Doubly Linked Lists

Aditya Gupta

September 2022

Introduction

In this assignment, we implement the doubly linked lists and compare them against their rudimentary counterparts: the singly-linked lists. The only difference between the regular linked list and this one is that this linked list stores two references that is the one to the next element and the one to the previous element. While in the case of a singly linked list we only store references to the next element. This means that in a doubly linked list we can remove an element without having to traverse the entire list each time. We gain the ability to traverse the list in both the forward and backward direction because of the previous pointer in the doubly linked list which isn't possible in the singly linked list.

Append Data

Implementation

First, we check if the list is empty or not; if the list is empty we add the element as the new head and set the previous and the next pointer of the head node to null. We set the tail element to the head as it is the first as well as the last element at this point. If the list has more than one element, we check for the nullity of the tail's next node and then set the tail's next node as the node with the data passed by the user as an argument. We set the tail's next node's previous node as the current tail and return. Thus we set the new node that has been added to the list's previous node as the old tail node.

If the tail element is null, we go through the entire list and add the data passed as an argument as a new node at the end of this list. We do this by creating a current node variable that initially stores the head, which is constantly updated to contain the next value. Once we reach the end of the list, we set the previous pointer for the next node to the current node. The overall operation of appending for this kind of list is a bit more than the time taken for the case of a singly linked list as we need to update both the next and the previous pointer.

If we keep track of the tail node in the case of the singly linked list while appending then the time complexity for both the single and the doubly linked list should be $O(1)$. This is because we just update the value of the tail node to be the new node to be added to the list rather than having to iterate through the entire list. In theory, the singly linked list might even be faster than the doubly linked list as it needs to update fewer nodes for each append operation compared to the doubly linked lists.

The assignment asked us to add items to the beginning for both of the lists. In my implementation, however, I always add to the end but as I keep track of the tail node in both the lists when adding elements, we still operate with time complexity of $O(1)$. Thus with my implementation, it doesn't matter if I add to the beginning or to the end as the execution time for both have constant time complexity.

```
public void appendData(int data) {
        if(head == null){
            head = new Node(data);
            head.prev = null;
            head.next = null;
            tail = head;
        }
        else{
            if(tail != null){
                tail.next = node;
                tail.next.prev = tail;
                return;
            }
            Node current = head;
            while(current.next != null){
                current = current.next;
            }
            current.next = new Node(data);
            current.next.prev = current;
        }
    }
```

Results

The data associated with the append operation in the 2 different kinds of linked lists(singly linked list - SL, doubly linked list - DL) have been tabulated below:

Case	n values	Time for DL lists(in nanoseconds)	Time for SL lists(in nanoseconds)
1	250	120	70
2	500	140	80
3	1000	100	80
4	2000	80	90
5	4000	80	120
6	8000	100	150
7	16000	100	170
8	32000	180	250
9	64000	300	340

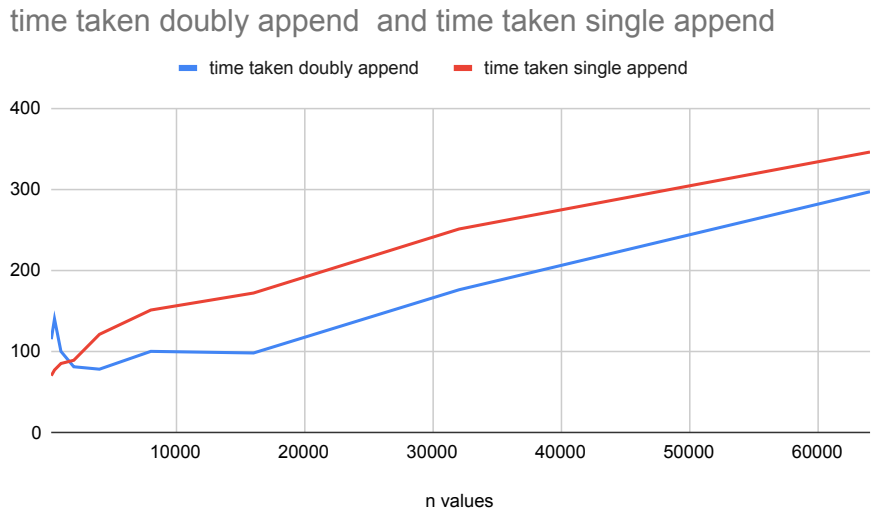

Figure 1: Append Data

This data was obtained upon doing 1000 iterations of the append operation for both of the lists. We see that the values for the append operation for the linked lists are not exactly constant and are even increasing with higher values of n in a somewhat linear fashion. This may be attributed as a side effect of caching. Till a certain value of n, the data is retrieved from the cache. But once it exceeds this threshold then we start seeing an increase in the execution time as data now has to be retrieved from a deeper part of the memory thus leading to a non-constant time trend for the function. But this is just a hypothesis and we can't say for certain if this is because of caching or some other factor.

Remove Data

Implementation

```
private void deleteNode(Node node) {
        if (node != null) {
        // Here we can rely on 'node' actually being in our list
        if (node.prev != null)
            node.prev.next = node.next;
        else
            head = node.next;
        if (node.next != null)
            node.next.prev = node.prev;
        else
            tail = node.prev;
    }
}
```

In the case of the doubly linked list if the node passed to this method is null this indicates that the data we wanted to remove did not exist in the list we just do nothing in that case. If the argument node is not null then we inspect the neighbouring nodes of the argument to remove it from the list. While the node's previous element is not null then we set the previous element's next pointer to the current node's next node otherwise we set the head to the node's next node value as the only node whose previous pointer should point to null is the head node of the linked list. Now we check for the nullity of the next pointer of our current node and we similarly set its next node's previous node as the current node's previous node to remove the current node. If the current node's next node is null then we set the tail node as the current node's previous node, we do this as the only node which is allowed to have its next node as null is the tail node.

