Doubly Linked lists

Serban Alin-Cristian

Spring Fall 2023

Introduction

In this assignment I have (in c):

- Implemented a doubly linked list
- Implemented an unlink and insert function
- Benchmarked unlinking and inserting n elements, with both a doubly and singly linked list

Implementation

What is a doubly linked list?

A linked list is a data structure that has it's data structured into cells, which contain the data and point to the next cell. This way of organizing data lets you easily add stuff to the end, take out elements midway and manipulate different sections of data without using too much memory.

A doubly linked list however also stores a reference to the **previous** cell inside every cell. This makes it a bit easier to work with when we're given specific cells, somehow.

Difficulties in implementation

Working in C means dealing with nullable warnings. In my specific implementation of linked lists, I have it so that the first element has the prev pointer null, and the last element the next pointer null. That works fine, however if I dare try to refer to element.next.next or element.prev.prev, the compiler cries about possibly dereferencing a null pointer. This made implementing the remove function a nightmare until I figured out a magic trick - not checking if the current element was the one to be removed, but the NEXT element! This let the compiler know that this element, the next one, (and manually checking) the one after that were all not null, and let me perform the neccesary swaps to remove it from the linked list! Huzzah!

New functions

In this assignment we introduce two new functions compared to the singly linked list, **inserting** a referenced cell, which we append to the start of the doubly linked list, and **unlinking** a referenced cell, which effectively removes it from the linked list, but returns the cell we just removed. These work together to check the next benchmark. Also, I implemented versions of these for the singly linked list I made in the previous assignment, although unlinking is much less efficient since being given the cell doesn't give us any advantages, since we can't reach the previous cell to fix the order.

Benchmarking

All these results were obtained as an average of 1000 runs, building a linked list of n elements and a corresponding cell array containing references to all those n elements, then taking n-1 randomly generated indices and doing a unlink-insert for each inside the linked list. (Also, as the Presentation part of the assignment asked, I cleaned up the numbers a bit.)

Table

n	**singly**	**doubly**
10	0.0005ms	0.0030ms
50	0.0010ms	0.0030ms
100	0.0025ms	0.0030ms
200	0.0070ms	0.0030ms
500	0.0100ms	0.0030ms
1000	0.0200ms	0.0030ms
2000	0.0500ms	0.0030ms
5000	0.0800ms	0.0030ms
10000	1.5000ms	0.0030ms

Table 1: Average execution time in milliseconds to unlink and insert n-1 random elements.

As we can see, the doubly linked list unlink-insert has a constant time for each one, which means it has an O(1). This makes sense, because all we're doing is messing with the next and previous pointers of the neighboring cells. The singly linked list however has a linear progression with n's size, because we have to go through the entire array to find the cell we want to remov, and as such, has an O(n).

Conclusions and takeaways

Doubly linked lists are useful in specific cases, because they allow us to access the previous element and as such easily remove a cell, granted we have it's reference. If we DON'T have a cell's reference, a singly linked list is almost always better and more memory efficient. Also easier to implement.