Doubly Linked List

Ayda Ghalkhanbaz

Fall 2023

Introduction

The main purpose of this assignment is to analyze a doubly linked list. We will look into the implementation details of a doubly linked list and benchmark the list as well. In order to gain a clearer comprehensive understanding of a doubly linked list, we will also compare the benchmarking results of doubly linked lists with equivalent operations on singly linked lists. This comparative analysis will be discussed in the later section.

Doubly Linked List

Doubly linked list is an extended version of the singly linked list data structure. In addition to tracking the next node, each node in a doubly linked list has a pointer to the previous node as well. This attribute of doubly linked lists is very useful and efficient for traversing in both forward and backward directions. Figure 1 illustrates a basic structure of a doubly linked list.

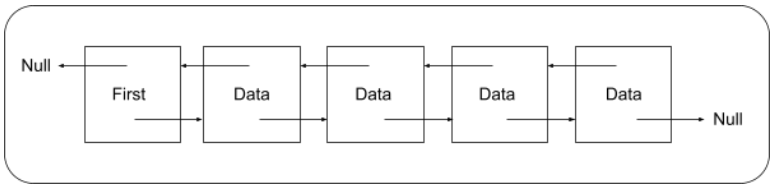


Figure 1: A basic illustration of a doubly linked list structure

Method

The implementation of a doubly linked list is quite similar to that of a singly linked list, the only difference is a pointer to the previous node. The code snippet below shows the creation of a single node with pointers to both previous and next.

```
    public class DoublyNode{
    int data;
    DoublyNode next;
    DoublyNode previous;

    public DoublyNode(int data){
        this.data = data;
        this.next = null;
        this.previous = null;
    }
}
```

Similar to the singly linked list, the doubly linked list contains methods such as add, remove, length and find. The 'add' method takes in an integer as a parameter and creates a node with it, then simply adds it to the beginning of the list. The 'find' and 'length' methods have exactly the same implementation as in the singly linked list. The implementation of the 'remove' method was initially a bit tricky but it didn't pose major problems. The 'remove' method searches for the target item in the list and then links its previous and the next node together. The difference between 'remove' method in doubly and singly linked lists is that we do not need to always keep a track of the previous node, since every node in the doubly linked list already has a pointer to its previous node.

In addition to the methods mentioned above, two additional methods were implemented for both doubly and singly linked list classes; 'unlink' and 'insert'. These methods function similarly to the 'remove' and 'add' methods, respectively. Below, you'll find a code snippet demonstrating the implementation of the 'unlink' method in the doubly linked list.

```
    public void unlink(DoublyNode node){
        // if the list is empty
        if(this.head == null){
            return;
        }
        // if the node is the head
        if(node == this.head){
            this.head = node.next;
            node.next = null;
            node.previous = null;
        }
```

```
        // if the node is not the head
        if(node.previous != null){
            node.previous.next = node.next;
            node.previous = null;
        }
        // if the node is not the last node
        if(node.next != null){
            node.next.previous = node.previous;
            node.next = null;
        }
        return;
    }
```

In the 'unlink' method several conditions are checked. Firstly, it returns immediately if we try to unlink a node from an empty list. The second if-statement handles the scenario where we want to unlink a node that is the head, i.e. the first node in the list. In this case we set the next node as the new head of the list and nullify the target node's previous and next pointers. The third and the fourth if-statements are for the nodes that are not the head nor the last node. In such a case, we nullify the target node's pointers and make its previous and next nodes point to each other.

The 'insert' method is quite similar to the 'add' method with the distinction that the 'insert' method accepts a node instead of an integer as a parameter and then inserts the node at the beginning of the list. The following code snippet illustrates the implementation of the 'insert' method.

```
    public void insert(DoublyNode node){
        // if list is empty
        if(this.head == null){
            this.head = node;
        }
        else{
            node.next = this.head;
            this.head.previous = node;
            this.head = node;
        }
        return;
    }
```

Singly Linked List

The singly linked list structure is discussed completely in the previous assignment, but in summary we can say that the singly linked lists consist of nodes, each containing a value and a pointer to the next node.

Method

In this assignment we have added the 'unlink' and the 'insert' methods to the implementation of the singly linked list. The implementation of the 'unlink' method closely resembles that of the doubly linked list. The only difference is that in the singly linked list we must keep track of the previous node while we didn't need to do this in the doubly linked list. The code snippet below demonstrates the implementation of the 'unlink' method in the singly linked list.

```
public void unlink(Node node){
    Node curr = this.first;
    Node prev = null;

    while(curr.next != null ){
        if(curr == node){
            // if the node is not the head
            if(prev != null){
                prev.next = curr.next;
            }
            else{
                this.first = curr.next;
            }
            node.next = null;
            break;
        }
        prev = curr;
        curr = curr.next;
    }
}
```

The 'insert' method in the singly linked list has the same implementation as in the doubly linked list, with the distinction that we don't have any previous pointer to update, so we only update the next pointer.

Benchmark

In order to measure the execution time of unlinking and inserting nodes in doubly and singly linked lists, two lists of each type were created. These lists were filled with nodes of random values. Additionally, an array with random keys was generated; this array then was used to choose a random node from the lists to be unlinked/inserted. With varying the size of lists and iterating thousands of times over the unlink-insert operations, the minimum execution time was picked for displaying the performance results of doubly vs singly linked lists.

