Heaps

Ruxandra-Stefania Tudose

Fall 2023

Introduction

This report aims to introduce a new way of storing data sets under the form of a priority queue, namely first as a sorted linked list and then as an unsorted one. In addition, this experiment will also explore the implementation of a heap as binary tree and as an array.

Comparing two different priority queues

In this assignment, two priority queues have been implemented:

1. one whose add method is O(1) and whose remove is O(n): stored under the form of an unsorted simple linked list;
2. one whose add method is O(n) and whose remove method is O(1): stored under the form of a sorted simple linked list;

The most tricky part in my implementation of these methods was the 'remove' method for the unsorted simple linked list since what I did was to traverse the linked list once and do two simultaneous operations: identify the smallest priority in order to return it and also delete that specific node by keeping track of its previous neighbour. In other words, after the 'while' loop, I simply reestablished the connections. There were of course certain special cases such as when there is only one element in the priority queue or when the node that had to be deleted was precisely the head.

```
public Integer remove() {
        if(head == null) //special case empty heap
            return null;
        if (head.next == null ){ //special case - one element only in the heap
            int d = head.item;
            head = null;
            return d;
        }
```

```
        Node remove = null; //node to keep track of what to remove
        //node to keep track of the neighbour of the removed element
        Node neighbour_remove = null;

        Node index = head;
        Node previous = null;
        int c = index.item;
        while(index != null) {
            if(index.item < c) { //check if a lower priority was found
                c = index.item;
                remove = index; //updating the node that could be removed
                neighbour_remove = previous; //updating the previous neighbour
            }

            previous = index;
            index = index.next;

        }
        if(c == head.item) { //check if the head is the one to be removed
            head = head.next;
            return c;
        }

     //remove the identified node (redo the connections)
        neighbour_remove.next = remove.next;
        return c; //return the smallest identified priority
    }
```

Since both implementations have one O(1) and one O(n) method, by combining them, the behaviour should be O(n) as well, idea which is highlighted in the presented results and which become more obvious after the lists have more than 1600 elements (after the line in the middle). In other words, from that point, in both types of priority queues the execution time kind of doubles with the size.

As it can be noticed in *Table 1*, the sorted priority queue undoubtedly outperforms the unsorted one. However, as far as the sorted priority queue is concerned, I was honestly not expecting a logarithmic behaviour at the beginning of the benchmark, which I assume is the way it is either because of the caches or because of certain lucky situations when the element that had to be added is placed quickly, immediately after the 'head'.

size	SortedPriorQueue[ns]	UnsortedPriorQueue[ns]
100	102	502
200	94	887
400	103	1656
800	106	1450
1600	2062	2776
3200	4511	5462
6400	10713	10775
12800	24246	28147
25600	48184	55618
51200	97659	109589

Table 1: Benchmark analysis of the two priority queue implemenatation.

On the other hand, the O(n) behaviour is quite visible from the every beginning as far as the unsorted linked priority queue is concerned. By and large, according to my benchmark, on average, the sorted priority simple linked queue is *0.8 times faster*.

A new method for the binary tree heap: the push method

The starting point of creating this method was the implementation of the heap under the form of a binary tree. Both methods (add and remove) make use of the recursive step. One thing I found a bit tricky to understand at first was the way we should balance the tree by using a variable that stores the size of the node's subtree.

In order to implement the *push* method, I decided to first call it outside the Node class and then run recursively with a different method inside it. At first, outside the class the method simply returns -1 if the heap is empty, otherwise it increments the root given the provided parameter. The root then calls the method inside the Node class by also sending the 'depth' parameter, which in the end will be returned. The implemented method is as follows:

```
public int push(int depth) {
    depth++; //incrementing the depth every time we enter the method

    if (this.left == null || this.right == null) //reaching the end of the heap
        return depth;
```

```
    if (this.prio > this.left.prio) {//condition to enter the left brench
        int temp = this.prio; //swapping values
        this.prio = this.left.prio;
        this.left.prio = temp;
        depth = this.left.push(depth); //the recursive step
    }

    if (this.prio > this.right.prio){ //condition to enter the right brench
        int temp = this.prio; //swapping values
        this.prio = this.right.prio;
        this.right.prio = temp;
        depth = this.right.push(depth); //the recursive step
    }

    return depth; //returning the final depth after all the recursive steps
}
```

Note: One mistake I initially did was not incrementing the 'depth' index in the recursive step. I forgot that a new instance of the variable of this type is created for each method call, therefore I was always ending up with going down one level only (which was more than wrong). After having corrected the method as presented above, everything was fine. Looking back, I could have perhaps implemented a 'swap' method to shorten the method's code even more and to use it inside the if statements.

After having run the benchmark as indicated, I was pleasantly surprised to see that by using the push method, we save half the time we use in order to first remove, increase and then add the element with the previous created methods. It should be taken into account that these are the obtained numbers after having done 1000 tries, taken the minimum time and depth of having pushed/added 100 elements, whose total execution time I then used in order to get an average one (I divided it by 100).

AvgMinDepth[levels]	AvgMinTimePush[ns]	AvgMinTimeAddRem[ns]
13	90	163

Table 2: The 'push' method time analysis.

As it can be noticed the push method's average minimum execution time is *almost half* the other one and *on average we go down 13 levels* after having incremented the root.

The heap as an array

The implementation of a heap under the form of an array consists of two methods: the add method and the remove method.

The idea that stands behind the add method is to always add the new item at the end of the array and then traverse the whole heap and put the newly included element to its right position. In order to do so, one index starts from the end of the array and keeps on going until it reaches -1 (the start of the array). In each iteration we look for the first parent using the floor function formula and continue comparing the two values (the parent and the new item). Only if the new element is less than the parent, we then swap them. If the array's space runs out, we call the 'resize' function in order to extend it and allocate a large one.

The remove method is somewhat similar, however this time we return the array's first position, replace it with last value and then traverse the heap in order to arrange the element that had just been replaced correctly. Again, we have to find the parent and compare the values in order to ultimately swap if necessary. In the end we have to decrease the last element index and the total heap size. The code has also been tested separately before jumping into the benchmark.

size	SortedPriorQueue[ns]	UnsortedPriorQueue[ns]	HeapArray[ns]
100	102	502	471
200	94	887	220
400	103	1656	219
800	106	1450	154
1600	2062	2776	138
3200	4511	5462	131
6400	10713	10775	87
12800	24246	28147	98
25600	48184	55618	83
51200	97659	109589	88

Table 3: Benchmark analysis of the two priority queue implementation and the array heap.

Considering the benchmark, I was impressed by the results since I was not expecting for the array implementation to be this fast especially because there is copying involved whenever we have to extend the array (the execution times decrease as the sizes increase). My pure assumption is that the time is so low since compared to the linked lists implementations, less checks and conditions have to be verified, which make up for the time lost copying.

In addition, in the beginning the values are probably higher because the caches are not yet sufficiently warmed up in spite of the fact that I actually do a warm up before benchmarking.

Conclusion

All in all, this paper has explored four different implementations of the heap and judging by the results, turns out that the array implementation is the most efficient. Nevertheless, the 'push' method should also not be forgotten since it does help and gains speed by using it.