Linked Lists

Aditya Gupta

September 2022

Introduction

This assignment explores how the node-based dynamic data structure called Linked Lists works. We check how efficient this data structure is at being able to append new nodes or lists to an existing list. We benchmark the append and the append list operation for the list and compare its time complexity against a more basic data structure such as an array. We do this for two cases, wherein in the first case we vary the length of the first list and keep the length of the second constant, and in the second case, we vary the length of the second list and keep the length of the first one fixed. After this, we attempt to create a dynamic stack using the linked list data structure that we have created and compare it against the previous version we made in the HP-35 assignment using arrays.

Linked List

The linked list is a node-based linked data structure which is similar to an array in the sense that it can store values in a list format. It has an advantage over the array: While in the case of an array which is a static data structure, one has to specify its length from the beginning while in the case of the linked list you can dynamically continue to increase its size without having to go through the trouble of having to copy all your existing elements into a larger data structure. But this comes at the cost of losing the freedom to have random access to all the elements inside your data structure as the linked list does not have an indexed based system i.e. the elements stored in this are not stored in contiguous memory, they can be at different locations entirely and not in consecutive memory spaces like an array. Here is my implementation of the linked list:

```
public class Node {
    public int data;
    public Node next;
    public Node(int data){
        this.data = data;
    }
    int getData(){
```

```
        return this.data;
    }
    Node getNext(){
        return this.next;
    }
}
```

I create a node class which will serve as the building blocks for the linked list. Each node contains data and a reference or a pointer to the next node. Now in the Linked list class, we create a Node object called head which will serve as the first element in our Linked List class. In the linked List class we add a few functions for the purposes of appending elements to the end of the Linked list and we create a method that allows us to append another Linked List to the end of our current linked list.

appendData():

```
    public void appendData(int data){
        if(head == null){
            head = new Node(data);
            return;
        } else{
            Node current = head;
            while(current.next != null){
                current = current.next;
            }
            current.next = new Node(data);
        }
    }
```

Here we first check if the head is null or not to check if the linked list is originally empty. This means we will now set the value of the head to the data that the user wants to add to the current list. Otherwise, if the list has at least one element, then we create a temporary variable that stores the value of the head and uses it to iterate through the list. We do this by checking if the value of the current is null or not (this is to check if in case we might have reached the end of our list), and update the value of the current to its next value so as to progress inside of our list. We keep going until we reach the end of the list and then we set the next value of the last element such that it points to a new node that contains the data passed by the user.

appendList():

```
public void appendList(SinglyLinkedList singlyLinkedList) {
        if(head == null){
            head = singlyLinkedList.head;
```

```
            return;
        }
        else if(this.head == singlyLinkedList.head) return;
        else{
            this.appendNode(singlyLinkedList.head);
        }
    }
```

This function makes use of the appendNode() function which is just a variation of the appendData() function with the only change being that rather than creating a new node each time we set the value of the head or the current node to the node variable that has been given to the function by the user as an argument. Here we traverse to the end of the first list and just append the head node of the second list to the first one.

Benchmarks

Task 1

Now it's time to benchmark the two cases provided in this week's task. Here the first task's first case was to append a fixed-sized linked list to a linked list that grows in size. While in the second case we benchmark the opposite case where we have a linked list of varying sizes that gets appended to the first list that has been set to a fixed size. The data for the two cases have been tabulated and graphed below:

Case 1 : A - Increases ; B - Fixed

Case	n	Time(in microseconds)
1	250	1.16
2	500	2.15
3	1000	4.37
4	2000	7.80
5	4000	15.16
6	8000	31.76
7	16000	67.65
8	32000	123.70

Here we see this trend as to append the list B to list A we traverse the entirety of List A as use the appendList() mentioned before which appends the head of the second list to the end of the first one. List A is increasing in size in a linear fashion with a factor of 2. Hence the time taken to traverse list A should also increase by a factor of approximately 2 which is what we can clearly see from the above table as well, which explains the linear trend we see in the time taken to complete the appending of the List.

