4. Linked List

Kristiāns Vinters

Fall 2023

Introduction

I solved the assignment in Go. I used Go because I want to become more familiar with it. Source code and benchmark data is available on GitHub*.

Implementation

There weren't any particular difficulties implementing linked lists in Go, as Go handles references with C-like pointers. I implemented the structure in a separate `llist` package.

```
type LinkedListItem[T comparable] struct {
    Head T
    next *LinkedListItem[T]
}

type LinkedList[T comparable] struct {
    first *LinkedListItem[T]
}
```

I named the `next` and `first` members all-lowercase, as then they will be private to the package. I then added additional functions for reading just those fields from outside the package. It's the closest to a private setter, public getter you can have in Go. I did it like this to enforce that only internal functions can directly rearrange the list.

```
func (l *LinkedListItem[T]) Next() *LinkedListItem[T] {
    return l.next
}

func (l *LinkedList[T]) First() *LinkedListItem[T] {
    return l.first
}
```

*https://github.com/Phanty133/id1021/tree/master/4-linkedlist

In addition to the required functions, I split off a `Last()` function from the `Append()` function, as that leads to a more readable implementation.

```
func (l *LinkedList[T]) Last() *LinkedListItem[T] {
    if l.first == nil {
        return nil
    }

    item := l.first

    for item.next != nil {
        item = item.next
    }

    return item
}

func (l *LinkedList[T]) Append(value T) *LinkedListItem[T] {
    item := &LinkedListItem[T]{Head: value}

    last := l.Last()
    last.next = item

    return item
}
```

Implementing the linked list-based stack was also straightforward.

```
type LinkedListStack[T comparable] struct {
    list *LinkedList[T]
}

func (s *LinkedListStack[T]) Push(value T) {
    s.list.Add(value)
}

func (s *LinkedListStack[T]) Pop() (T, error) {
    item := s.list.First()

    if item == nil {
        var result T
        return result, errors.New("stack is empty")
    }
```

```
    s.list.first = item.next

    return item.Head, nil
}
```

Both the array and linked list-based stacks have the same time complexities, however, the latter is not required to be contiguous in memory. As a result, there is no need to reallocate the entire stack when it grows.

Benchmarking

I benchmarked the linked list and array by running them 250 times with a fixed size of 500 and changing sizes $\{10, 100, 1000, 5000, 10000, 15000\}$.

Size	t_{LL1}, ms	t_{Arr1}, ms	t_{LL2}, ms	t_{Arr2}, ms
10	0.006	0.001	0.142	0.001
100	0.006	0.001	0.184	0.001
1000	1.00	0.001	0.620	0.001
5000	14.6	0.002	2.57	0.004
10000	53.4	0.003	4.90	0.004
15000	116	0.004	7.27	0.005

Figure 1: Median times for (1) appending n elements to a fixed number of elements, (2) appending a fixed number of elments to n elements.

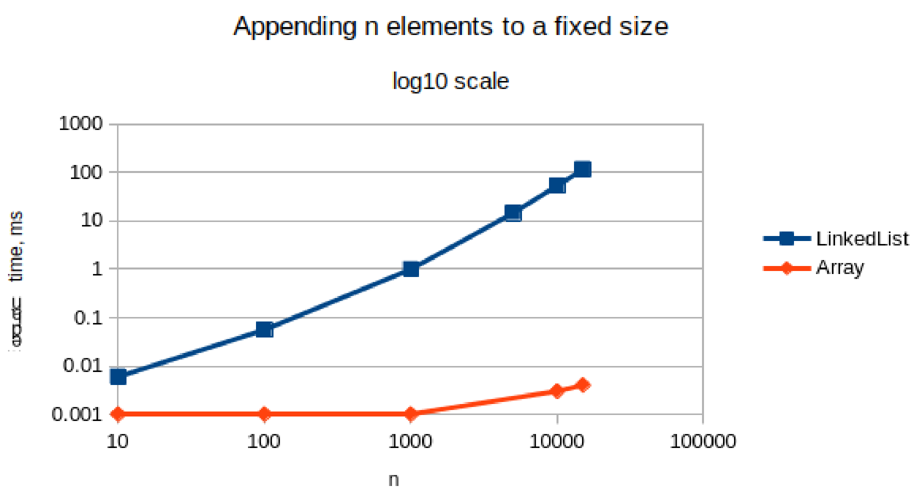

Figure 2: Median time for benchmark 1

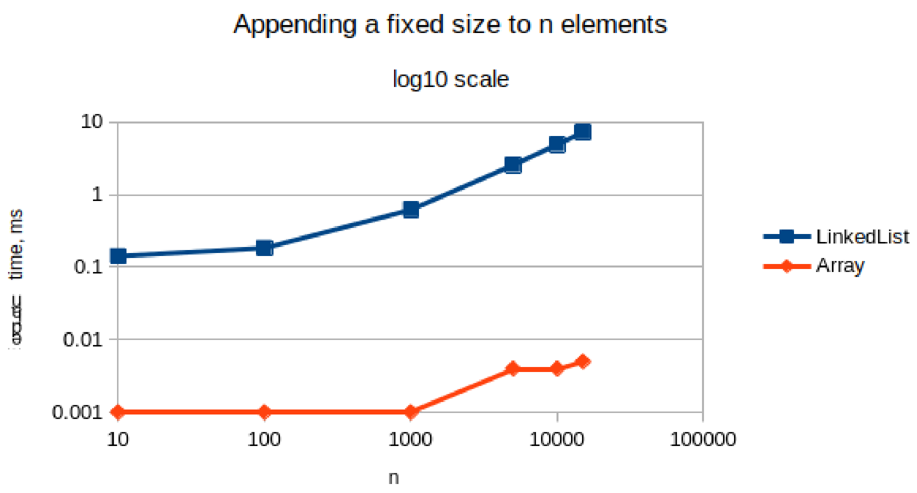

Figure 3: Median time for benchmark 2

From fig. 2 and 3 we can see that the linked list is slower than the array in both cases. This is because appending elements to the linked list is an operation with $O(n)$ time complexity, as the list has to be traversed to find the last element. The array, on the other hand, has $O(1)$ time complexity for appending elements, as it only has to increment the length counter and write the element to the next index.

In fig. 3, a slight hump in the array execution time is visible around $n = 5000$, as the number of appending elements exceeds the initial capacity of the array.

There is also a significant difference in execution time between the first and second benchmark because when appending the fixed size, the entire linked list needs to be traversed less times.