Practical implementation of algorithms and data structures

Introduction

Presented in the following report will be a “Parse(data)” method that can retrieve words through a text file and put them in an Array List. Also, the Bubble Sort algorithm and Merge sort algorithm will be used showing the difference and efficiency between them. Finally, an explanation of the 2 methods “addAtPosition” and “deleteAtPosition” found in the given “MyLinkedList.java” file.

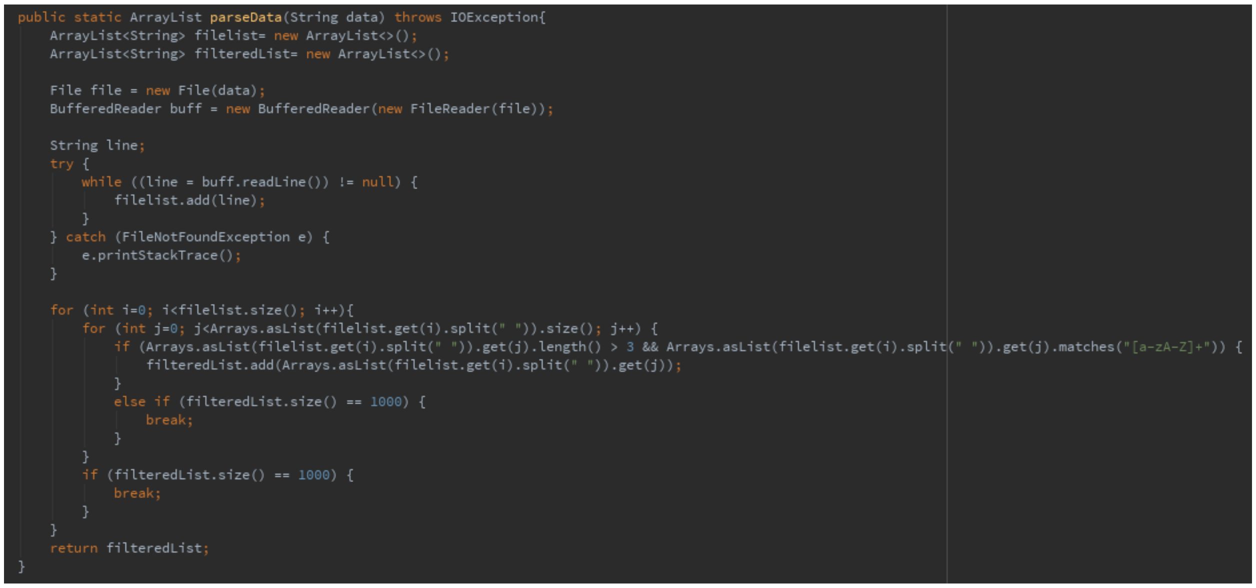
Bubble Sort

Bubble sort may be one of the least efficient algorithms for sorting but can be easy to understand. They algorithm compares items 1 and 2 and exchanges if necessary, then compares 2 and 3 and so on. It passes through all the elements in sequence and at the end of the list they will all be sorted.

Merge Sort

Merge sort basically divides the list into subsets recursively until every element is its own subset. When it sorts the last 2, it moves upwards sorting the next subset, until all the subsets are sorted putting the two halves together and returning a sorted list.

