Linked List Report.

Adrian Jonsson Sjödin

Fall 2022

Introduction

The purpose of this assignment is to gain a better understanding of pointers and references and how they can be used to create more complicated data structures. In particular we will gain a deeper understanding of how linked lists functions.

Task

Implement a linked list class from the ground up that utilizes a stack structure. Also create a method that allows one to append another linked list to said created linked list. Having implemented that, benchmark the run time of the append operation. Vary the size of the first linked list **a** and append it to a fixed size linked list **b** and examine how the run time changes with the size of list **a**.

Lastly implement the equivalent append operation using arrays and benchmark this operation. How does this compare to the append operation for the linked list class? Without doing any measurements, describe the difference in execution time for this linked list stacked as compared to the stack implemented using arrays from the previous assignment.

Method & Theory

I implemented the linked list stack using a private helper class to create nodes that contains the value we want to add to the list, and a pointer to the next node in the list. So when a linked list are created it is created empty and when we want to add a value to it we create a new node that will then have a pointer *next* that will be null for the first node. But since every new node will be created and added to the left of it in the list, they will have a pointer to the previous node already in the list. Figure 1 shows how this would look like and code overview 1 shows the code implementation of this.

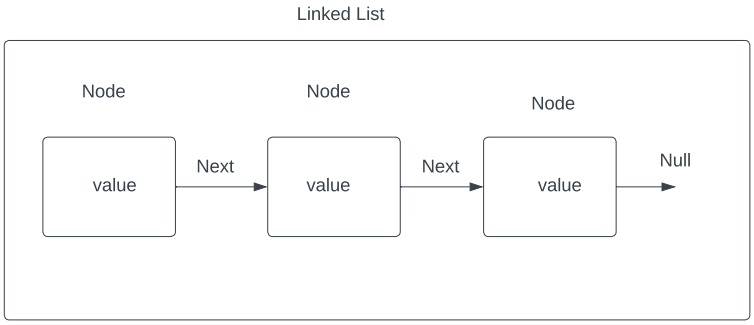

Figure 1: Depiction of node implementation in a linked list

When we want to add a new value to the list, the linked list class then only need to keep track of the *head* node and rereference it to point at the new node, and make sure that the new node point at the old head node. The old node in turn already has a reference to the node before it and thus there's nothing else that needs to be done. This operation should take constant time ($\mathcal{O}(1)$) regardless of how big the list already is since we don't need to do any kind of operation on the list apart from rereferencing the old head node. The code implementation of this can be seen in code overview 2.

A push operation was also implemented so that the last inserted value could be removed. It works on the same principle as the add operation and for those interested a link to the full code will be included at the end of this report.

For the append method that should append one linked list to another, two different methods where created. One where we append the list to the end of the first list, and one where we instead append it to the front of the first list. In theory appending the list **a** that varies in size to the end of the fixed size list **b** should have a time complexity of $\mathcal{O}(1)$. This since list **b** is fixed and we simply need to step through that list until we reach the node that has a null pointer and rereference it to point to the head of list **a**.

Appending list **a** to the beginning of list **b** however should have a time complexity of $\mathcal{O}(a)$, where a is the size of list **a**. This is because just as in the other append method we need to first step through list **a** until we reach the null pointer. We then need to make it point to the head of list **b** and also move the head pointer for list **b** to point to the head of list **a**. An illustration of this can be seen in figure 2 and the code can be seen in code overview 3.

Lastly we implement the equivalent append method for appending two arrays. I implemented this in accordance to the instructions provided in the assignment. This method should have a time complexity of $\mathcal{O}(a+b) = \mathcal{O}(n)$, where a and b is the size of array **a** and **b**. This since we need to loop through

both arrays and copy each element to a new array big enough to hold all elements from both arrays.

