Linked Lists

Emanuel Paraschiv

Fall 2024

Introduction

In this assignment we will take a step back from arrays and introduce a new way of storing data, Linked Lists. We will explore the implementation, run some performance test and discuss the advantages and disadvantages of it.

Implementing a Linked List

A Linked List is a data structure that consists of individual memory locations with 2 components:

- a value to store (of type int in this experiment);
- a reference to the next location

A Linked List can be compared to a chain. It starts with one pointer that indicates the start of the list, called `first`. Then we have one to point for each next memory location (i.e. `Cell`) called `tail`. We know we reach the end of the list when the `tail` pointer in null. To create such a data structure we implement the following code:

```
Cell first;
private class Cell
    int head;
    Cell tail;
    Cell(int val, Cell tl)
        head = val;
        tail = tl;
LinkedList()
    first = null;
```

When constructing a Linked List, we therefore start with only a null reference. Thus, one important method to implement is one that can add new Cells to this list. I implemented such a method that adds elements from the beginning of the list:

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

As we add elements, the list will get bigger. In many applications it is important to be able to keep track of the length of such a list. In order to do so, I implemented a simple method labeled `public int length()` to return the number of cells in a list. It works by first checking if the list is empty, case in which it returns 0:

```
if (first == null)
        return 0;
```

Otherwise, if the list is non-empty, it iterates through each cells until it reaches the end, and returns the number of iterations (i):

```
    Cell current = first;
    int i = 0;
    while (current != null)
        i++;
        current = current.tail;
    return i;
```

Another interesting case is when we are dealing with 2 different lists and want to append one to the other. To do that, I implemented a method that achieves it by connecting the last cell of one list to the first cell of the appended list. It will first check if either of the lists is empty:

```
public void append(LinkedList b)
    if (b.first == null) // case 1: second list is empty
        return;
    Cell nxt = this.first;
    if (nxt == null)     //case 2: first list is empty
        this.first = b.first;
        return;
```

If it is not the case, then it will iterate all the way to the last element of the first list, and connect it to the first element of the second list:

```
    while (nxt.tail != null)
        nxt = nxt.tail;
    nxt.tail = b.first;
```

In the end, to not create any confusion, the method will delete the `"first"` pointer of the second list:

```
    b.first = null;
```

In order to investigate the execution time of the append operation, I ran a benchmark using the same principle as in the last experiments: create 2 linked lists, pass them to the `append()` method and measure the execution time using the `Sysytem.nanoTime()` method.

Input Size	**Execution in μs**
100	0.4
200	0.6
500	1.3
1000	2
2000	4
5000	10
10000	27

Table 1: b fixed a changing

In the scenario above I tested different size inputs for the first array. As we can see, the change in execution time is linear. It makes sense that the time complexity should be $\mathcal{O}(n)$ as the size of list **a** is increasing in the same manner.

Length of b	**Execution in μs**
100	2
200	1.9
500	1.9
1000	2
2000	2
5000	2
10000	2

Table 2: a fixed b changing

On the other hand, in the table above we can see the results for keeping **a** fixed and only changing the size of list **b**. The time complexity seems constant as expected, $\mathcal{O}(1)$. This is happening because the `append()` method only deals with iterating through the elements of **a**. Thus a change in the size of **b** should have no direct impact.

Linked List compared to Array

If we have to append 2 array, the process is similar. In order to do this, a third array is created where these 2 are copied and thus connected:

```
public static int[] append(int[] arrayA, int[] arrayB) {
    int n = arrayA.length;
```

```
        int m = arrayB.length;
        int[] result = new int[n + m];
        System.arraycopy(arrayA, 0, result, 0, n);
        System.arraycopy(arrayB, 0, result, n, m);
        return result;
    }
```

This time however, if the arrays are larger than the copying process will take longer: as the first array will grow in size, there will be more elements to be copied to the new array. The same is true for the second array, as we can see from the following results:

Length of b	**Time in μs**
100	0.9
200	1.4
500	3
1000	6
2000	12
5000	40
10000	90

Table 3: Length of b vs Time

Again, the change in execution time is linear, $\mathcal{O}(n)$. As the arrays get larger, it would take a greater amount of time to copy them.

Stack Linked List

Linked Lists also come in handy when dealing with Stacks. They offer a more versatile implementation than arrays do, as they can be more memory efficient in some cases.

Before, in a `'push()'` operation, if the array reached its capacity, we had to first create a new bigger array, copy the old one and then add the element. This approach can indeed result in a huge memory usage. However using a Linked List all we have to do is to create a new Cell and add it to the list, making the stack pointer point to it:

```
    public void push(int data)
        Cell newCell = new Cell(data, first);
        first = newCell;
        length++;
```

This way, the push operation takes a constant time now. Similarly, if

we want to pop an element all we have to do it move the stack pointer one position:

```
public int pop()
    // *throw exception if empty*
    int data = first.head;
    first = first.tail;
    length--;
    return data;
```

One inconvenience can however be the memory allocation. Cache performance can be poor on the linked list implementation because it is non-contiguous, and thus we can notice a difference in execution time for larger lists.

One more thing to consider with this implementation also comes down to memory. For each cell, we are using the memory twice: once for the actual data and once for the pointer to the next cell. Again, if we deal with large lists the memory usage will be noticeable. On the other hand, in an array implementation we won't have any additional overhead for such pointers.