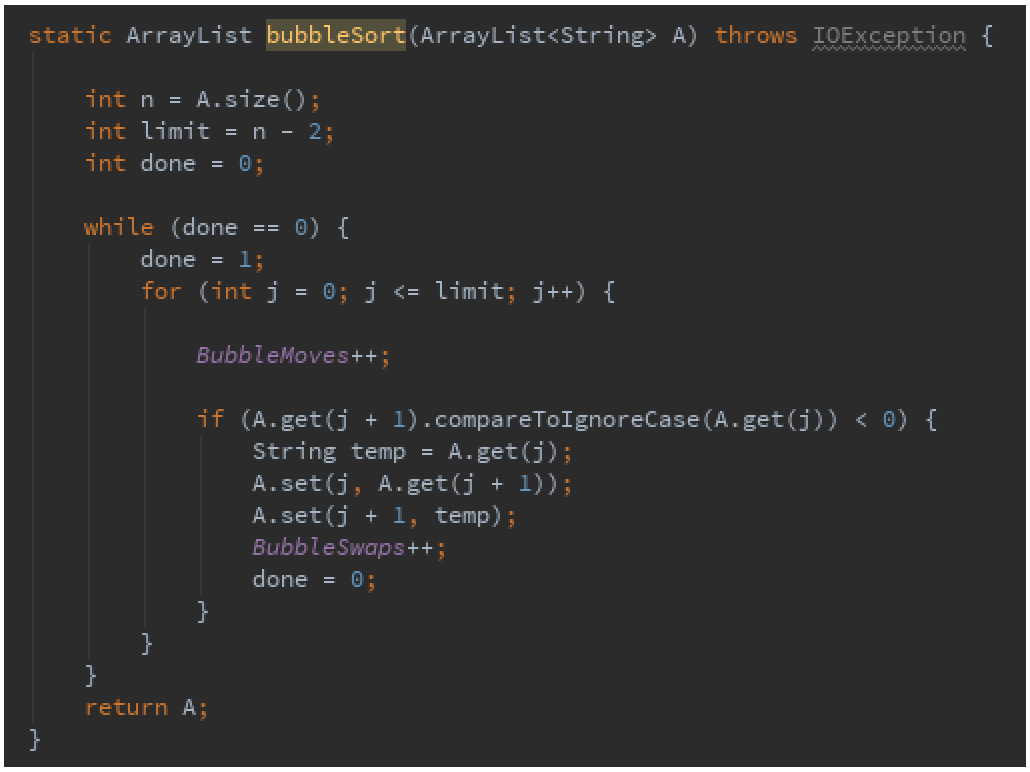
parseData(data) method



```java
public static ArrayList parseData(String data) throws IOException{
    ArrayList<String> filelist= new ArrayList<>();
    ArrayList<String> filteredList= new ArrayList<>();

    File file = new File(data);
    BufferedReader buff = new BufferedReader(new FileReader(file));

    String line;
    try {
        while ((line = buff.readLine()) != null) {
            filelist.add(line);
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }

    for (int i=0; i<filelist.size(); i++){
        for (int j=0; j<Arrays.asList(filelist.get(i).split(" ")).size(); j++) {
            if (Arrays.asList(filelist.get(i).split(" ")).get(j).length() > 3 && Arrays.asList(filelist.get(i).split(" ")).get(j).matches("[a-zA-Z]+")) {
                filteredList.add(Arrays.asList(filelist.get(i).split(" ")).get(j));
            }
            else if (filteredList.size() == 1000) {
                break;
            }
        }
        if (filteredList.size() == 1000) {
            break;
        }
    }
    return filteredList;
}
```

The "parseData(data)" method takes the name of the text file used "data.txt" as a parameter and uses it when it reads the file. A while loop reads line by line inside the text file using the buffer and stores all the contents in an array list called "filelist". Then we have a for loop that iterates through all the lines in the file and another for loop goes through all the words in the file splitting them with a (" ") delimiter. Then an if statement checks if the size of the word is bigger than 3 characters and checks if the word matches a regular expression for a word containing only characters [a-z]. If both the expressions pass the word is stored in another Array List called "filteredList". After that, it checks if the list has filled in with 1000 words and breaks exiting the inner for loop then another if statement breaks through the last for loop finally returning the "filteredList".