Here is where we start to see the true benefits of using a doubly linked list, as in the singly linked list we always need to traverse the entire list to find the element to be deleted as we first need to locate the node with the data to be deleted. We check if the head node's data is the same as the data to be deleted, and if it is then we replace the head with its next node. Otherwise, we iterate through the list using a node variable which is initialized with the head. After which we update the current variable to the next node so as to Thus the time complexity of removing an element in a singly linked list is $O(n)$ while in the case of doubly linked lists only need to update a few nodes which means that the time complexity of removing elements from a doubly linked list is $O(1)$.

Results

The data associated with the remove operation in the 2 different kinds of linked lists(singly linked list - SL, doubly linked list - DL) have been tabulated below:

Case	n values	Time for DL lists(in nanoseconds)	Time for SL lists(in nanoseconds)
1	250	40	60
2	500	40	110
3	1000	60	200
4	2000	70	320
5	4000	60	710
6	8000	60	2000
7	16000	70	4000

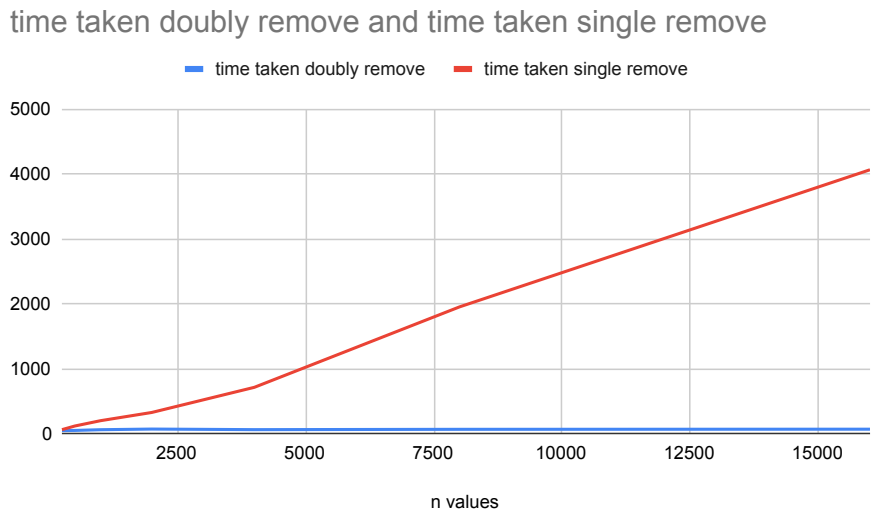

Figure 2: Remove Data

This data was obtained upon doing 1000 iterations of the remove operation for both of the lists. We observe that the remove operation for the doubly linked list behaves in a similar way to the append operations that we analysed before. The only way this behaviour can be explained is based on the caching hypothesis: lower values of n caching save us some time while with higher n values we see that the time taken to do data retrieval leads to a greater overall execution time. While in the case of the singly linked there is a clear linear trend that we can see in both the tables and the graphs. Even when in the tables we can see that there is an increase in the execution time values of the doubly linked list remove operation, it looks constant when compared with the time taken by the same operation of the singly linked list.

Conclusion

The data associated with the remove operation in the 2 different kinds of linked lists(Time for singly linked list append - SLA, Time for doubly linked list append - DLA, Time for singly linked list remove - SLR, Time for doubly linked list remove - DLR) (all the time values given here are in ns) have been tabulated below:

Case	n values	DLA	SLA	DLR	SLR
1	250	120	70	40	60
2	500	140	80	40	110
3	1000	100	80	60	200
4	2000	80	90	70	320
5	4000	80	120	60	710
6	8000	100	150	60	2000
7	16000	100	170	70	4100

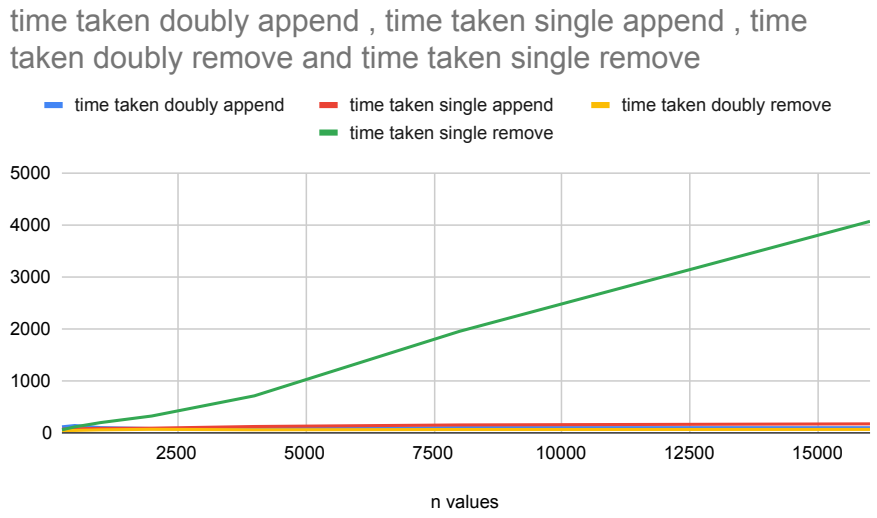

Figure 3: Conclusion Data

Thus we can conclude that the execution time for the singly linked list append, doubly linked list append and doubly linked list remove is constant or has a time complexity of $O(1)$. While the execution time for the singly linked list removal is given by a time complexity of $O(n)$ as we need to search for the element in order to remove it from our list. The behaviour where the time values should appear constant but look linear is attributed to caching, and an increase in time for data retrieval with an increase in the number of elements stored in the lists.