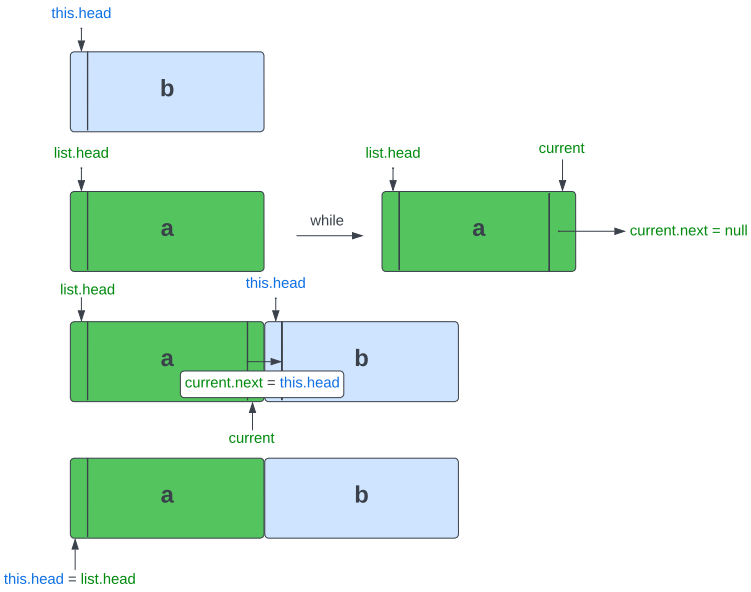

Figure 2: Diagram of how the append first method works

Code Overview 1: Class Structure

```
public class LinkedList {
    private int size; //track size of list and used to see if list is empty
    private Node head;
    private class Node {
        private int value; // the value for the item we add
        private Node next; // pointer to next element in the list
        public Node(int value, Node node) {
            this.value = value;
            this.next = node;
        }
    }
     public LinkedList() {
        this.size = 0;
        this.head = null;
    }
}
```

Code Overview 2: Add integer to stack

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

Code Overview 3: Append to front of list

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

Result

Figure 3 shows the minimum time for one add operation out of a thousand add operations done 10,000 times for each benchmark. On average one add operation took 49.9 ns.

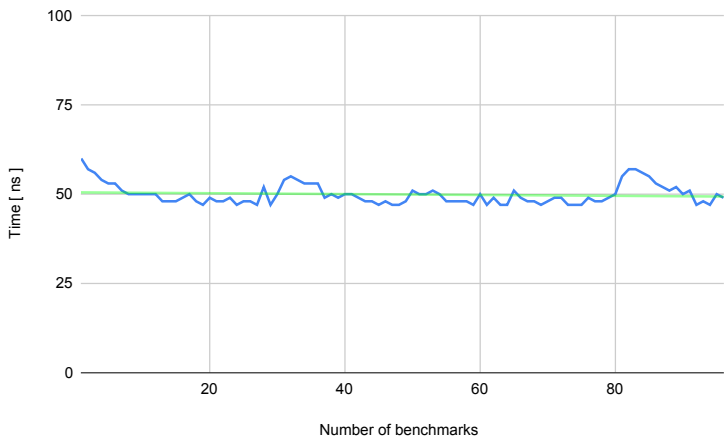

Figure 3: Time for one push operation

Figure 4 shows the benchmark data for appending a growing list to a fixed list. The list increases in size by 10 for each benchmark up to a size of

1000, and the time is the minimum time from doing this append operation 10,000 times in each benchmark.

