Linked lists in Java

Kinga Anna Koltai

Fall 2024

Introduction

In this assignment we looked at linked data structures, specifically a simple linked list and compared it with previously implemented data structures like arrays or the stack from assignment 2.

Implementation

The idea is that instead of the primitive data types, we can define more complicated structures that can hold several values, but also a pointer to a next element in a list. For the purpose of this assignment, we implemented a linked list that holds integers, so in every node (aka list element) there is an integer stored and a reference to the next node. We looked at the most basic single linked list without a sentinel node, so the head points to its first element which can hold a value and then of course the reference, where the last node has a null reference. It is initialized as an empty list, so first points to null.

add() method

One necessary operation we need to be able to perform is adding elements to the front of the list. This is the easiest way to add elements and has time complexity of O(1) as opposed to adding to the end which would be O(n) or adding to a certain index. Below are my implementation:

```java
public void add(int item){
    Cell new_cell = new Cell(item, this.first);
    this.first = new_cell;
}
```

As you can see, I initialize a new node (called Cell in the code, but I prefer the node terminology, so I will stick to that) with it's tail pointing to the previous first node and have the list itself point to this new node as its first. This is very straight-forward, I need not worry about the list being empty or something like a stack overflow as linked list has dynamic length handled by the operating system behind the scenes.

length() method

Another simple method, we just need to iterate through the entire list from first to null and count how many steps we took. It has O(n) time complexity.

```
public int length(){
    int length = 0;
    Cell current = this.first;
    while(current!=null){
        length++;
        current=current.tail;
    }
    return length;
}
```

find() method

Next we looked at a method that takes an int as its parameter and returns a boolean stating whether that integer is found in the list or not.

```
public boolean find(int item){
    Cell current = this.first;
    while(current!=null){
        if(current.head==item){
            return true;
        }
        current=current.tail;
    }
    return false;
}
```

Also rather simple, if the stored int is equal to what we're looking for we return false, if we iterated out of the list with no success, we return false. Also returns false for empty lists. Its worst case time complexity is also O(n) (best case is O(1), so average case should be somewhere in between, but usually linear).

remove() method

This next method removes a specified element (if it appears in the list). This is a little more interesting as we need to be able to access the tail property of the element in front of the one we are deleting to keep the rest of the

list attached, but since we cannot iterate backwards, we need to compare `current.tail` to the int value while still standing on current. This can raise problems with iterating beyond the bounds of our list if we are not careful. One way to do this is using two pointers setting one one step behind the other one and traversing the list as usual. Another solution is to iterate until `current.tail` is null, so we stop one step before the end.
With both solutions we need to handle separately:

- if the list is empty (meaning `this.first.tail` doesn't exist, this would throw a `nullPointerException`)
- the first element
- the rest of the list (in a loop)

I prefer using a single pointer, so this is the one seen in my code below, but both solutions appear to be O(n) as we need to iterate through the same elements.

```java
public void remove(int item){
    if (this.first==null) return;
    if (this.first.head==item){
        this.first=this.first.tail;
        return;
    }
    Cell current=this.first;
    while(current.tail!=null){
        if (current.tail.head==item){
            current.tail=current.tail.tail;
            return;
        }
        current=current.tail;
    }
}
```

append() method

As the name suggest, this method will be able to append another `LinkedList` at the end of our list.

```java
public void append(LinkedList b) {
    if (this.first==null){
        first=b.first;
        return;
    }
    Cell current = this.first;
    while (current.tail != null) {
        current=current.tail;
    }
    current.tail=b.first;
    b.first=null;
}
```

Once again we could've used two pointers or this way we need to take care of an empty list separately. This also has a time complexity of O(n) where n is the size of the first array, as I explain later. Ideally afterwards we should probably not only set it to null but delete all references to the list that no longer exists, so that the user will not by mistake keep working with the wrong list. This can be done by setting the list to null and calling the garbage collector, but I learnt that even then the JVM may decide not to oblige so I ended up not going through with this idea.
Note, that with doubly linked lists[1] we could reduce this method to O(1), but it would require even more space to store another reference in each node, so depending on the use-case it might not be worth it.

Benchmarks

Next I set up a benchmark to test the time complexity of the `append()` method. I initialized two randomized list and appended the second one to the first, my results are shown in the table below.

list1 size (n)	list2 size (n)	execution time [µs]
100	100	0.2
200	100	0.3
400	100	0.6
800	100	1.1
1600	100	2
3200	100	4
6400	100	7.7
100	100	0.2
100	200	0.3
100	400	0.2
100	800	0.2
100	1600	0.3
100	3200	0.3
100	6400	0.3

[1]Doubly linked lists store a reference to the next as well as the previous node, so that inserting an element to the first and the last position is both O(1), it also makes some of the previous methods somewhat easier

As you can see, increasing the size of the first and the second list had different effects. That is because as I've already mentioned, we only have to iterate through the first list to find the end, so the algorithms time complexity is independent of the second lists length. The numbers in the first half of this table also confirmed that it is O(n).

Comparisons

append for arrays

If we were to implement the append() function on arrays, we would need to create an array that has size equal to the sum of the two arrays sizes, and then copy everything over from the first array (has time complexity O(n)), copy from the second array (also is O(n)), so their combined time complexity is of O(n+m). This is obviously much worse for big n,m than having just one or the other.

stack using linked list

In assignment 2 we implemented a dynamic stack in an array, that had the two methods `push()` and `pop()`. We saw that the fix size of arrays meant that in certain cases we need to reallocate the array to have more space (and then probably decrease it back again), which affected our execution time. On the contrary, if we use a linked list to store the elements on the stack, both the `push()` and `pop()` will have constant time complexity. It would look something like this:

```java
public push(int val){
    if(this.first==null) this.first = new Cell(val,null);
    this.first = new Cell(val, this.first.tail)
}

public int pop(){
    if (this.first==null) return -1; //some kind of exception
    int val = this.first.head;
    this.first=this.first.tail;
    return val;
}
```

Since we don't require stacks to offer the ability to look up a value by index, it seems like a fitting choice to use linked lists. For the most common methods that a stack uses (push(), pop(), peek(), isEmpty(), clear()) we have O(1) since we only need to look at the first element. Size() on the other hand is O(n) (as it is with arrays in many programming languages, but not in Java), though we could easily keep track of it in a variable that is for example stored in a sentinel node. Another important thing to consider is the space complexity. Linked lists inherently use more memory space since every node stores not only the value but also a reference, which may also slow down the program by having to go further down the memory hierarchy more often.

Summary

In this assignment, we saw the main benefits of linked data structures, but also discussed some of their drawbacks. Linked lists can easily be used to implement abstract data structures beyond a stack too, such as queues or hash maps that we are going to see later in the course, but are also highly efficient if insertions/deletions at one (or both) end occur frequently. Despite the memory overhead, they also have a advantages with their memory usage, namely that they don't need to be stored sequentially, but can be scattered across memory, which may be beneficial when huge sequential memory blocks are not available. On the other hand arrays make it easier to delete elements at indices anywhere in the array, and also benefit from caches' spatial locality.
These comparisons have shown that neither are superior, but depending on our use-case, we need to be mindful of which one we choose.