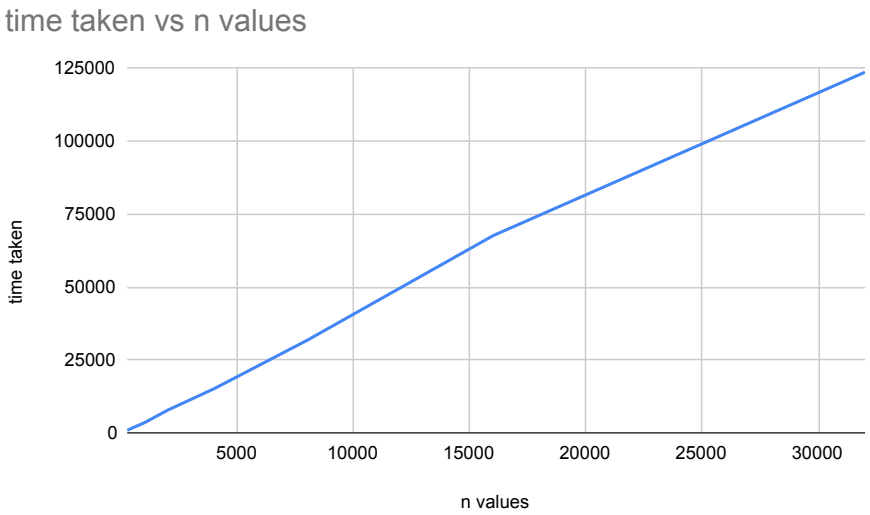

Figure 1: Linked Lists Case 1: A - Increases; B - Fixed

Case 2 : B - Increases ; A - Fixed

Case	n	Time(in microseconds)
1	250	0.22
2	500	0.28
3	1000	0.38
4	2000	0.55
5	4000	1.17
6	8000	1.66
7	16000	2.33

Here since the length of the first list remains fixed the time taken to append any list should remain the same as we always traverse the same amount of elements each time. However, from the infographics above it is clear that the time taken to append a list with an increasing size grows linearly with a factor of about 1.6.

The reason for such an increase might be attributed to some overhead time taken while calculating the benchmark. This might also be associated with caching. That is for smaller values of n we are able to quickly retrieve data from memory while as the value of n increases this data retrieval time increases and thus we end up seeing a linear trend instead.

The time taken might also be increasing in such a fashion due to the Java Garbage collection taking place on a separate thread which might be taking up

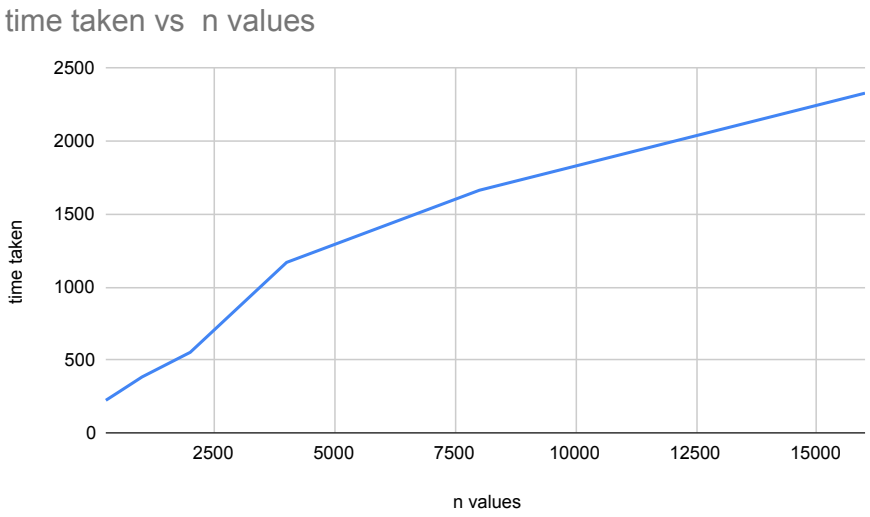

Figure 2: Linked Lists Case 1: B - Increases; A - Fixed

more and more time as the size of our linked lists increases. But this is merely speculation as we cannot check which of the above factors is truly responsible for this strange phenomenon.

All Data

Case	n	Time Case 1(in microseconds)	Time Taken Case 2(in microseconds)
1	250	0.86	0.22
2	500	1.76	0.28
3	1000	3.50	0.38
4	2000	7.80	0.55
5	4000	15.16	1.17
6	8000	31.76	1.66
7	16000	67.65	2.33