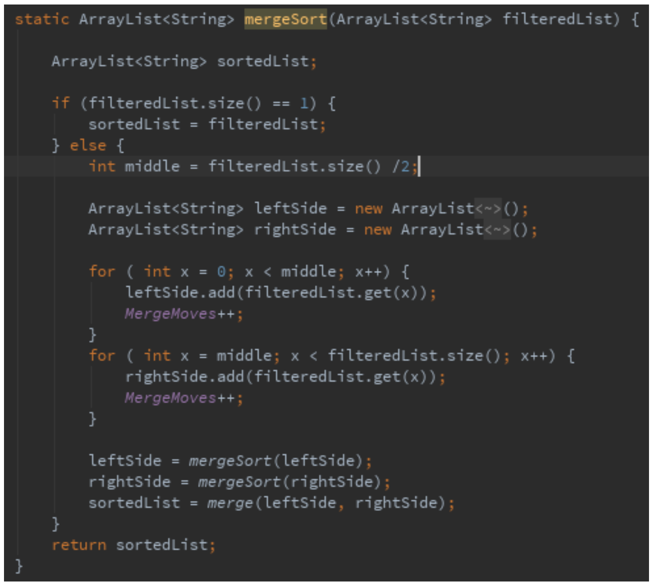
Bubble Sort Description



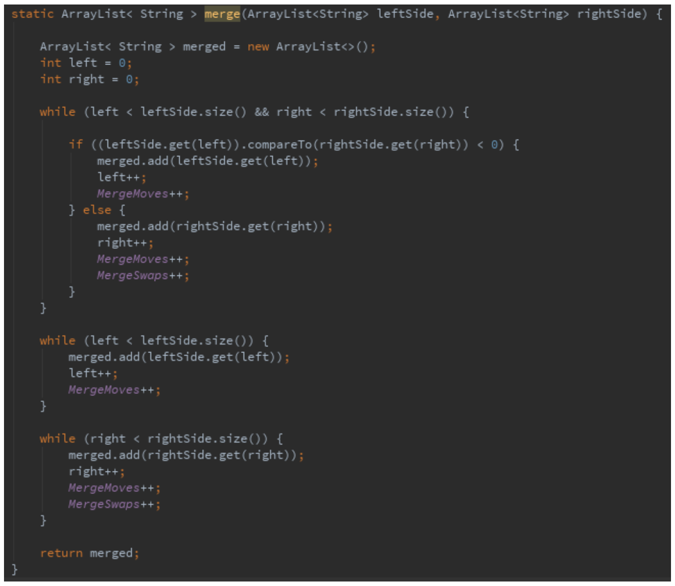
```java
static ArrayList bubbleSort(ArrayList<String> A) throws IOException {

    int n = A.size();
    int limit = n - 2;
    int done = 0;

    while (done == 0) {
        done = 1;
        for (int j = 0; j <= limit; j++) {

            BubbleMoves++;

            if (A.get(j + 1).compareToIgnoreCase(A.get(j)) < 0) {
                String temp = A.get(j);
                A.set(j, A.get(j + 1));
                A.set(j + 1, temp);
                BubbleSwaps++;
                done = 0;
            }
        }
    }
    return A;
}
```

The “bubbleSort(A)” takes an Array List as a parameter. This array list is filled with the 1000-word list created using the “parseData(data)” method. It creates a for loop that goes through all the element in the list sequentially and compares them. If the left side word (e.g. index(3)) is not in the position when sorting alphabetically with the index(4), index(3) is stored temporarily in a variable called “temp”, then the word in index(3) is replaced with the index(4) word. Finally, the index(4) word replaced with the word stored in the “temp” variable.

Merge Sort Description



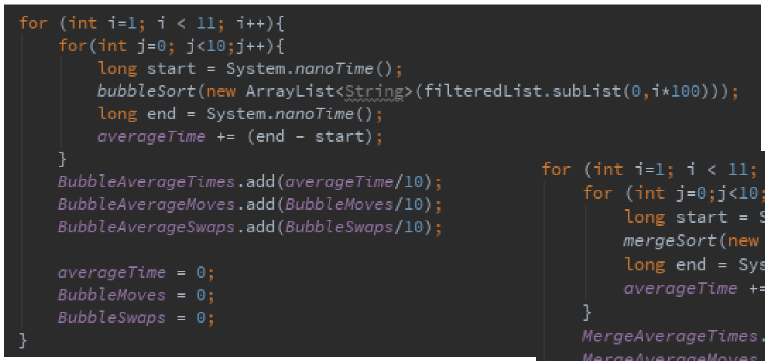
```java
static ArrayList<String> mergeSort(ArrayList<String> filteredList) {

    ArrayList<String> sortedList;

    if (filteredList.size() == 1) {
        sortedList = filteredList;
    } else {
        int middle = filteredList.size() /2;

        ArrayList<String> leftSide = new ArrayList<~>();
        ArrayList<String> rightSide = new ArrayList<~>();

        for ( int x = 0; x < middle; x++) {
            leftSide.add(filteredList.get(x));
            MergeMoves++;
        }
        for ( int x = middle; x < filteredList.size(); x++) {
            rightSide.add(filteredList.get(x));
            MergeMoves++;
        }

        leftSide = mergeSort(leftSide);
        rightSide = mergeSort(rightSide);
        sortedList = merge(leftSide, rightSide);
    }
    return sortedList;
}
```



