Double Linked Lists

Ruxandra-Stefania Tudose

Fall 2023

Introduction

This report aims to open a window on an 'updated' linked list, which includes several new features, among which the ability of traversing the data structure both forward and backwards. Therefore, the double linked list consists of an extra field in the cell, which always keeps track of the previous neighbour. The main focus of this experiment are the two new operations, namely *unlink* and *insert*, but most importantly how they behave depending on the type of the linked list and their sizes.

Note: This is the new report after having revisited my code, rerun my benchmark and taken into consideration the comments on my previous work.

Implementing a double linked list: the methods

In order to create the double linked list, the starting point was my implementaion of the simple linked list. In other words, I first added the new **previous** node and then, the only methods that had to be adapted were the *remove* and *asArray* methods. The second method, *asArray*, is a method I created in order to debug and see the changes to the list after having called any of the methods. Having said that, below there are the parts of the *remove* method that have been changed for the double linked list, which include connecting the references of all nodes to the to previous ones as well.

```
// code for special case - remove first element
if(previous.number == item) {
    first = first.next;
    first.prev = null; //extra for double linked list
}

while (current.next != null) {
    if(current.number == item) {
```

```
                previous.next = current.next;
                current.next.prev = previous; //extra for double linked list
            }

        //updating the references in order to move forward in the list
            previous = current;
            current = current.next;
        }
```

My approach: Checking accuracy first

As far as my approach is concerned, before moving on with the assignement, the next step was to check the accuracy of the new code upgrades. Therefore, in order to make sure that the previous cells are also connected, I navigated the list both forward and backwards using the 'length' method and displayed the size, which as expected was double to the real one. After I made sure it was working correctly , I commented these extra lines of code included below:

```
//navigating the double linked list backwards and incrementing 'count'
    while(index.prev != null) {
        count ++; //variable that keeps track of the list's size
        index = index.prev;
    }

    return count + 2; //'+2' to count the ends of the list
```

Setting up the *unlink* and *insert* methods

While the *insert* is pretty similar in both cases, it is shorter for the simple linked list since the only thing we have to do is to connect the new node and upgrade the head of the whole data structure.

```
    void insert (Node toAdd) {

        toAdd.next = first; //linking the new node
        first = toAdd; //updating the head of the list
    }
```

On the other hand, as far as the double linked list is concerned, more operations have to be executed since there are more fields that have to be linked, as presented below:

```
void insert (Node toAdd) {
    toAdd.next = first; //connecting the node to the list
    toAdd.prev = null; //setting the prev field
    first.prev = toAdd;//updating the prev field of the initial list head
    first = toAdd; //updating the head of the list
}
```

Note: When writing these operations it is highly important to keep track of the order they are executed in, as they are not symmetrical. And what do I mean by that is for instance, the head of the list should not be updated before the linking with the new node is done. If done this way, the reference of the whole list is lost, which is something to be avoided! Therefore, the order matters!

As far as the unlink method is concerned, there are several differences between the two lists, however, the main one is that regardless of having the reference of the new node, in the simple linked list we still have to traverse all the way through in order to properly identify and remove it. Conversely, the node of the doubly linked list gives us all necessary pieces of information, the only thing being left is to make the correct linking. That is precisely why this operation's time complexity is *O(1)*, while the simple linked list's is *O(n)*.

In order to better understand the differences between the two, the following table has been created:

Type	unlink	insert
LinkedDouble	O(1)	O(1)
LinkedSimple	O(n)	O(1)

Table 1: Time complexity analysis of the *unlink* and *insert* operations.

Unlinking and inserting nodes: explaining the benchmark results

After having set up the methods, the next step was identifying the difference in the execution time when unlinking and inserting a node in both lists. From the perspective of relevance, this comparison has been done by taking into consideration *the ratio* between the simple and double linked list execution time as well.

Moreover, although the nodes which were unlinked and then inserted were generated randomly, in order for the results to be once again *relevant* (especially because we have to go through the whole simple linked list for the unlink operation), the two operations were executed using the same node in both data structures.

In order to better understand the analysis of the benchmark, I thought it would be a great idea to first start looking at the results generated by inserting random nodes only as presented in *Table 2* below.

size	doublyinsert[ns]	simplyinsert[ns]
25	459	493
50	298	367
100	527	615
200	686	226
400	1098	479
800	881	869
1600	2154	2025
3200	3306	3284
6400	4967	4944
12800	8756	8761
25600	16786	16777
52000	33750	33671

Table 2: The evolution between the simple linked list and double linked list taking into consideration the **insert operation only**.

By looking at the numbers, although there are certain exceptions, most of the time, the execution times are extremely close, especially after having reached the size of 1600 elements, therefore for growing arrays. In other words, these results back up our conclusion of the *insert* operations being *O(1)*, which will be of use in the next part of the analysis.

By now looking at *Table 3*, which shows the execution time of both unlinking and inserting a node in both lists, we come to the conclusion that the numbers are no longer close to one another at all and by far, the doubly linked list outperforms the simply one. If we look back: by only by inserting nodes we got pretty close results, now after having added a new operation to our analysis, namely the *unlink*, the situation changes. It, therefore means that this huge difference is made by this operation only. And it does make sense: as far as the simply linked list is concerned, the entire list has to be traversed in order for the element to be found and then, unlinked, being an *O(n)* operation. Unlike the doubly linked list, where this takes place instantaneously, where if we look for large arrays (e.g size = 12800) the unlink and insert execution time is almost double the insert time only.

Note: The *ratio* in the table below is calculated as the ratio between the simple linked list's execution time over the double linked list's.

size	doublytime[ns]	simplytime[ns]	ratio
25	355	434	1.22
50	180	386	2.14
100	258	550	2.13
200	326	660	2.03
400	544	1154	2.12
800	875	2099	2.40
1600	2131	4903	2.30
3200	4636	10785	2.33
6400	9011	20958	2.33
12800	18053	43285	2.40
25600	29397	76488	2.60
52000	54267	146680	2.70

Table 3: The ratio evolution between the simple linked list and double linked list.

The graph below demonstrates a better visualisation of the ratio's included in the table above.

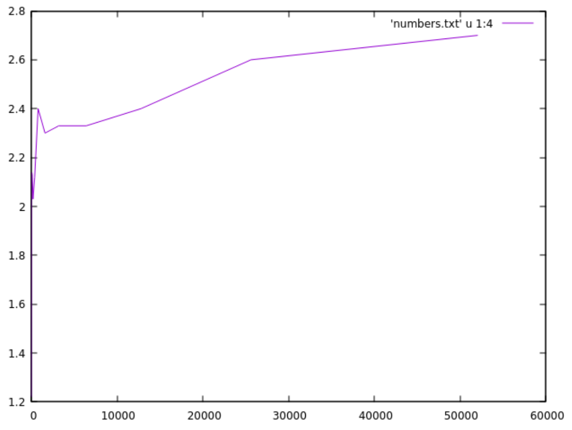

Figure 1: The graph describing the evolution of the ratio between the doubly and simply linked lists.

Difficulties

The main difficulty I encountered in this assignment was to always keep track of all the connections that I had to set up when unlinking or inserting a node, especially when it comes to the double linked list. For instance, at some point I kept getting an error and that was because I had forgotten to upgrade the new head of the list.

Conclusion

All in all, setting up a double linked list requires an organised approach and a lot of attention mostly when creating the node connections. However, once everything runs smoothly, it does pay off! And that is especially when removing a node from the whole data structure!