QuickSort

Ruxandra-Stefania Tudose

Fall 2023

Introduction

This report aims to make use of two previously used types of data structures, namely the array and the simple linked list in order to compare and contrast the execution of the QuickSort algorithm run on a benchmark.

Implementing QuickSort

As far as I am concerned, the concept of the QuickSort implementation is pretty similar to MergeSort. However, this time we no longer first split the array to make use of recursion, but pick a pivot (I have chosen the last element of the data structure) and separate all elements smaller than it to the left and the greater ones to the right. Last but not least, the pivot is moved to its right place in the array and then recursion makes its presence felt again by applying the same process described above for the all elements until the pivot and for those after it. After having understood the concept of the algorithm, my expectation was that its time complexity regardless of its implementation (in an array or linked list) will be similar to merge sort, namely *O(n*log n)*.

Getting ready to benchmark

The first step in getting ready for the benchmark was setting up the two different implementations in order to run QuickSort and the linked list, was by far the most difficult one since it took some time until I understood how the separation between smaller and greater elements compared to the pivot is done, by simultaneously not cloning the nodes. In other words, it was simply a matter of having two poiners, reffering to the same elements. It should be taken into consideration that I have made use of my implementation of the simple linked list and added an extra node that always keeps track of the last inserted element in order to quickly identify the pivot.

Below, the start of the QuickSort method has been included to show how the two new lists are initialised.

```
public void QuickSort() {
    if(first.next != null) {
    //initialising a new list for smaller elements
        LinkedSimple small = new LinkedSimple();

    //initialising a new list for larger elements
        LinkedSimple large = new LinkedSimple();

    //copying the reference of the main lists's first node
        Node index = first;

    //copying the reference of the main list's last node (the pivot)
        Node pivot = last;
        ...
}
```

After that, grouping the elements into the smaller and greater ones compared to the chosen pivot is just a matter of traversing the main list as done before and comparing the stored value to the pivot's one. If the value is lower than the pivot's, the node is added to the the list keeping track of the small elements, otherwise it is included in the one keeping track of the large ones.

Note: The addition of the node to the correct list is done using the **addNode** method, which simply connects the reference of the previous last element to the new one and updates the new last element.

```
        while(index.number != pivot.number) {
            if(index.number < pivot.number)
                small.addNode(index); //example of how addNode is called
                         ...

        public void addNode(Node index) {
            if(first == null)
                first = index;
            else
                last.next = index;
            last = index;
        }
```

After the distribution is done correctly, as long as there is an element in the sublists, the QuickSort method is once again called. This is precisely when we make use of recursion. Below I have included how this applies for example to the list holding the reference to the nodes with lower values than the pivot's.

```
if(small.first != null) {
    small.QuickSort(); //the recursive step
    first = small.first; //update the main list's new 'first' reference
    small.last.next = pivot; //insert the pivot
}
```

Running the benchmark

Before jumping into the benchmark results, it should be taken into consideration that the algorithm I implemented for the QuickSort in the linked list doesn't handle duplicates, so I had to make sure that when I populate it using an array, no element appeared twice. In other words, I wrote a method that generates an ordered array and then swaps each element with another one, placed on a randomly generated index.

size	QuickSortArray[us]	QuickSortLinked[us]	ratio
25	11	25	2.2
50	12	38	3.2
100	29	102	3.5
200	66	91	1.3
400	118	62	0.5
800	74	116	0.5
1600	125	293	2.3
3200	273	596	2.1
6400	597	1253	2.1
12800	778	1677	2.1
25600	1697	4195	2.4
51200	3311	8735	2.6
102400	7085	20458	2.8
204800	14607	45505	3.1
409600	31998	115368	3.6

Table 1: The QuickSort benchmark results.

Note: The ratio in Table 1 is calculated as the ratio between the execution of QuickSort in an array over the one in a simple linked list.

As far as the benchmark is concerned, this is by far the most extensive one I have run so far since I went up until a total of **409600 elements**. The reason why I did that was the fact that I wanted to see if the behaviour of the execution time results were consistent as 'n' increased. The benchmark execution took approximately ***four and a half minutes*** and that is precisely why, unlike previous assignments, the numbers presented above were obtained as a result of *one try only*.

As it can be noticed, in the vast majority of the cases (except for the size = 400 elements), the array by far outperforms the simple linked list and according to my calculations it is on average ***2 times faster***. As the size doubles, mostly so does the execution time (with exceptions). However, since it doesn't exactly double, we therefore conclude that as expected the numbers back up the assumption that the algorithm is in both situations of complexity *O(n*log n)*. Moreover, the array becomes even more efficient as the size of the array grows, fact which can be noticed in the graph below displaying the ratio evolution.

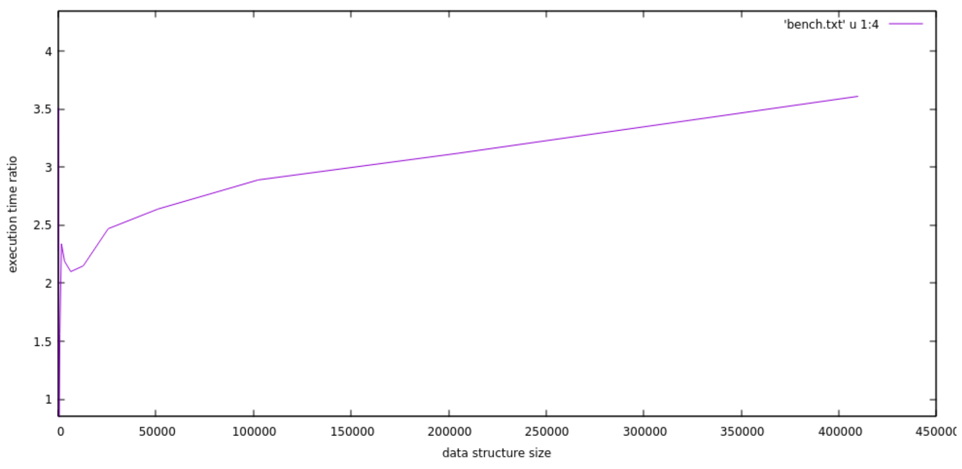

Figure 1: The graph describing the evolution of the ratio between executing QuickSort in an array and in a simple linked list.

I was in a way expecting the array to be more efficient than the linked list (maybe not as efficient as twice as faster) and that is because of the following two main reasons:

- unlike the optimised version of merge sort, there is no cloning of the initial array involved, therefore, time is saved; it's always a matter of simply choosing the new pivot and arranging the elements compared to its value (the lower ones to the left, the larger ones to the right) - no new array is allocated;

- there is less code that has to be executed, therefore less time spent evaluating the code lines compared to the linked list implementation, which has several additional checks that have to be executed;

On a different note, it should be taken into account that the linked list is also optimised since we always have a pointer referring to its last element, therefore we save time by avoiding always traversing the list all the way until the end.

Conclusion

All in all, executing QuickSort in an array definitely outperforms the linked list one. This has also been an interesting experiment especially because I have extended the interval of my benchmark (up until 409600) and because of the way we were indicated in the instructions to keep track of the nodes holding the smaller and larger values in two different lists.