Linked List

Axel Månson Lokrantz

Spring Fall 2022

Introduction

This far we only worked with primitive data structures such as arrays. For the fifth assignment we were given the task to work with a linked data structures. In short a linked data structure is a structure where all elements are linked to each other using references such as pointers or links. The program was written in Java in collaboration with Axel Lystam.

Task 1: Appending Linked Lists

Task one was to measure the time it takes to append two linked lists to each other. Linked list (a) had a fixed size while list (b) had a dynamic size with a growing number of elements. For the first benchmark I made a fixed list (a) and appended a dynamic list (b) as its last element. Measurements and the function used to append can be seen below.

n	ns
200	50
400	43
800	58
1600	65
3200	61
6400	70
12800	80
25600	84

Table 1: Appending dynamic list (b) as last element of fixed list (a)

```
public void appendAfter(LinkedList list){
    Node current = this.head;
    while(current.next != null){
        current = current.next;
```

```
        }
        current.next = list.head;
    }
```

As seen in Table 1, the time complexity for the first benchmark is $O(1)$. Which makes a lot of sense, since the function appendAfter goes through the fixed list and assigns current.next (null) to the head of the parameter list. For benchmark two I instead appended the dynamic list to the head of my fixed list. To do this operation I had to iterate through all nodes in the dynamic list until I found the last node which I assigned as the head of my fixed list.

n	ns
200	2380
400	3601
800	7433
1600	11025
3200	16868
6400	24922
12800	40506
25600	73434

Table 2: Appending dynamic list (b) as first element of fixed list (a).

```
public void appendBefore(LinkedList list){
    Node current = list.head;
    while(current.next != null){
        current = current.next;
    }
    current.next = this.head;
    this.head = list.head;
}
```

The time it takes to perform the operation becomes $O(n)$ since we have to go through n number of elements in the dynamic list to reach the last element. A "smarter" way of doing this benchmark would have been to not only have a pointer at the list head but also at the list tail, to always keep track of the last node in the list. With such a code the benchmark for appendBefore would have resulted in a similar time complexity of $O(1)$.

Task 2: Appending Arrays

The second task was to do the equivalent operation as in task 1 but for arrays. For this function I created two arrays, one with a fixed number of

elements (a) and and another with a dynamic number of elements (b). I then created a new array with the combined length of a and b. Through two for loops I then filled the new array with the elements of my fixed and dynamic arrays.

```
public static int[] arrayAppend(int[] array1, int[] array2){

        int arraySize = array1.length + array2.length;
        int[] newArray = new int[arraySize];
        int j = 0;
        for(int i = 0; i < array1.length; i++){
            newArray[j] = array1[i];
            j++;
        }
        for(int i = 0; i < array2.length; i++){
            newArray[j] = array2[i];
            j++;
        }
        return newArray;
    }
```

I did a similar test where I appended the dynamic array (b) to the last element of the fixed array (a) but this operation yielded the same result as the above benchmark. The time complexity for merging two arrays becomes $O(n)$. To further evaluate the differences between the two structures I implemented a new function to measure the time variance of building a list structure compared to an array structure where both have n elements.

To create a list of n elements I simply repeated the addNode function n number of times.

```
    public void addNode (int value){
        Node newHead = new Node(value, this.head);
        this.head = newHead;
        this.size++;
    }
```

n	ns
200	13027
400	23078
800	43639
1600	78493
3200	162688
6400	306638
12800	550496
25600	1031685

Table 3: Building a linked list of n elements.

As seen in table 4, it is quite costly to build the list structure compared to an array. For a growing n the time complexity becomes $O(n)$.

Task 3: Dynamic Stack vs. Linked List

In assignment 2 we worked with a array and a dynamic stack that would expand and collapse. In the third and final task we were asked to implement a stack data structure with the regular push and pop operations and compare it to the dynamic stack. My push function remained the same as addNode. The code for the push function can be seen below.

```
public void removeNode(){
        if (isEmpty()) {
            System.out.println("List is empty");
        }
        this.head = head.next;
        this.size--;
    }
```

Through these two functions pushing or popping nodes will always be done at the top of the stack.

```
5 <- Top of stack
4
3
2
1
```

It is clear that both of these structures have their respectively weaknesses and strengths. For example one of the disadvantages of arrays is wasted memory. Often the programmer does not know how many inputs the user

will make. Lets say you initialize an array with thousand indexes but the user only inputs 10 elements in the array, then the majority of the allocated space is wasted. Of course a dynamic stack would mitigate some of the loss, but some memory loss would still be there. Linked lists however, are not pre-defined, allowing the structure to increase / decrease in size as the program runs.

Arrays also have slow insertion and deletion times compared to the list structure. As seen in task 1, the time complexity for inserting a new node to the beginning or end of a linked list takes constant time of (O(1)).

Searching for elements is where the array structure shines, through indexing it is easy to find a particular element. As for linked lists random access is not allowed, which means we have to iterate through the list in order to find what we are searching for.