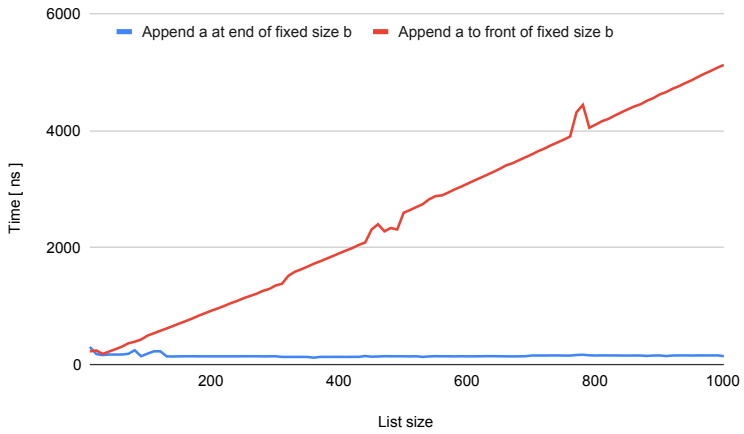

Figure 4: Time it takes to append growing list **a** to fixed size list **b**

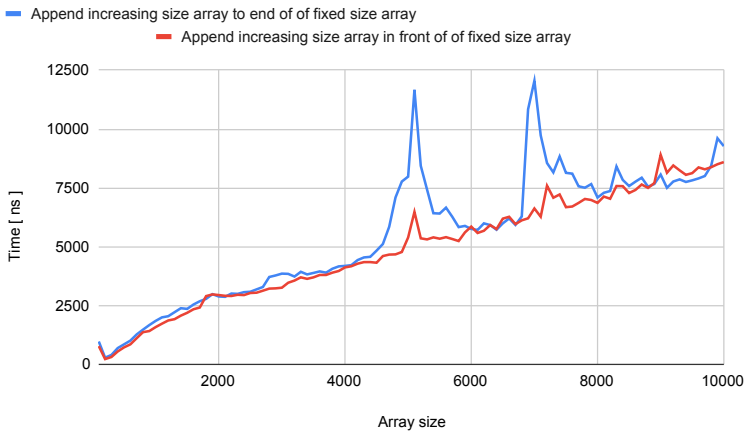

Figure 5: Time it takes to append growing array **a** to fixed array **b**

Figure 5 shows the benchmark data for appending a growing array to a fixed size array. The array increases in size by 100 for each benchmark up to a size of 10,000, and the time is the minimum time from doing this append operation 10,000 times.

In both the benchmark for the list and the array, the benchmark are run on a new unmodified list and array.

Discussion

Starting with the benchmark data for the push operation we see that the operation is as expected of time complexity $\mathcal{O}(1)$. The deviation that can be seen in figure 2 is around ± 10 ns and probably do to other process running in the background of the PC and slightly affecting the result.

The benchmark data for the append method for lists also support our expected result, with it giving us a constant time to append an increasing size list at the end of a fixed list, thus giving us a time complexity of $\mathcal{O}(1)$. When the list instead is appended to the front of the fixed size list the time execution is linear, which was expected and support our hypothesis of it being of time complexity $\mathcal{O}(n)$. There's a small spike in our data just before the list becomes 800 nodes large. This could be because of a number of different factors such as the program need to make a big jump in memory to go to the next node in the list or that java's garbage collector is executing in the background. Whatever the reason is the rest of the data support our hypothesis and it is safe to assume that it has no bearing on our conclusion.

The append method for arrays are according to the benchmark data also linear and of time complexity $\mathcal{O}(n)$. However we seem to have two strange spikes in the data at around array size 5000 and 7000. The spikes are much more prominent when we append the increasing size array to the end of the fixed size array, but they are there for both methods. I do not know exactly why these spikes arises, but a guess would be that java's garbage collector get started and starts cleaning up all the old arrays. After all when we run the benchmark we create a lot of arrays that are used just one time before being thrown away. This could likely fill up the memory and thus require java's garbage collector to step up and make more space available, slowing the append method down and thus generating those spikes.

The data also shows us that the append operation for arrays are actually faster than for lists. When the array is of size $\sim$2000 the append operation takes around 2,6μs while it takes around 5μs to append a list of size 1000. A reason for this could be that it goes faster to loop through an array and write each element to a new array than it takes for the list to go through each node and check if it is the node with a null pointer. This however is just a guess.

When it comes to which stack that would be preferred I would say that a dynamic array stack should be faster. The linked list stack does have a time complexity of $\mathcal{O}(1)$ for both push and pop, however the constant factor of these $\mathcal{O}(1)$ terms seems to be higher, which is probably due to the expense of dynamic allocations. Each element must have a pointer to the next element which require extra storage, and they might also not end up saved next to each other in the memory. In a dynamic array we can access any element in $\mathcal{O}(1)$ time and also append an element in amortized $\mathcal{O}(1)$ time. This means that the most common implementation of a dynamic stack array has

best-case $\mathcal{O}(1)$ push and pop, worst-case $\mathcal{O}(n)$ push and $\mathcal{O}(1)$ pop, and amortized $\mathcal{O}(1)$ push and $\mathcal{O}(1)$ pop, while a list stack has a worst and best case of $\mathcal{O}(1)$. However each new element added to the list stack requires a new allocation which seems to be more expensive.

All the code can be found here: GitHub