Linked Lists in Java

Malte Berg - ID1021

HT 2024

Introduction

This report covers the fifth assignment assigned in the course ID1021. The assignment is to implement a working linked list, as well as analyzing the time complexity for appending dynamic sized and fix sized linked lists to each other. Furthermore, there will be a discussion of how a linked list compares to stacks and arrays as well as how to implement different features between the types.

Background

A linked list is a simple linked structure, where a number of different nodes or cells are in sequence. For this assignment, a cell contains an integer, the `head` and a reference to the next cell, the `tail`. Compared to an array however, only the first cell can be accessed as there is no index and the only way to search further down the list is by looking at the reference, or the tail, of the cells.

Implementing Linked List in Java

In the assignment, code was given for starting the class `LinkedList` by creating the private class `Cell` made up of an integer, `head`, and reference to the next cell, `tail` as well as a cell `first` being the first cell of the list used for navigating the rest of the list.

```java
public LinkedList() {
    first = null;
}
public LinkedList(int n) {
    Cell last = null;
    for (int i = 0; i < n; i++) last = new Cell(i, last);
    first = last;
}
```

As can be seen in the code above, an empty list can be created when passing no arguments when initializing a linked list. A linked list of size n can also be created by passing the wanted size which the second constructor handles with a loop. The remaining methods of the class which were asked to be implemented are as follows:

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

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

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

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

    if (current.tail != null) {
        current.tail = current.tail.tail;
    }
}
```

The last method of the class is the method `append`, which is used to append a linked list to another linked list.

```
public void append(LinkedList toAppend) {
    Cell current = first;
    while (current.tail != null) current = current.tail;
    current.tail = toAppend.first;
    toAppend.first = null;
}
```

The tail of the last cell of the first list points to the first cell of the second list, and the reference `first` of the second list is updated to `null` to make sure there is no copy of the list that has been appended.

Benchmarks

The first task of the assignment was to benchmark the program to find the time complexity of both appending a linked list of varying size to a fixed size linked list as well as appending a fixed size linked list to a linked list of varying size. The first case will be called "Dynamic to Fixed" and the second "Fixed to Dynamic". The fixed size will be set to 16, and the dynamic will go from 16 to 65 536 with the steps being the powers of 2.

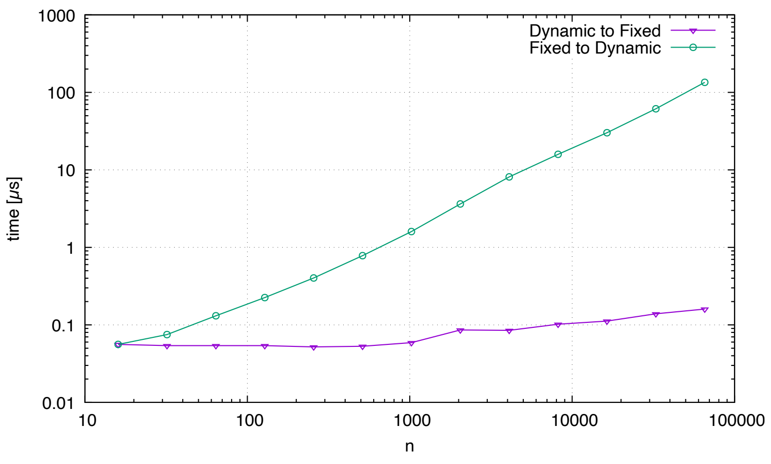

Figure 1: Time taken for linked lists of size n being appended/appended to, in microseconds

Looking at Figure 1 it can be seen that the best performing variant of appending is to append to a fixed size. In fact, the time complexity for Fixed to Dynamic is $O(n)$ while the time complexity for Dynamic to Fixed is $O(1)$. It is clear that the time taken to append one list to the other is completely dependent on the length of the list being appended to. The reason for this is due to the fact that the method is required to first go to the last cell of the first list, which of course takes more time as the list grows in size, and then point the `tail` of the last cell to the first cell of the second list.

A Comparison of Appending to an Array

One way of recreating the `append` method using arrays instead of linked list is to create a new array with a length of the two arrays to be appended. One would then loop through the elements of the first array adding them to the new array keeping track of the index. One would then continue from the next index and loop through the second array adding its elements to the new array. Now, the difference between appending a linked list and appending an array is of course the added need to loop through the elements of the array being added where the linked list being added does not require any looking at or copying.

The comparison of the time complexity would be $O(n)$ for the appending of a linked list, where n is the size of the linked list being appended to, and $O(n+m)$ for the appending of an array, where n and m are the sizes of the two arrays.

Looking at Stacks

Both push and pop, the two main functions of a stack, can be implemented using a linked list. One might be reminded of the method `add` in the class `LinkedList` which adds an item as the first cell of the list, similar to how one would push an item to the top of a stack, so one could argue the push method is already implemented. The pop can be implemented just as easily. Simply save the value of the item on ”top”, set `first` to `first.tail` and return the saved value.

A benefit of using linked list for implementing a stack is how linked lists are dynamic and new items can easily be added to the stack. Using arrays however has a downside here where for it to increase in size when it has grown to its limit, as arrays have fixed size, one is required to create a larger array and copy all of the elements over to the new array, taking $O(n)$ time.

A benefit of using arrays for implementing a stack is the fact that only the elements in the stack and one index variable for the whole stack is needed in terms of memory. Now the linked list implementation instead has a downside as every cell contains an item and a pointer to the next cell, being less memory efficient than a stack using arrays, in the ideal case.

The expected difference in execution time is negligible, until one arrives at the point where a stack based on arrays needs to grow or shrink in size, where one must copy over all the elements to the new array, where the linked list variant should perform better.