Linked lists

Serban Alin-Cristian

Spring Fall 2023

Introduction

In this assignment I have (in c):

- Implemented a one way linked list
- Improved it and reduced it's appending speed to O(1)
- Compared it to an array appending
- Compared array vs linked list stack implementation

Implementation

What is a 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.

For simple tasks, it is less efficient (appending two 10 element arrays is much faster than appending two 10 element linked lists), but for very large datasets it is much better, as you will see later on.

Improvements to linked lists

The linked list data structure defined above has a cell object which contains a **value** and **pointer to the next cell**, alongside a **first** cell is simple but has problems if you, for example, want to find the length of a very large linked list, or add an element to the end of it quickly.

This is where the **last** cell comes in! Saving the last cell lets us very, very easily append other linked lists or add to the end. This turns our append and add functions from O(n) to O(1), because we have access to the last element at all times instead of having to find it by going through the entire linked list from the start.

Adding a **length** integer to keep track of the amount of elements in the linked list live slows down every add, remove and append operation very, very slightly, but turns our length function from O(n) to O(1)! Worth it.

Also, adding a **previous** cell address would let us travel both forward and backward in the array which would help with some operations, but that is for next assignment, doubly linked lists.

Difficulties in implementation

Working with linked lists (coding, not using them) can be a bit mind boggling. I struggled a bit with implementing the add function because I had two special cases: where the list was 1 element and where the list was with 0 elements because we **removed** the last element. So I ran into a problem, how do I work with an array that is empty?

My remove function, when it removes an element from an array of 1 element just sets the length of the array to 0. Using the length int that I described earlier, I can just check if the length is 0 or 1 and account for these special cases easily.

Speed of different functions

The functions with the name in bold are the versions with the last and length improvements made. To save some time and LaTeXwork, the efficiencies of all functions are:

- Length - O(n)
- **Length** - O(1)
- Add - O(n)
- **Add** - O(1)
- Remove - O(n)
- Find - O(n)
- Append - O(n)
- **Append** - O(1)

The **Length, Add, Append** functions require going through the entire linked list to find the last element. As such, saving the last element (and the length) lets us get these to O(1), in my case about 0.06ms time for all.

The **Remove, Find** functions need to actually go and check if an element exists in the linked list, so we always have to go through the whole linked list until we find it. Because of the nature of a linked list we can't run any fancy searches after we sort the linked list, so O(n) is the best we get.

Comparing array appending and linked list appending

All the following results were obtained as an average of 1000 runs, appending a list of 50 elements at the end of a list of n elements. I could not find a difference in either by changing the order of the appending (50 + n is the same speed as n + 50) for both.

Table

n	**linked**	**array**
10	0.0773ms	0.0157ms
50	0.0009ms	0.0204ms
100	0.0013ms	0.0306ms
200	0.0006ms	0.0534ms
500	0.0012ms	0.1119ms
1000	0.0009ms	0.1455ms
2000	0.001ms	0.2805ms
5000	0.0018ms	0.7212ms
10000	0.0013ms	1.8985ms
100000	0.0076ms	21.1556ms
1000000	0.0083ms	208.9379ms

Table 1: Average execution time in milliseconds to append two lists, repeated 1000 times

Ignoring the first result being titanicly slower than the rest for linked list (this is usually how it goes from experience, C's fault), we can see that linked list has an average execution time of around 0.0050ms, while array's time grows linearly. This is because linked list's append has a O(1), and array's has a O(n+m)!

Stack stuff

As the assignment asked, I've implemented a linked list version of a dynamically allocated stack, and from what I can run it looks to be much faster. Saving the last element lets us very easily add elements to the top of the stack, and remove from the top of the stack as well. This skips the whole rigamarole of making a new array of twice the size and guessing what time we should wait when we shrink the array back down.

Conclusions and takeaways

Linked lists are fantastically useful for data that we need to merge together or remove elements from while keeping the same order, or for implementing stacks. It is MUCH better than arrays in this way. **However**, sorting algorithms are much harder to implement, we cannot access a specific position in the linked list without counting up to it, and such we can't run binary search.

Also they are a bit more difficult to code because of all the pointers involved, but in the end it is very satisfying.