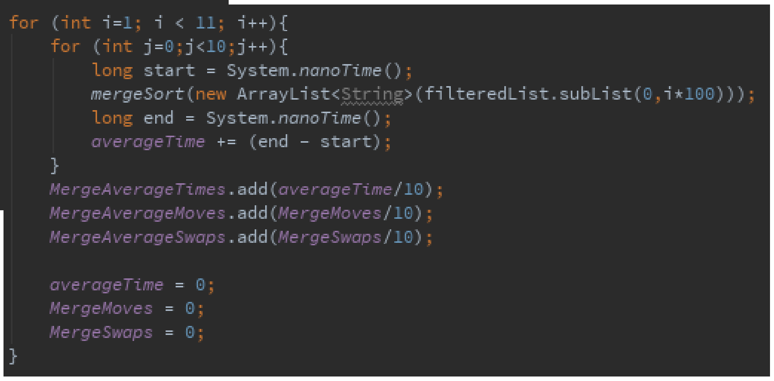
```java
static ArrayList< String > merge(ArrayList<String> leftSide, ArrayList<String> rightSide) {

    ArrayList< String > merged = new ArrayList<>();
    int left = 0;
    int right = 0;

    while (left < leftSide.size() && right < rightSide.size()) {

        if ((leftSide.get(left)).compareTo(rightSide.get(right)) < 0) {
            merged.add(leftSide.get(left));
            left++;
            MergeMoves++;
        } else {
            merged.add(rightSide.get(right));
            right++;
            MergeMoves++;
            MergeSwaps++;
        }
    }

    while (left < leftSide.size()) {
        merged.add(leftSide.get(left));
        left++;
        MergeMoves++;
    }

    while (right < rightSide.size()) {
        merged.add(rightSide.get(right));
        right++;
        MergeMoves++;
        MergeSwaps++;
    }

    return merged;
}
```

The “mergeSort(filteredList)” takes an Array List as a parameter. This array list is filled with the 1000-word list created using the “parseData(data)” method. Simple if statements check if the array list is one element. If not, it is split in the middle dividing the array list into the “leftSide” and “rightSide”. Then recursive calls are used splitting the sides into halves until all the elements are split into arrays of size 1. These arrays are then sorted between them by calling another method called “merge(leftSide, rightSide)”. This method “merge(leftSide, rightSide)” takes the 2 array lists and compares its contents, sorts them alphabetically then in the end merges and returns them together.

Algorithmic Timing Results

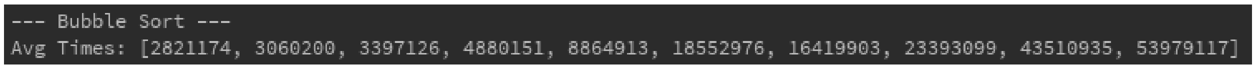


```
for (int i=1; i < 11; i++){
    for(int j=0; j<10;j++){
        long start = System.nanoTime();
        bubbleSort(new ArrayList<String>(filteredList.subList(0,i*100)));
        long end = System.nanoTime();
        averageTime += (end - start);
    }
    BubbleAverageTimes.add(averageTime/10);
    BubbleAverageMoves.add(BubbleMoves/10);
    BubbleAverageSwaps.add(BubbleSwaps/10);

    averageTime = 0;
    BubbleMoves = 0;
    BubbleSwaps = 0;
}
```

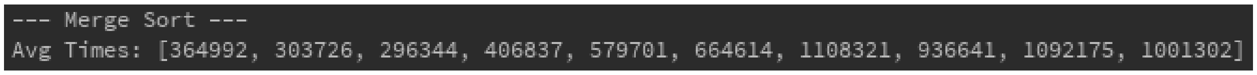


```
for (int i=1; i < 11; i++){
    for (int j=0;j<10;j++){
        long start = System.nanoTime();
        mergeSort(new ArrayList<String>(filteredList.subList(0,i*100)));
        long end = System.nanoTime();
        averageTime += (end - start);
    }
    MergeAverageTimes.add(averageTime/10);
    MergeAverageMoves.add(MergeMoves/10);
    MergeAverageSwaps.add(MergeSwaps/10);

    averageTime = 0;
    MergeMoves = 0;
    MergeSwaps = 0;
}
```

