A Doubly Linked List

Mahdi Nazari

September 2022

Introduction

In this assignment the doubly linked list will be presented. It will be discussed about doubly linked list attributes and the differences between a single linked list and a doubly linked list. A benchmark will be done to see the execution time to delete a node from a single linked list versus doubly linked list.

Double Linked List

A doubly linked list has the all attributes that a single linked list has, such as nodes with a value and a reference to the next element. But beyond those it also has a reference to the previous element. This feature will be useful to perform some operations but it will also cost more execution time since a node in a doubly linked list has two references which should be updated every time we traverse in the list. In this assignment we will focus on the removing an element and execution time for that in a linked list and in a double linked list. In order to get a reliable result we will try to minimize all other factors as long as possible.

Benchmark

The benchmark was set as follows: we copied the same list of elements into the linked list and double linked list. For each of these lists an array of type Node was constructed to store the references to each element in the list. A common array for these lists was constructed and this array will contain the indexes to the reference-array. The elements of the index-array will be chosen randomly. The procedure in the benchmark is as the first element in the index-array gives an index to a reference in the reference-array which will gives us a reference to an element in the linked list and double linked list which should be removed and added.

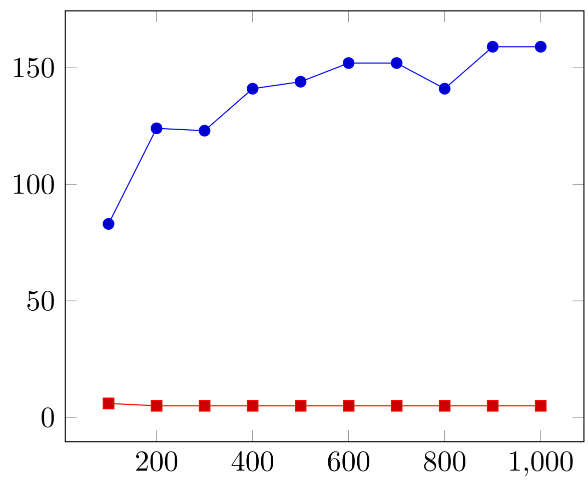
In the benchmark we will calculate the execution time for removing the same element from respective lists and add it back to the list. The calculated time will be the time to remove an element and add it back to the list. But since we add the removed element to the beginning of the list in both single- and double linked list, the execution time for add-operation will be the same for both data structures and will not affect our results. The removing function for a double linked list looks like as below:

```
public void removeNode(doublyLinkedNode node){
            if(node.previousNode != null){
                node.previousNode.nextNode = node.nextNode;
            }
            if(node.nextNode != null){
                if(node.previousNode == null){
                    head = node.nextNode;
                }
                node.nextNode.previousNode = node.previousNode;
                node.nextNode = null;
            }
        }
```

The benchmark was performed and the result is as in the Graph 1 and Table 1. To calculate the runtime a loop of 10000 times was set and in every iteration the calculated time was added up in a local variable (t-total) and to determine the average time for a single remove-add operation t-total is divided by loop(10000), times number of remove-add operations which was 100 .

Number of Nodes	**Linked List**	**Double Linked List**
100	83	6
200	124	5
300	123	5
400	141	5
500	144	5
600	152	5
700	152	5
800	141	5
900	159	5
1000	159	5

Table 1: The average time for removing and adding an element to the single-double linked list. Time in ns



Graph 1: The result from Table 1 plotted in a graph. Linked list(blue graph) and double linked list(red graph) . x-axis, size of arrays, y-axis execution time in ns.

Conclusion

The obvious result is that the execution time of remove-add operation in a double linked list(red graph) is constant thus time complexity is an O(1) type. But why?!
To remove an element in a double linked list the program just need to update two references and this process will take the same time regardless to the list size. To add an element to the start of the list as we discussed earlier will have a constant execution time too. Thus remove-add operation has a constant runtime in a double linked list.

In a single linked list to remove an element, the program has to to traverse in the list until it reaches the node. Traversing in a list to find an element in the position (n) has a time complexity of O(n) type. But it could have an O(1) type for removing an element if we could keep track of the previous element and the element that should be removed at the same time then we could construct a new remove function for linked list with a constant time complexity. But in this case we have just reference to the element that should be removed. The add operation has a constant runtime as earlier. This means that time complexity of remove-add operation in a linked list should be an O(n) type in total. But is the graph for the linked list(blue graph) a linear type?!1
The graph is not maybe a perfect linear graph but it still shows a slightly increasing of execution time when the size of the list increase. Another possibility can be that the nodes which were removed from the list and were added back to the list were chosen by a random generator and this random

generator could had chose the nodes in the early part of the list which could have affect the result since traversing in the early part of the list does not take as long time as traversing until the end of the list. The other reason that the result is not as perfect as it was expected can be that there are some other factors in the program which they are affecting the result despite the effort to identify and prevent them.