Assignment D: Doubly linked list

Mahad Ahmed

Spring Fall 2022

Introduction

In this assignment we wanted to move on from singly linked list and explore a doubly linked list. The concept is pretty much the same except for the key difference that a doubly linked list has not only a forward pointer but also a previous pointer. This can help us when we want to remove something without traversing the whole list from the beginning. In this assignment I have implemented a doubly linked list and some methods to help us test how it stacks up against a singly linked list.

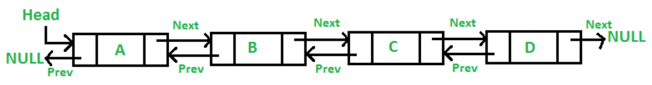

Figure 1: An illustration of how doubly linked list works

Doubly linked list

We implemented a doubly linked list and ran some benchmarks.I have two classes, doublyNode and doublyLinkedList.

```
public class doublyNode{
    int data;
    doublyNode next;
    doublyNode prev;


    public doublyNode(int data){
        this.data = data;
        this.prev = null;
        this.next = null;
    }
}
```

In this code section above you can see my doublyNode class. Here is where I defined what a node in a doubly linked list is. So a node in a doubly linked list can hold three things, an integer, the previous node *prev* and the next node *next*.'

How my head is defined

```
public class doublyLinkedList{

doublyNode head;


doublyLinkedList(doublyNode head){

    this.head = head;
}
.
.
.
.
}
```

Here is how my head is defined. Our first node is the head.

Remove method

This is how I implemented my remove/delete method.

```
public void remove(doublyNode nodeToBeRemoved){

    // If head is null then there is no list
    if(head == null){
        System.out.println("Nothing here to remove");
    }
    //if nodeToBeRemoved is null then
    //is nothing to remove
    if(nodeToBeRemoved == null){
        System.out.println("No node detected");
    }
    //Case if we want to remove the head
    if(head == nodeToBeRemoved){
       head = head.next;
    }
    //Case if we want to remove a node that is not the last node
    if(nodeToBeRemoved.next != null){
        nodeToBeRemoved.next.prev = nodeToBeRemoved.prev;
    }
    //Case if we want to remove a node that is not the head
    if(nodeToBeRemoved.prev != null){
        nodeToBeRemoved.prev.next = nodeToBeRemoved.next;
    }
    return;
}
```

To get this took a bit of brain power and finesse. I arrived at five different cases I could end up with. The first two are just base cases. The other three is if the node we want to remove is the head node. Or if the node we want to remove is not last. Lastly one for when the node to be removed is not the head node.

Add method

The add method was fairly simple to implement because we were just going to add in the front of the list. The only thing I did was really just update the pointers to the node after when doing the add operation.

```
void addAtTheStart(int data){
    doublyNode node = new doublyNode(data);

    //base case to see if there are any nodes
    if(head == null){
        head = node;
    }
    //making the node we are adding the head and updating pointers
    else{
        node.next = head;
        head.prev = node;
        head = node;
    }

}
```

Benchmark, singly vs doubly linked list

Elements n	**Run time(ns)**
100	450
200	830
400	1700
800	3310
1600	6307
3200	13021
6400	29213

Table 1: Benchmark for remove method singly

Elements n	**Run time(ns)**
100	53
200	54
400	53
800	50
1600	54
3200	53
6400	51

Table 2: Benchmark, remove method doubly

Elements n	**Run time(ns)**
100	87
200	87
400	89
800	87
1600	87
3200	86
6400	88

Table 3: Benchmark, add method singly

Elements n	**Run time(ns)**
100	89
200	87
400	89
800	89
1600	89
3200	88
6400	88

Table 4: Benchmark for add method doubly

Discussion

Remove method: Singly vs doubly

In the benchmarks we see how different the results are. If we look at table 1 and 2, benchmarks for the removal methods. We see that the singly is linear and the doubly is constant. Why is that? Because for the doubly linked list we already have a reference to the node we want to remove. But for singly you don't have the reference to a previously pointer which means that we will have to traverse through the list until we find the element to remove.

Singly remove: *O(n)*
Doubly remove: *O(1)*

Add method: Singly vs doubly

The benchmark for the add methods of both singly linked list and doubly are both in fact constant. Because we are just adding a node in the beginning of the list. This is the best case scenario. But in the benchmark we can see that the add a node in the singly linked list is faster. It is because our doubly linked list has more pointers to update which makes it less efficient in this case. We need to update the prev of the current first node while our singly doesn't need to do that. There is a slight cost which makes it a little more expensive than our singly friend.

Singly add:*O(1)*
Doubly add:*O(1)*