1841485 - Marton Zeisler

Implementation of two sorting algorithms and the queue data structure

Introduction

This report starts with describing the implementation and analysing the efficiency and performance of two popular sorting algorithms, insertion sort and merge sort. Finally, the queue data structure will be examined and an explanation of the enqueue and dequeue algorithms will be given as well.

Insertion Sort

Insertion sort is one of the simplest and easiest sorting algorithm to use. The algorithm works the same way we sort playing cards in our hands. Insertion sort is based on the idea that one element from the input list of elements is consumed in each iteration to find the position to which it belongs in a sorted array.

Insertion sort can be used when the number of elements in a list is small, but it is not fast at all when it is used against a large set of data. Other sorting algorithms such as merge sort or quicksort perform much better on larger lists. It has an average and worst complexity of $\mathbf{n^2}$, which is considered a slow algorithm. However, the best case complexity is **n**, if the list is already sorted, in that case the algorithm only iterates the list once.

Here's my implementation of insertion sort:

```
public static void insertionSort(ArrayList<String> nonStopwords){
    int n = nonStopwords.size();

    for(int i=1; i<n; i++){
        String item = nonStopwords.get(i);
        int j = i-1;

        while(j>=0 && nonStopwords.get(j).compareTo(item) > 0 ){
            nonStopwords.set(j+1, nonStopwords.get(j));
            j = j-1;
        }

        nonStopwords.set(j+1, item);
    }
}
```

The function performs the insertion sort algorithm on the given ArrayList of elements of String type. The algorithm looks at each item and checks whether the adjacent element in the left side is alphabetically grater than the currently inspected item. If the current item comes first in the alphabet then it leaves the current element in its place and moves on to the next element. Otherwise, the algorithm finds the item's correct position in the sorted array and moves it to that position.

Merge Sort

Merge sort is slightly more complex than the previous sorting algorithm but it is much more efficient. It is a divide and conquer type of algorithm, which means that it works by breaking down the problem into smaller subproblems and then recursively solve these problems and combine the solutions at the end.

Merge sort keeps dividing the given list into subsets until only element is left in a subset. Then the algorithm sorts each of those halves, and then merges them back together at the end. The same sorting algorithm is applied to each of the divided subsets. It consists of two algorithms, one for merging the halves and one for sorting a given set.

Here's my implementation of merge sort:

```
public static void mergeSort(ArrayList<String> array){
    if(array.size() < 2){
        return;
    }

    int middle = array.size() / 2;

    ArrayList<String> left = new ArrayList<String>();
    for (int i = 0; i < middle; i++) {
        left.add(array.get(i));
    }

    ArrayList<String> right = new ArrayList<String>();
    for (int i = middle; i < array.size(); i++) {
        right.add(array.get(i));
    }

    mergeSort(left);
    mergeSort(right);

    merge(left, right, array);
}
```

```java
public static void merge(ArrayList<String> left, ArrayList<String> right, ArrayList<String> merged){
    int k = 0;

    while (left.size() != 0 && right.size() != 0) {

        if (left.get(0).compareTo(right.get(0)) < 0) {
            merged.set(k++, left.get(0));
            left.remove(0);
        }else{
            merged.set(k++, right.get(0));
            right.remove(0);
        }
    }

    while (left.size() != 0) {
        merged.set(k++, left.get(0));
        left.remove(0);
    }

    while (right.size() != 0) {
        merged.set(k++, right.get(0));
        right.remove(0);
    }
}
```

The first code snippet is responsible for breaking down the given array into two subsets. After the two subset arrays were created, it recursively attempts to perform another breakdown. If only one element is left in a a subset then the recursion will end and the algorithm will proceed with merging each pair of subsets using the merge function.