Result

The results from the benchmark are presented in table 1. For a clearer visualisation, the measured execution time for only unlinking nodes from doubly and singly linked lists are plotted in figures 2 and 3. Additionally, figure 4 displays a comparative analysis of only inserting nodes into doubly vs singly linked lists.

Array size	Doubly LL	Singly LL
100	5.4	140
200	5.5	270
400	4.5	530
800	4.6	1000
1600	4.8	1500
3200	5.4	2000
6400	6.1	2400
12800	6.8	2600
25600	7.7	2700
51200	8.0	2800

Table 1: Benchmark results of unlink and insert nodes in doubly vs singly linked lists in μs

Figure 2. Unlink nodes from singly LL

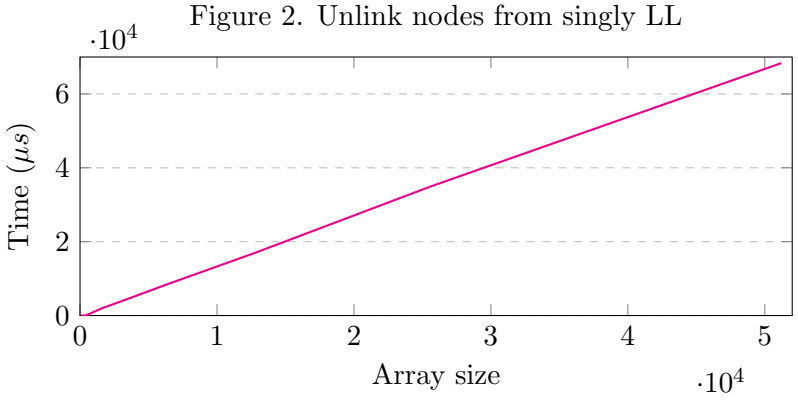


Figure 3. Unlink nodes from doubly LL

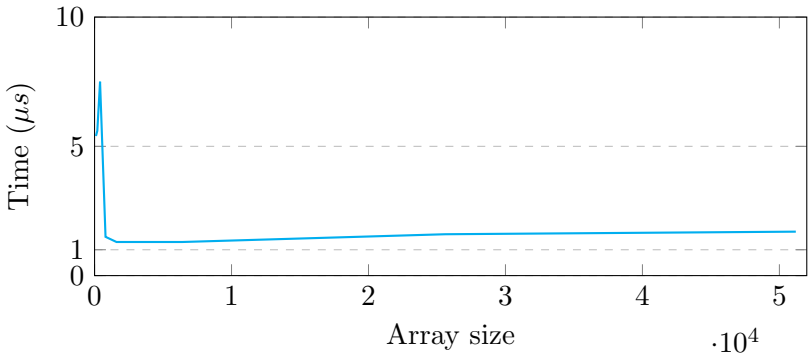
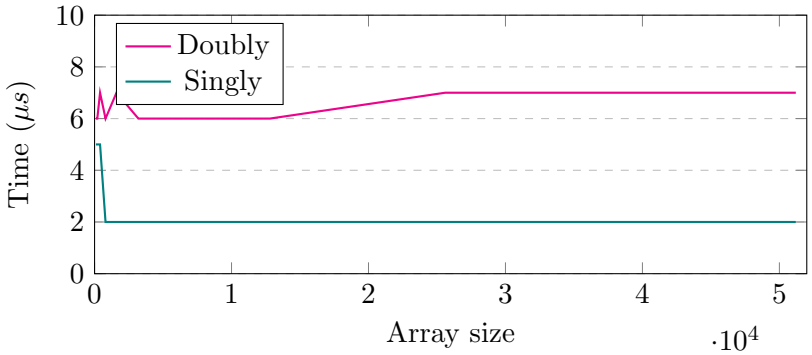


Figure 4. Insert nodes to doubly vs singly LL



Discussion

According to the outcomes from the benchmark shown in table 1, it is evident that in general, a doubly linked list outperforms a singly linked list in handling the unlink-insert operations. Upon closer examination of figures 2 and 3, we can see that unlinking nodes from a singly linked list has a linear behaviour whereas the same operation takes a constant time for the doubly linked list. While the doubly linked list takes the advantages of having two

pointers pointing to the next and previous nodes, the singly linked list needs to constantly keep track of the previous node and in its worst case, travers the entire list for finding the target node. This leads to a time complexity of $\mathcal{O}(n)$ and $\mathcal{O}(1)$ for unlinking nodes from singly linked list and doubly linked list, respectively.

Figure 4 confirms that inserting a node into both doubly and singly linked lists takes a constant time as both structures have the same implementation of the insert method, adding a node to the beginning of the list. Thus, the time complexity of adding a node into the beginning of a linked list maintains $\mathcal{O}(1)$ for both doubly and singly linked lists.

It is also noteworthy that despite having the same time complexity for inserting a node, we can see that the singly linked list performs the operation faster.The reason lies in the structural differences between them; in addition to the 'next' pointer, the doubly linked list has to update the 'previous' pointer as well, which can cost a few microseconds for the doubly linked list.