgnoreCase(key) > 0) {
            listofWords.set(j + 1, listofWords.get(j));
            insertionCount++;
            j--;
        }
        listofWords.set(j + 1, key);
    }
    return insertionCount;
}
```

Section 3 – Merge Sort -

I had to implement a merge sort for the same purpose as Q3. The merge sort recursively divides the ArrayList into two sublists until each sublist contains only one element. It then merges the sublists back together in a sorted order using the merge method I created. The merge method takes two sorted ArrayLists and merges them into a single list; to do this it uses pointers to track the current index of the left and right sublists. Both the mergesort and merge method return the number of swaps made during the process which will be used later. The merge sort has a time complexity of O(n log n) making it more efficient for larger input sizes. By using recursion to divide and conquer the input list it is a more robust and efficient way of sorting large data sets. Furthermore, using an ArrayList allows for ease of implementation



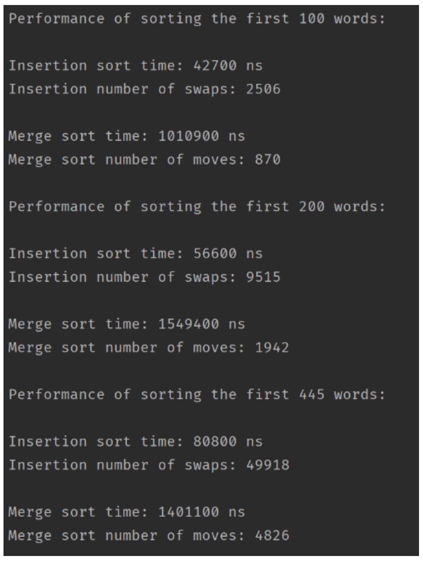
```
3 usages
public static int mergeSort(ArrayList<String> listofWords) {
    int moves = 0;

    // Base case
    if (listofWords.size() <= 1) {
        return 0;
    }

    // Divide List into left and right sublists
    int middle = listofWords.size() / 2;
    ArrayList<String> left = new ArrayList<>(listofWords.subList(0, middle));
    moves ++;
    ArrayList<String> right = new ArrayList<>(listofWords.subList(middle, listofWords.size()));
    moves ++;

    // Use recursion to sort sublists
    moves += mergeSort(left);
    moves += mergeSort(right);

    moves += merge(left, right);

    return moves;
}
```



```
1 usage
public static int merge(ArrayList<String> left, ArrayList<String> right) {
    ArrayList<String> mergedList = new ArrayList<>();
    int i = 0;
    int j = 0;
    int moves = 0;

    // While both pointers are within sublists
    while (i < left.size() && j < right.size()) {
        // If element pointed to by left pointer is less than or equal to element by the right pointer
        if (left.get(i).compareToIgnoreCase(right.get(j)) <= 0) {
            // Add left element to the mergedList
            mergedList.add(left.get(i));
            i++; moves++;
        } else {
            // Add right element the the mergedList
            mergedList.add(right.get(j));
            j++; moves++;

        }
    }

    while (i < left.size()) {
        //While there are elements remaining in the left sublist - add them to mergedList
        mergedList.add(left.get(i));
        i++; moves++;
    }

    while (j < right.size()) {
        mergedList.add(right.get(j));
        j++; moves++;
    }

    return moves;
}
```

Section 4 – Measuring Sort Performance –

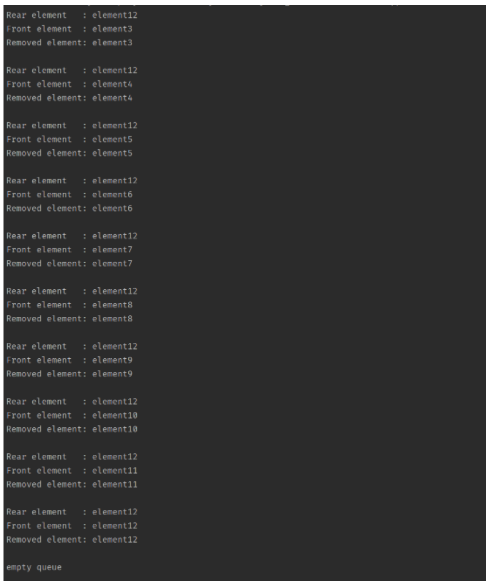
Question 4 required me to compare the difference between sort methods when sorting the input.txt. I created a method where the ArrayList of sorted words are passed in along with the number of words that you want to be sorted. I did this because the question required me to compare the first 100,200 and 500 words and the time taken to be sorted. By writing this in a method it promotes reusable code as I can enter any number of words I want to be compared. I then use the system.nanoTime method to measure the time taken between the start of each sort and the end. I then outputted the time taken and then the number of swaps for the first 100 words, the first 200 words and the whole length of the ArrayList. I was not able to compare 500 words due to the fact that after the stopwords were removed, the ArrayList did not contain 500 words; so instead I compared the whole ArrayList



```
Performance of sorting the first 100 words:

Insertion sort time: 42700 ns
Insertion number of swaps: 2506

Merge sort time: 1010900 ns
Merge sort number of moves: 870

Performance of sorting the first 200 words:

Insertion sort time: 56600 ns
Insertion number of swaps: 9515

Merge sort time: 1549400 ns
Merge sort number of moves: 1942

Performance of sorting the first 445 words:

Insertion sort time: 80800 ns
Insertion number of swaps: 49918

Merge sort time: 1401100 ns
Merge sort number of moves: 4826
```

Section 5 – Circular Queue Implementation

I had to implement the enqueue method and dequeue methods for a circular queue. This was simple to achieve as I have worked on queues previously. The most difficult part to achieve was making sure the size of the queue is doubled if the queue is full; this was an easy fix as I created a new queue with double the length, then copied the elements into this new queue and updated the front and rear pointers. The algorithm used in my implementation has a time complexity of O(1) for both methods apart from when the queue needs to be resized in the enqueue method; in that case the time complexity becomes O(n) because the elements need to be copied across.



```
Rear element   : element12
Front element  : element3
Removed element: element3

Rear element   : element12
Front element  : element4
Removed element: element4

Rear element   : element12
Front element  : element5
Removed element: element5

Rear element   : element12
Front element  : element6
Removed element: element6

Rear element   : element12
Front element  : element7
Removed element: element7

Rear element   : element12
Front element  : element8
Removed element: element8

Rear element   : element12
Front element  : element9
Removed element: element9

Rear element   : element12
Front element  : element10
Removed element: element10

Rear element   : element12
Front element  : element11
Removed element: element11

Rear element   : element12
Front element  : element12
Removed element: element12

empty queue
```