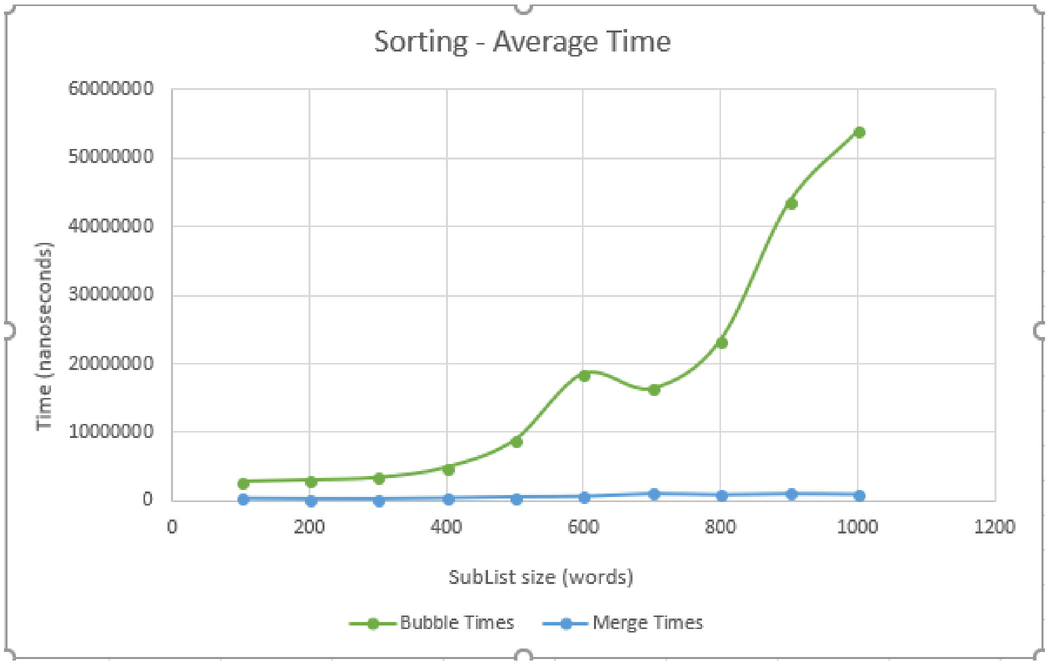
Both algorithms are run for sorting Array Lists of 100 words, 200 words and so on until the 1000 words. They are ran and timed 10 times for each and an average is stored in a list. In the end we have 2 lists of 10 long numbers that contain the average times. By using the results obtained we can create a graph that shows the efficiency of each of the both algorithms “Bubble Sort” and “Merge Sort” for the different sublists.



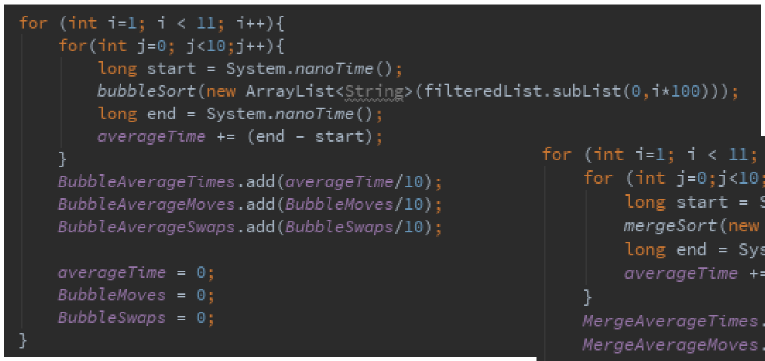
```
--- Bubble Sort ---
Avg Times: [2821174, 3060200, 3397126, 4880151, 8864913, 18552976, 16419903, 23393099, 43510935, 53979117]
```



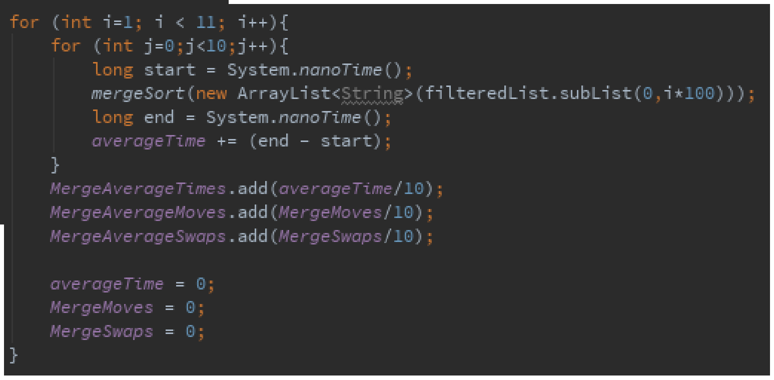
```
--- Merge Sort ---
Avg Times: [364992, 303726, 296344, 406837, 579701, 664614, 1108321, 936641, 1092175, 1001302]
```