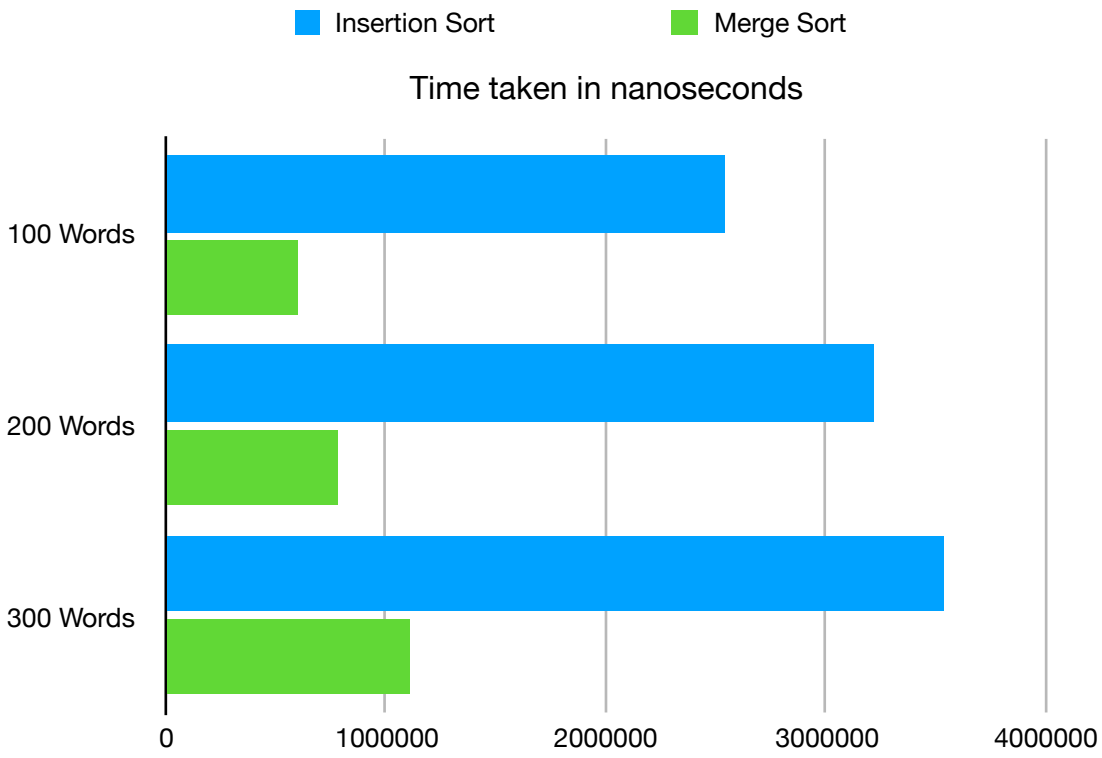
The merge function contains a while loop which will run until one of the subsets becomes empty. During this loop, the algorithm compares one item from each of the two subsets and places the alphabetically smaller value in the merged array. After this while loop is executed, the algorithm places any leftovers from either the left or the right subset into the merged array.

Merge sort has a worst, average and best complexity of **n*log(n)**, which is considered a very powerful and efficient algorithm among the sorting algorithms.