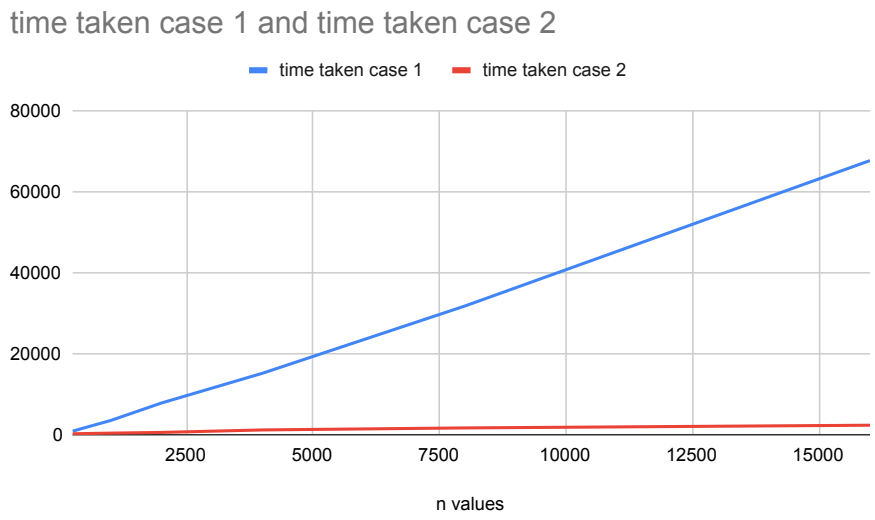

Figure 3: All Data

Task 2

This task involves us doing the same task as before but this time we choose to do it using arrays. We use arrays to represent both lists, and we have the same 2 cases as before where we vary the size of the second list while keeping the first constant and vice versa. The data for the two cases have been tabulated and graphed below:

Case 1 : A - Increases ; B - Fixed

Case	n	Time(in microseconds)
1	500	1.77
2	1000	4.78
3	2000	9.37
4	4000	18.41
5	8000	41.48
6	16000	109.30
7	32000	233.31

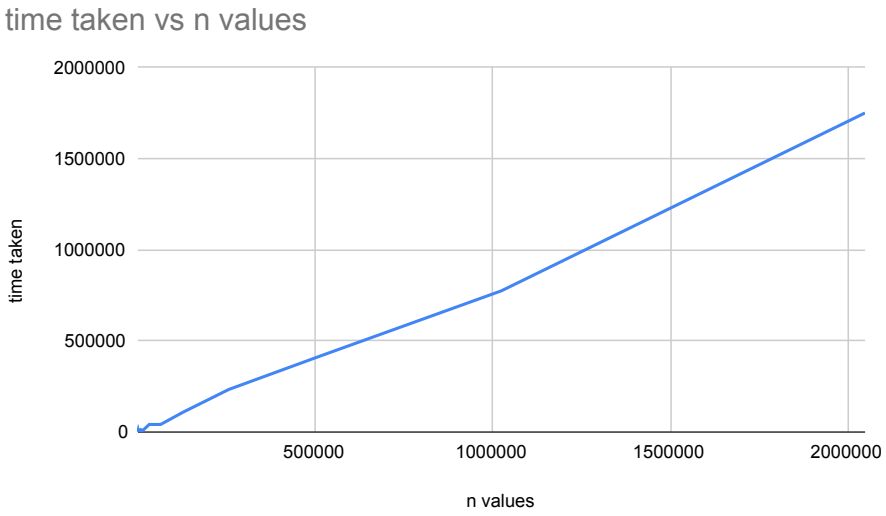

Figure 4: Arrays Case 1: A - Increases; B - Fixed

Here we see this trend as to append list B to list A we need to copy all the elements from list A and List B into a larger array of size = (Size of List A + Size of List B). This means we always need to do operations corresponding to copying all the elements from both lists which would have a time complexity of $O(n)$. We see this also from the tables and graphs that the time taken for doing this task seems to follow a linear trend.

Case 2 : B - Increases ; A - Fixed

Case	n	Time(in microseconds)
1	250	1.16
2	500	2.14
3	1000	4.37
4	2000	8.96
5	4000	18.74
6	8000	33.77
7	16000	68.55