Algorithmic Swap and Move Results

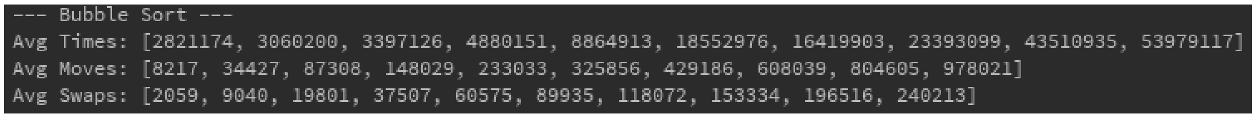


```
for (int i=1; i < 11; i++){
    for(int j=0; j<10;j++){
        long start = System.nanoTime();
        bubbleSort(new ArrayList<String>(filteredList.subList(0,i*100)));
        long end = System.nanoTime();
        averageTime += (end - start);
    }
    BubbleAverageTimes.add(averageTime/10);
    BubbleAverageMoves.add(BubbleMoves/10);
    BubbleAverageSwaps.add(BubbleSwaps/10);

    averageTime = 0;
    BubbleMoves = 0;
    BubbleSwaps = 0;
}
```

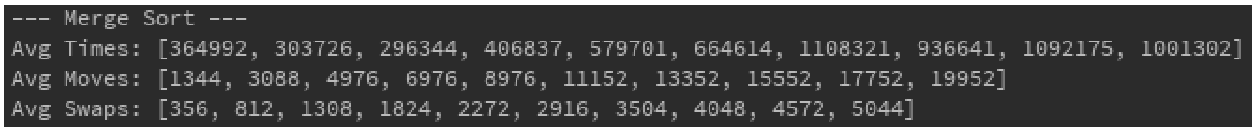


```
for (int i=1; i < 11; i++){
    for (int j=0;j<10;j++){
        long start = System.nanoTime();
        mergeSort(new ArrayList<String>(filteredList.subList(0,i*100)));
        long end = System.nanoTime();
        averageTime += (end - start);
    }
    MergeAverageTimes.add(averageTime/10);
    MergeAverageMoves.add(MergeMoves/10);
    MergeAverageSwaps.add(MergeSwaps/10);

    averageTime = 0;
    MergeMoves = 0;
    MergeSwaps = 0;
}
```