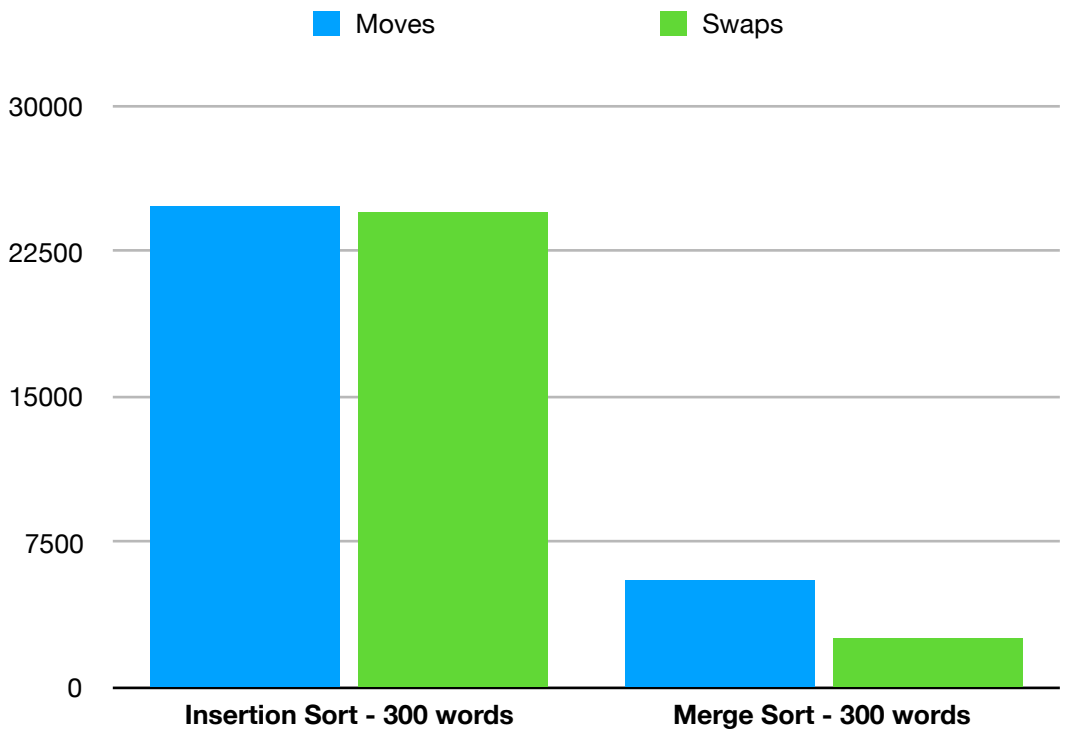
Comparison

After running both algorithms, the results are clear. The merge sort is much faster and efficient than the insertion sort. The merge sort performs significantly less iterations over the given array and also does fewer swaps compared to the insertion sort method.

The following chart compares the amount of time taken to run the two algorithms to sort three different sizes of lists of words.



The following chart compares the number of moves and swaps performed under each algorithm over an array containing three hundred words.



User Guide

There's two function calls in the main method of the Assignment.java file:

- *startInsertionSort(nonStopWords);*
- *startMergeSort(nonStopWords);*

By default both of the function calls will be executed when the Assignment.java file is compiled and run but feel free to comment out any of the two function calls to only run one sorting algorithm at a time.

Queue

A queue is a linear data structure and it's a first in first out (FIFO) type of data structure, which means that the first item that gets inserted is the first that gets removed as well. Queue is open at both its ends. The front end is always used for removing data(dequeue) and the other end is used for adding new elements to the queue. However, both ends can be accessed by the user to retrieve the first and last element of the queue.

Here's my implementation for the enqueue() method which adds a new element to the end of the queue:

```
public void enqueue(Object theElement)
{
   int currentFront = front % queue.length;
   int newRear = (rear+1) % queue.length;

   if(currentFront == newRear){
      // Queue would be full so let's duplicate the array
      Object newQueue[] = new Object[queue.length*2];
      int newQueueCounter = 0;

      for (int i = 0; i < queue.length-1; i++) {
         newQueueCounter++;
         newQueue[newQueueCounter] = getFrontElement();
         front = (front + 1) % queue.length;
      }

      queue = newQueue;
      front = 0;
      newQueueCounter++;
      newQueue[newQueueCounter] = theElement;
      rear = newQueueCounter;

   }else{
      // There is space for insertion
      rear = newRear;
      queue[rear] = theElement;
   }
}
```

The function starts with calculating the current front and the new rear indexes and checks if the two indexes are equal. If they are not equal, then that means the queue will not be full after the insertion so the function just updates the rear index and sets the rear of the queue to the new element.

If the two indexes are equal, then it means that the queue will be full after insertion and in order to keep operating the queue, the algorithm needs to double the size of the queue's capacity first before it can add the new element. The function creates a new queue with a two times bigger capacity and copies every element from the current queue to the new one. After all the elements were moved, the algorithm updates the front and rear indexes and finally adds the new element to the queue.

Here's my implementation for the dequeue() method which removes an element from the front of the queue:

```
public Object dequeue()
{
   if(isEmpty()){
      System.out.println("Queue is empty, cannot dequeue!");
      return null;
   }else{
      Object removed = getFrontElement();
      queue[(front + 1) % queue.length] = null;
      front++;
      return removed;
   }

}
```

The method first checks if the queue is empty. If it is empty, then no element can be removed and it will display an error message and return null. If the queue is not empty, then the program retrieves the front element and stores it in a variable so it can be returned at the end of the function. The function sets the front element of the queue to null and updates the front index of the queue.