# A doubly linked list

Isak Wilkens TIDAB

Fall 2022

## Introduction

This is a report on the first higher grade assignment "A doubly linked list" on the course "ID2021" TIDAB, KTH. The assignment is a continuation on the topic of linked lists where the concept is expanded on through having a pointer to the next and the previous element in the list, doubly linked. In the previous assignment we only had a pointer to the next element in the list. This new feature makes the linked list more efficient in some regards. To test this we will benchmark the operation of removing a random element from both a single linked list and a doubly linked list and then adding it to the beginning again repeating both $n$ times.

## A doubly linked list

Firstly to be able to implement a doubly linked list the feature of a backwards pointer needed to be added. For that implementation the backwards pointer was named "head" and the value that the node keeps is called "item". Then the feature to add or remove a node element had to be implemented. Since the list had pointers to the elements on both its sides we do not need to iterate through the list as we had to do in the case of the single linked list. The remove function needs three conditional statements depending on the input. The normal case is that the input node that is to be removed is in the middle of the list where we move the pointers of the previous and following element to each other thus removing the element. The two other special cases we need to account for is the first and the last element where we instead need to change the one pointer that is pointing to it to null. For the add operation we simply link its tail to the previously first element and then return the pointer to the new element. In these operations we need return statements to be able to send back the correct first element of the linked list. The code for these operations is given below.

Remove operation:

```
public LinkedList remove(LinkedList linklist){
    LinkedList first = this;
```

```
        LinkedList nxt = linklist;
        if(nxt == first){
            nxt.tail.head = null;
            return nxt.tail;
        }
        else if (nxt.tail != null){
            nxt.head.tail = nxt.tail;
            nxt.tail.head = nxt.head;
            return first;
        }
        else{
            nxt.head.tail = null;
            return first;
        }
    }
```

Add operation:

```
    public LinkedList add(LinkedList linklist){
        LinkedList first = this;
        first.head = linklist;
        linklist.tail = first;
        return first.head;
    }
```

## Bench-marking

To be able to bench-mark comparably the creation of both the single and doubly linked lists had to be changed. First an array of random values was created with the intended size for the linked lists:

```
    public static int[] randArray(int nElements) {
        Random rnd = new Random();
        int[]temp = new int[nElements];
        for(int i = 0; i < nElements; i++){
            temp[i] = rnd.nextInt(300);
        }
        return temp;
    }
```

A linked list is then created with the values from this array:

```
    public static LinkedList createList(int[] randArray) {
        int nElements = randArray.length;
        LinkedList temp = new LinkedList(randArray[0], null, null);
```

```
        for(int i = 1; i < nElements; i++){
            temp.append(new LinkedList( randArray[i], null, null));
        }
        return temp;
    }
```

An array of the nodes from this linked list is then created to be able to keep track of the nodes and randomize them:

```
    public static LinkedList[] indexArray(int length, LinkedList linkList) {
        LinkedList[] temp = new LinkedList[length];
        LinkedList nxt = linkList;
        int i = 0;
        while (nxt.tail != null){
            temp[i] = nxt;
            nxt = nxt.tail;
            i++;
        }
        return temp;
    }
```

Table 1 shows the results from bench-marking a doubly linked list of size $k$ $n$-times. We can see from the results that this gives a time complexity of O(1) for the run-time as a function of input size. The number of remove and addition operations ,$n$,was kept constant for varied sizes of $k$. This constant result can be explained by the fact that in the doubly linked list we can go to any given element and remove it without iterating through the list. This means that it will take the same time no matter the size of the list.

| **Varied array size(k)** | **2** | **8** | **32** | **128** | **512** | **2048** | **8192** | **16384** |
|---|---|---|---|---|---|---|---|---|
| run-time(ns) | 102 | 110 | 100 | 90 | 100 | 100 | 185 | 188 |

Table 1: Mesurements of the run-time as a function of linked list size. Run-time in nano seconds.

The results from bench-marking the same operation on the single linked list are shown in table 2. The number of remove and add operations was here also kept constant. The results show a time complexity of O(n) increasing linearly with the size of the linked list. This can be explained by the fact that we have to iterate through the linked list each time to remove a random element.

| **Varied array size(k)** | **2** | **4** | **8** | **16** | **32** | |
|---|---|---|---|---|---|---|
| run-time(ns) | 600 | 860 | 1600 | 400 | 4000 | 7800 |

Table 2: Mesurements of the run-time as a function of linked list size. Run-time in nano seconds.

## Conclusion

Finding the special cases for the first and the last element took some testing to find out but was managed well using return statements. In conclusion a doubly linked list has advantages but requires extra memory for the additional pointer and extra coding. In this case the extra code was well worth the value since we got a much better time complexity for the run-time of removing a random elementand adding it again.