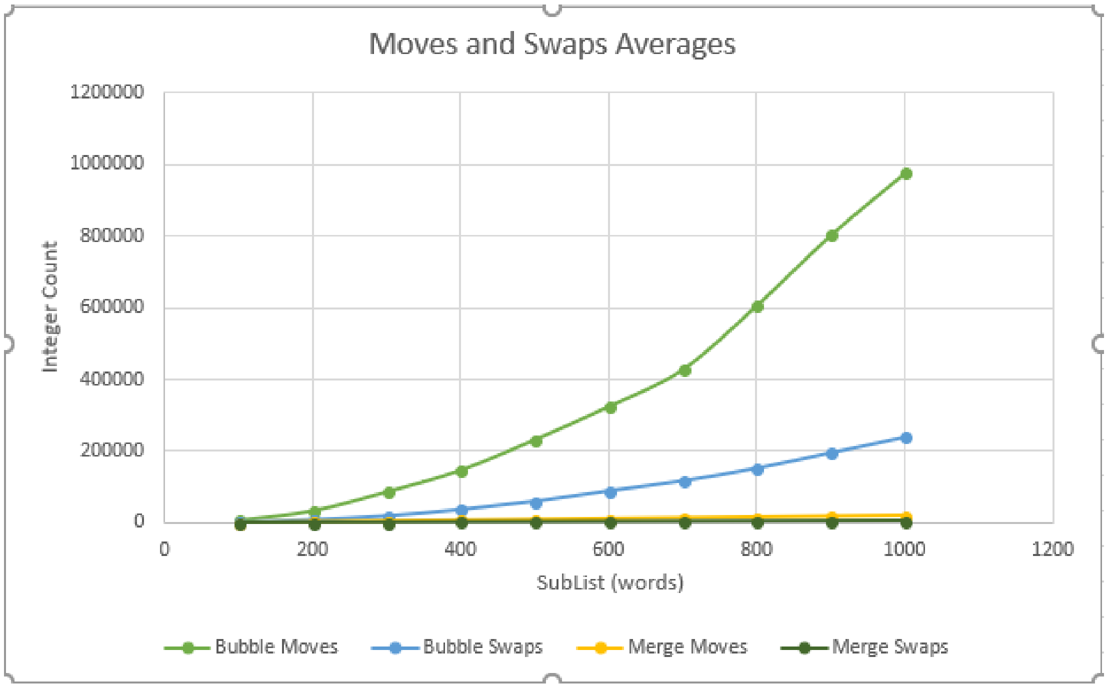
Like the timing average the procedure is used to also find an average for the number of moves and swaps that occur for the different subLists. The results are stored in array lists that can then be printed off and be used to create a graph to visualize again the efficiency of both algorithms.



```
--- Bubble Sort ---
Avg Times: [2821174, 3060200, 3397126, 4880151, 8864913, 18552976, 16419903, 23393099, 43510935, 53979117]
Avg Moves: [8217, 34427, 87308, 148029, 233033, 325856, 429186, 608039, 804605, 978021]
Avg Swaps: [2059, 9040, 19801, 37507, 60575, 89935, 118072, 153334, 196516, 240213]
```



```
--- Merge Sort ---
Avg Times: [364992, 303726, 296344, 406837, 579701, 664614, 1108321, 936641, 1092175, 1001302]
Avg Moves: [1344, 3088, 4976, 6976, 8976, 11152, 13352, 15552, 17752, 19952]
Avg Swaps: [356, 812, 1308, 1824, 2272, 2916, 3504, 4048, 4572, 5044]
```



Experiment Results

Timing Results

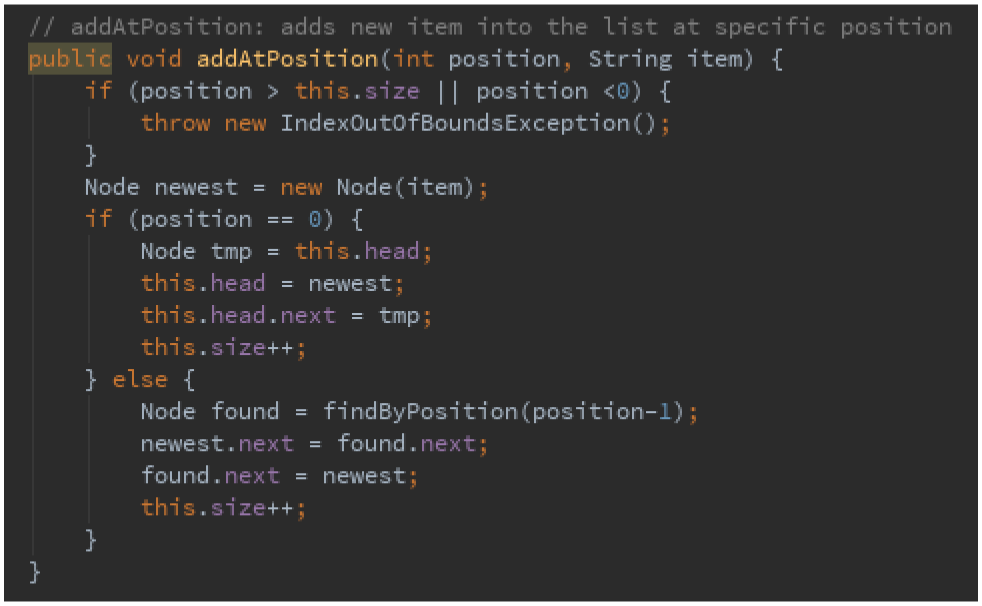
There is a huge difference in timing for both algorithms. As presented in the (Sorting – Average Time) graph above the Merge algorithm times were substantially less than the Bubble Sort algorithm. In all cases of the sublists the Merge Sort was always faster, even in the 1000-word sublist the difference between the 2 was gigantic. The Bubble Sort algorithm taking 53979117 nanoseconds whereas the Merge Sort only took 1001302 nanoseconds.

