Linked Lists

Jonathan Nilsson Cullgert

Fall 2023

Introduction

In this assignment we are supposed to create a linked structure and study it as a way to store data instead of using an array. A linked structure stores objects that have certain properties but the important part is that the also have a reference or pointer which points to the next object in the list. The pointers are one-way which means that the objected which is being pointed at does not know that is being pointed at. There are linked structures where the an object has two pointers, one pointer towards the next object and one pointer towards the previous object. In this assignment we will look at objects that only point to the next object, more specifically, a linked list.

Linked List

In a linked list there are cells or nodes which contain some data and a reference to the next cell. A linked list has these cells in a sequence but it can only look at the first cell. It can only traverse the list by following the references that each cell has. If the cell has a null-pointer or rather it references null, that cell is the final cell in the linked list.

A linked list can function exactly as an array meaning it can add elements, remove them, find the length of the list or search for a specific object. A linked list might even do this even more efficiently than an array. As an example it is not necessary to create a new larger array and then copy all elements and then **add** the new element. A linked list can just attach it at the end or beginning of the list. The same could be true for the other operations.

In this assignment the methods to **add()**, **length()**, **find()** and **remove()** are supposed to be implemented to the linked list. These methods are pretty self-explanatory, we should be able to add a cell to the list, find the length of the entire list, find if a cell exists in the list and we should be able to remove a cell from the list. Besides these methods **append()** should also be implemented which will connect the last cell in one linked list to the first cell on another list.

The important part in the **length()**, **find()** and **remove()** is to check if the first cell exists, if the second cell exists, and if they exist you can search through the linked list easily with a while-loop. As an example in the length() method we do these steps to find the length of the linked list

```
if (first == null){
    return 0;
}
if(first.tail == null){
    return 1;
}
while(nxt.tail != null){
    len += 1;
    nxt = nxt.tail;
}
len += 1;
return len;
```

This important in these methods because otherwise we they would not work if the linked list is empty or if the linked list only contains one cell. In the **add()** method it might not be obvious but the cell that is added will be added in the beginning of the list compared to the end in an array.

As mentioned previously the **append()** method is also added and this method is the one that is benchmarked to compare to an array. A linked list with a varying size is appended to the end of a linked list with a fixed size.

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

In this example the linked list with a fixed size has 100 cells and the varied one goes between 100-2000 elements in steps of 100. As we can see in 1 when the varying list is appended on the fixed list it is much quicker than the other way around. The reason for this is because it only has to go search through 100 elements and then it is linked together. If it is the other way around 100 to 2000 elements has to be searched through until the other list can be linked together. The time complexity when linking the varying onto the fixed has the time complexity of $O(1)$ while the other way around has the time complexity of $O(n)$.

Compared to an array

The next part is to compare this **append()** method between a linked list and an array. With arrays, an array is not appended onto another array. Instead there is one array with a fixed size and one array which varies from 100-2000 elements, these two arrays are then copied into a new array which is as big as the other two put together.

As we can see in 2 the time it takes to create a new array as well as copying the data from the first two arrays into the new one is faster than appending a fixed linked list onto a dynamic linked list. However appending a dynamic linked list onto a fixed linked list is still faster than appending arrays. The different time complexities become $O(n)$, $O(1)$ and $O(n)$ for fixed list appended on dynamic list, dynamic list appended on fixed list, and array appended onto array. This clearly shows the importance of which way you append the lists, while the arrays will have the same speed no matter if you add array a or b into the new array first.

In this benchmark I start the clock, create the linked lists, fill them then append them and stop the clock. For the arrays i start the clock, create all three arrays, fill two of them, copy that data into the third and then stop the clock. This shows that creating and filling arrays is much faster than creating and filling linked lists which makes sense. The reason that it makes sense is that the linked list uses data that is not necessarily connected closely in the memory. This means that there will be access made in different places in the memory which might not be cached and that will take longer than an array. The array will use a block of memory and when you access that array all of it will be cached given that the array is not very large.

These findings show that if you want to connect two sets of data using append with linked lists will be faster than arrays if you have a a fixed linked list and that the list is smaller than the other linked list which is dynamic.

```
for(int i = 0; i < tries; i++){
    double start = System.nanoTime();

    for(int j = 0; j < loop; j++){
        LinkedList listB = new LinkedList();
        listB.Linkedlist(100);
        LinkedList listA = new LinkedList();
        listA.Linkedlist(n);
        listA.append(listB);
        }
    double t = System.nanoTime() - start;
```

Stack

The final task is to implement the stack with linked lists instead of with arrays and to compare the two stacks. In this task no benchmarks are made instead we will only compare the different time complexities. The push and pop are similar to those methods with arrays but there are some small differences in how they are implemented which makes using these two structures very different. In an array you could just push and pop an element given that the array is not full or empty. With linked lists you have to search through the entire list to end to know either which cell to pop or which cell to the new cell that has been pushed onto the stack. This is obviously going to take a lot of time compared to the array since you have to do a lot of searching each time you push and pop. The different time complexities for the stack depending on if it is with a linked list or an array becomes $O(n)$ and $O(1)$.

The benefit of implementing the stack with linked lists is that unlike an array, it will not become full and there will be a stack overflow. You can always link a new cell onto the stack without having to allocate new memory for it. This means that if you have a stack which will vary in size a lot it is going to be much better to implement it as a linked list rather than an array. There will be no memory that is allocated unnecessarily to compensate for the largest the stack could become. There will not be the cost of increasing/decreasing the stack as there will be if you would implement a dynamic array stack. This shows that which method of the stack that is used depends heavily on what the intended use area is.

Benchmark

n	B append on A	A append on B
100	400	400
200	400	200
300	600	200
400	600	100
500	700	100
1000	1300	200
1500	2000	100
2000	2700	200

Table 1: A table which compares appending two linked lists one is fixed (100) and one changes size (n), and the time it takes to append one on the other. (ns)

n	TimeL	TimeA
100	1500	600
200	900	800
300	1500	1000
400	1900	1200
500	2600	1500
1000	4400	3300
1500	6500	4300
2000	8600	5600

Table 2: A table which shows the difference of appending two linked lists compared to arrays. (ns)