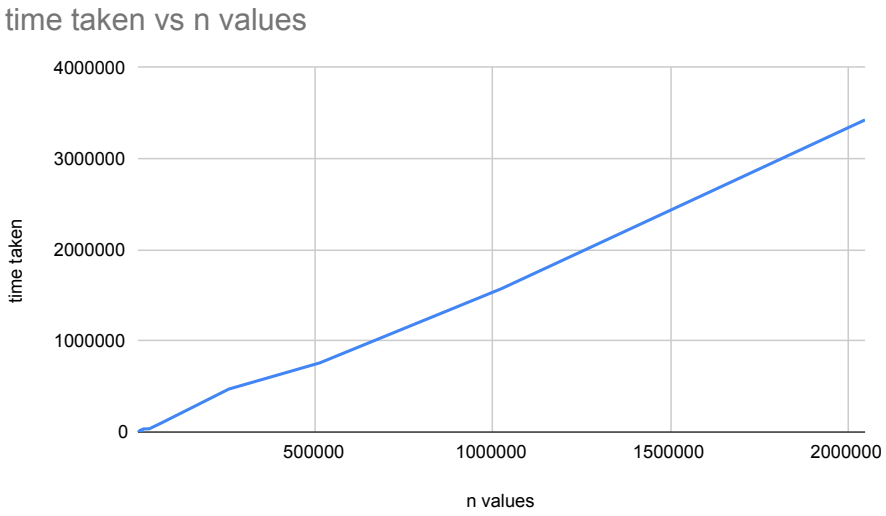

Figure 5: Arrays Case 1: B - Increases; A - Fixed

Since we need to copy both lists into a new list of a larger size this means that the overall time taken to copy things into the list remains in the same time complexity as the previous case, thus we end up seeing a similarly linear trend for this case as well.

Comparison between Linked Lists and Arrays

In the case of the linked lists, the time taken is the same as traversing the length of the 1st list in each case (Increasing length as well as in case of Fixed Length). While in the case of arrays it is the same as copying both the lists into a larger array with size = sum of the length of list A and list B. Thus we end up doing more work to do the same task using an array as compared to a linked list.

Here we can see the benefit of a dynamic data structure as compared to a static one such as an array. The fact that linked lists are easily modifiable to contain more elements makes them an ideal choice for the above tasks. This also means that we don't have to pre-allocate more memory for the arrays to make sure we don't have to keep copying to larger arrays and hence saving some memory in the process. Thus at least for the purpose of the above task, linked lists are far better suited to this job than arrays.

Dynamic Stack using Linked Lists

We made a dynamic stack data structure using an array in the second assignment where we used it to make the HP-35 calculator. We quickly realized some flaws in implementing this data structure using an array. We noticed that every time we ran out of space in the internal stack array we need to replace it with an array of twice the size which meant we need to copy all the elements from the original stack and place them into the new larger stack.

This has multiple problems. This means that in case we just needed space enough to store just one more element than the initial max capacity then we end up wasting a lot of memory space that could have been better utilized by some other task. Besides this, we also waste time copying elements from the original array and writing them at a new location. We also have the same problem when we want to shrink the size of the internal array as there as well copy all the elements and replace the internal array with an array of half the size and put the copied elements inside of it. This process is not only tedious but also quite wasteful in terms of space used.

To make the dynamic stack more optimized we should use a dynamic data structure for the internal storage mechanism. Here we can use linked lists which lend themselves very well for this purpose. The push and pop operations can be performed in constant time if we implement them such that we always add and remove elements from the head or the top in this case. As we always have access to the first element in the linked list we can easily prepend it or remove it without having to traverse through the linked list.

This means we get to push and pop in constant time. Aside from this, we will not encounter the problem with stack overflow where we need to copy ele-

ments to a larger array, we just simply add a new node whenever we need to add another element. Thus there is theoretically no cap over how many elements there can be inside of the stack. The same goes for shrinking, we don't need to prompt the storage space to grow or shrink as in this implementation we will always have enough.

push():

```
public static void push(int data){
    if(top == null){
        top = new Node(data);
    }
    else{
        Node temp = new Node(data);
        temp.next = top;
        top = temp;
    }
}
```

Here we implement the stack such that the top element in the stack is supposed to be the head of the singly linked list being used as the internal stack. In case of the push() operation, we just prepend the head by storing the data to be pushed as a temporary node whose next element is set to be the current top element and after that we make the top node refer to the newly added node as the head, thus completing the push operation and adding the element onto the stack.

pop():

```
public static Node pop() throws StackUnderflow{
    if(top == null){
        throw new StackUnderflow();
    }
    else{
        Node temp = top;
        top = top.next;
        return temp;
    }
}
```

We will not encounter a stack overflow error but we can still get a stack underflow if the stack is empty and the user tries to pop elements that don't exist. For this purpose, we create an error class called StackUnderflow, which is thrown whenever the stack is empty and this function is called. If the stack is not empty we just replace the top element with its next element and return the value which was at the top position before.