Moves and Swaps Results

As shown in the diagram above (Moves and Swaps Averages) the average results exported on a graph show a great difference in efficiency between the 2 algorithms with Merge Sort coming again superior. In the smaller sublists both algorithms gave similar results but as the number of words in the subLists increased a spike in the number of moves and swaps of the Bubble Sort algorithm has shown its inefficiency as an algorithm.

MyLinkedList Class and methods

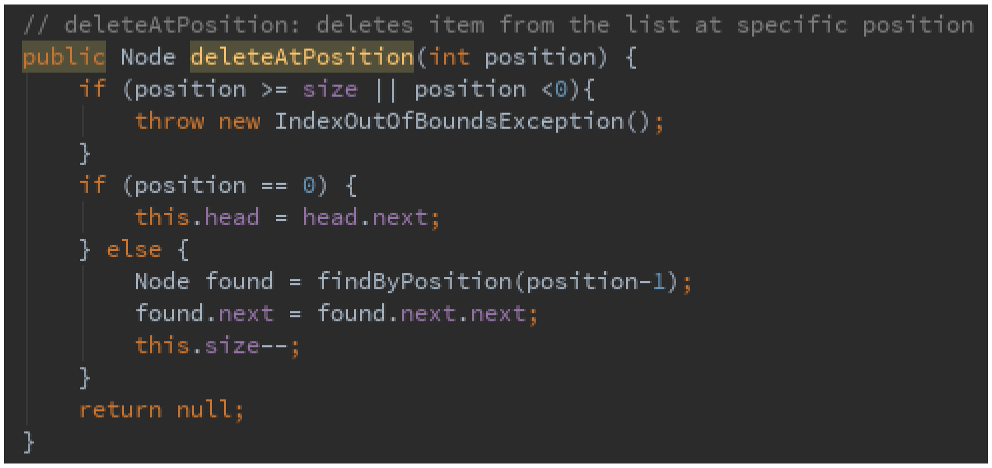
addAtPosition(int position, String item) method



```
// addAtPosition: adds new item into the list at specific position
public void addAtPosition(int position, String item) {
    if (position > this.size || position <0) {
        throw new IndexOutOfBoundsException();
    }
    Node newest = new Node(item);
    if (position == 0) {
        Node tmp = this.head;
        this.head = newest;
        this.head.next = tmp;
        this.size++;
    } else {
        Node found = findByPosition(position-1);
        newest.next = found.next;
        found.next = newest;
        this.size++;
    }
}
```

This method uses 2 parameters, the 1st is an integer for the position and the other is a String of the item being added in the linked list. First, it checks if the position entered is valid, else it throws an exception for out of bounds error. Following, it creates an instance Node for the item being entered. If the position is 0 the item being inserted will become the head of the linked list. Else it will be entered between the previous position and the next position, the previous will change its reference to the new item being added and the item will have its reference to the item that it took its place.

deleteAtPosition(int position) method



```java
// deleteAtPosition: deletes item from the list at specific position
public Node deleteAtPosition(int position) {
    if (position >= size || position <0){
        throw new IndexOutOfBoundsException();
    }
    if (position == 0) {
        this.head = head.next;
    } else {
        Node found = findByPosition(position-1);
        found.next = found.next.next;
        this.size--;
    }
    return null;
}
```

This method uses 1 parameter which is an integer for the position of the item that will be removed from the linked list. First, it checks if the position entered is valid, else it throws an exception for out of bounds error. Then it checks that if the position entered is 0 to assign then head to the 2nd item in the list. Else it finds the previous item in the list from the specified position and assigns its reference to the element after the position. This reassigning skips the reference from pointing to the item in that position removing it from the linked list.

Conclusion

Overall in this Report it is shown that both algorithms, Bubble Sort and Merge Sort can be used to sort array lists alphabetically but, by plotting the data found in graphs we can conclude that the Merge Sort is both faster and efficient with a significant difference to Bubble Sort. In addition, manipulating data in linked list be quick and easy as long as methods are set for carrying out the actions. All in all, implementing algorithms can solve a problem in hand but finding the better one can help in speed and efficiency but also, data structures are useful in manipulating different sorts of data.