Doubly Linked Lists

William Frid

Fall 2023

Introduction

Linked lists come in different formats. Two of which are singly and doubly linked. "Singly" means each cell in the linked in has a reference to the next cell, and double means it has both a reference to the previous and next cell. This comes with a bit more logic and memory required for each cell, but also advantages when it comes to writing algorithms as the reference to the previous cell can add to simplify code and foremost, better performance.

This report will focus on how the performance increases with the doubly linked list by focusing on the delete algorithm as well as implementing the two methods `insert()` and `unlink()` to the cell object.

Implementation

Class Structure

Implementing a doubly linked list is similar to the implementation of the singly linked list. With the difference being support for the additional, reference for the previous cell, and major changes to the method can can take advantage of this reference, that being `remove()`, and the new methods for the objects of type `Cell`.

Note also that some methods have some logic only to handle this additional variable (`add()` *and* `append()`*).*

```
class LinkedListDoubly {

    Cell first;

    public class Cell {
        int val;
        Cell nxt;
        Cell prv;

        Cell(int target_val, LinkedListDoubly list) { ... }
        public void unlink() { ... }
        public void insert(LinkedListDoubly list) { ... }
    }

    public void add(int val) { ... }
    public int length() { ... }
    public boolean find(int key) { ... }
    public void remove(int key) { ... }
    public void append(LinkedListDoubly second_list) { ... }

}
```

remove()

Focusing on the remove method, the difference now is that there is no need to keep track of multiple cells at a time. Let's say you wish to remove the cell with value
t key, which happens to be the second element in the list. Then the method `delete()` will iterate until it finds this element, then set the next cell reference of the previous element equal to its own next cell reference, and the other way around for the previous cell reference. This allows for a cleaner algorithm with fewer variables. See the snippet below.

```
public void remove(int key) {
    Cell target = this.first;

    if (target.val == key) {
        this.first = target.nxt;
        this.first.prv = null;
        return;
    }

    while (target != null) {
        if (target.val == key) {
            if (target.nxt == null) {
                target.prv.nxt = null;
            } else {
                target.prv.nxt = target.nxt;
                target.nxt.prv = target.prv;
            }
            return;
        }
        target = target.nxt;
    }
}
```

insert() & unlink()

Looking at the two new methods inside the class `Cell`, we have `insert() and unlink()`. At a quick overview, one might think they are similar to the methods `add()` and `remove()`, but they do differ. Starting `insert()`, instead of inserting a new cell into a linked list, we take a specific cell, and insert that into the beginning of a target linked list. This method can for instance be used when one wishes to move a cell from one list to another.

```
public void insert(LinkedListDoubly list) {
    if (list.first != null) {
        list.first.prv = this;
        this.nxt = list.first;
    }
    list.first = this;
}
```

This method is very similar when working with a singly linked list. The difference is that the singly linked list just doesn't require the support for the previous cell reference. Next is
unlink(), which comes with a great advantage over `remove()`. This is because `unlink()` is called from a certain cell, and thus doesn't require any iteration of the linked list. Changing the big-O complexity from $O(n)$ to $O(1)$. This can be accomplished thanks to the previous cell reference.

```
public void unlink() {
    if (this.prv != null) {
        this.prv.nxt = this.nxt;
    } else {
        first = this.nxt;
        first.prv = null;
    }
        if (this.nxt != null) {
        this.nxt.prv = this.prv;
    }
}
```

Comparing this method to how we would write it for a singly linked list cell, we see that the corresponding method would share a lot of similarity to `remove()`. because it lacks the previous cell reference, it still requires a loop, which means that the big-O complexity remains $O(n)$. See the snippet below.

```
public void unlink() {
    Cell prv = first;
    if (this == first) {
        first = nxt;
        return;
    }
    while (prv.nxt!= this) {
        prv = prv.nxt;
    }
    prv.nxt = nxt;
}
```

Benchmark

The two new methods `insert()` and `unlink()` can be very useful. But how well do the different versions of these compare to each other, that is the version for doubly and singly linked lists. This was measured using a benchmark that first called `unlink()` then `insert()` 1000 times targeting random cells, and did so 40000 iterations for different sizes to collect enough data and find the best results with the minimum negative effect from other factors from outside the benchmark.

Analyzing the benchmark and algorithms before looking at the results, we get that for the singly linked list, the big-O complexity for `unlink()` and `insert()`, are $O(n)$ and $O(1)$ respectively. While the complexities for the doubly linked list are $O(1)$ and $O(1)$ respectively. Thus the doubly linked list should perform better and at a constant speed.

Size	**Singly(k)(ns)**	**Doubly(k)(ns)**
2	90	130
4	140	140
8	210	130
16	270	130
32	320	130
64	400	130
128	560	120
256	840	120
512	1200	120
1024	1800	130
2048	4300	150
4096	8600	150
8192	20000	180

Table 1: Benchmark comparing time to perform 1000 unlink and insert operations between singly and doubly linked lists.

The first thing that stands out from the benchmark results in Table 1 is that the time for the doubly linked list is constant, just like predicted since both the big-O complexities for `unlink()` and `insert()` are $O(1)$. For the singly linked list, on the other hand, the time increases linearly. This is due to the fact that its algorithm for the unlink operation has the big-O complexity of $O(n)$ thus giving the linearly increasing time.

Discussion

Because the benchmark uses two different operations, both unlink and insert, the benchmark result becomes harder to understand how it's connected to the two big-O complexities. Additional benchmarks to only compare one method at a time singly versus doubly linked lists would be a great addition so that one would easily understand how each of these operations affects the end result differently (the result of the only benchmark run for this report).

Conclusion

It's clear that a doubly linked list is preferred over a singly linked list if lots of operations such as the unlink operation are going to be performed. The benchmark proved how superior an algorithm with big-O complexity $O(1)$ is over a $O(n)$. However, if the list is going to be used as a stack, then there's no real advantage to use the doubly over the singly linked list.