Linked Lists

Dhruba Das

Spring Fall 2023

Introduction

In this assignment, we are tasked with the creation and benchmarking of a new data structure: linked lists. There are 3 problems in this assignment: creation of the linked list data structure (i.e Cell structure/its properties), implementation of methods given, and benchmarking against arrays of similar size;

Cell Structrue

The cell or node, is an object with a pointer to its next cell, and a value it contains. Below is the code snippet for cell class:

```
public class Cell {
        int head;
        Cell tail;

        public Cell(int val, Cell tl) {
            head = val;
            tail = tl;
        }
    }
```

As seen above, a constructor is made for the cell object, which allows us to change its pointer and the value it contains.

Implementation of the methods

For each method, I had to keep in mind that I had to traverse the entire list most of the time; this is important since we're only working with pointers, without any actual references to the cells themselves.

Add method

This was easy to implement as we only had to add an cell with a specific value to the beginning of the list. Below is the code snippet:

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

Length method

Here, we set a counter in order to count the number of cells we have traversed (i.e contained in the list) as we traverse the entire list, in order to find the size of the list. Below is the code snippet:

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

Find method

This method traverses the list in search of a cell which contains the value we are looking for. Below is the code snippet:

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

The time complexity of this search is O(n).

Remove method

This was the most difficult among the methods given to implement; it removes the cell containing the value we are looking for, from the list. In order to do so, we link the previous cell of the current cell we are looking at, to the next cell of the current cell we are looking at. It also contains a special case, for the instance that the value we are looking for is in the first cell. Below is the code snippet:

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

Append method

This method appends one list to another given list. Below is the code snippet:

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

Benchmarks

Constant list and Increasing list

I have represented the difference in performances between appending two linked lists, in a table below; note that the size of the list being appended increasing in size:

Size of b (n)	**Time(ns)**
200	2416
400	2875
800	2250
1600	2209
3200	2125
6400	2417
12800	1875
25600	2833

Table 1: (For each size, append is carried out for 1000 loops; the min. time (in nanoseconds) is taken; b is the list being appended)

Increasing list and Constant list

Next, below is the table for when the list being appended to is increasing in size:

Size of a (n)	**Time(ns)**
200	2375
400	2259
800	3000
1600	5584
3200	11125
6400	21625
12800	30833
25600	80583

Table 2: (For each size, append is carried out for 1000 loops; the min. time (in nanoseconds) is taken; a is the list being appended to)

And as seen above, in both cases, the time taken depends on the array being appended to, with a time complexity of O(n). If the list being appended to is fixed, the time taken to traverse the list stays approx. constant, whereas, when the list is growing, the traversal time increases.

Linked vs Array

Constant list and Constant array

I have represented the difference in performances between appending a constant and an increasing linked list, and a constant and an increasing array, in a table below:

Size(n)	Linked List (ns)	Arrays(ns)
200	2166	2417
400	2833	3125
800	2083	542
1600	2041	833
3200	2125	1416
6400	2250	2458
12800	2209	4750
25600	2292	9292

Table 3: (For each size, append is carried out for 1000 loops; the min. time (in nanoseconds) is taken)

Increasing list and Increasing array

I have represented the difference in performances between appending an increasing and a consant linked list, and a increasing and a constant array, in a table below:

Size(n)	Linked List (ns)	Arrays(ns)
200	2375	2458
400	2458	3125
800	2916	625
1600	4875	958
3200	8625	1541
6400	15917	2917
12800	30333	5375
25600	93084	10583

Table 4: (For each size, append is carried out for 1000 loops; the min. time (in nanoseconds) is taken)

As seen from the above tables, the cost of appending arrays is about the same, regardless of whether the array being appended is increasing in size, or the array being appended to is increasing in size.

Allocation

I have represented the difference in performances between allocating an array and a linked list, in a table below:

Size	Linked List (ns)	Arrays(ns)
200	1417	167
400	1041	208
800	2792	209
1600	6458	333
3200	13125	500
6400	26791	916
12800	55333	750
25600	107208	1125

Table 5: (For each size, allocation is carried out for 1000 loops; the min. time (in nanoseconds) is taken)

As we can see, the cost of building a linked list is much higher than that of an array.

Linked List Stack

Below is the code snippet for the linked list stack implementation:

```
public class LLStack{
    Cell first;

public class Cell {
        int head;
        Cell tail;

        public Cell(int val, Cell tl) {
            this.head = val;
            this.tail = tl;
        }
    }
    public LLStack(int n) {
        first = null;
    }

public void push(int val) {
Cell newC = new Cell(val, first);
first = newC;
```

```
}

public int pop() throws Exception {
if (first == null) {
throw new Exception("Tried to pop empty stack");
}
int tmp = first.head;
first = first.tail;
return tmp;
}
}
```

As we can immediately see, the linked list implementation of the stack data structure is much more efficient and requires a lot less memory allocation, especially when it comes to the push and pop.