Implementing Linked Lists

Abdinasir Abdi

September 2024

1 Introduction

This report concludes the fifth assignment in course ID1021 taken in the fall of 2024. This report aims to explore how linked lists implement linked data structures and compare them to primitive data structures and how efficient they are.

2 Linked Lists

A linked list is a data structure used to store a sequence of elements. It consists of a sequence of elements (nodes or cells), each of which contains data and a reference (or link) to the next element. The functionality of how elements are sorted in linked lists differs from arrays and stacks. There are no contiguous memory locations for the objects in a linked list. Instead, The elements are stored as nodes, each pointing to another node with an element and associated pointer. In the first benchmark, a static list of fixed size is appended(linked) to a list of varying sizes. The following results were obtained through the benchmark:

Iterations	**Size(n)**	**Static Size**	**Average Runtime**
1000000	100	1000	227
1000000	200	1000	416
1000000	400	1000	811
1000000	800	1000	1664
1000000	1600	1000	3501
1000000	3200	1000	6905

Table 1: A linked list where a static fixed list is linked to a varying list, measured in ns

The results show that the runtime increases proportional and confirm the time complexity to be linear $O(n)$. The whole array is traversed, from the first until the last element in which the pointer is altered to point to the first element in the second list. Therefore the measured average for each list size follows a conditional and expected pattern.

In the next benchmark, the actions reversed and the varying-sized list was linked to the static-sized list. The following results were obtained from the benchmark:

Iterations	**Size(n)**	**Static Size**	**Average Runtime**
1000000	1000	100	4.7
1000000	1000	200	4.7
1000000	1000	400	4.7
1000000	1000	800	4.7
1000000	1000	1600	4.7
1000000	1000	3200	4.7

Table 2: A linked list where varying sized lists are appended to a static list, measured in ms

The size of the appended list does not affect the method's runtime, which confirms that the time complexity is constant $O(1)$.

From the benchmark, one can see that the runtime for the method is dependent on the size of the list that is appended. The logic behind it can be explained by taking a look at the method itself:

```
public void append(LinkedList b) {

        if (first == null) {
            first = b.first; // point to b's first if empty
        } else {

            Cell current = first;
            while (current.tail != null) {
                current = current.tail; // Traverse to the end of this list
            }

            current.tail = b.first;
        }
        b.first = null; //
    }
```

In all cases, it iterates through the list and finds the last element where the address for the new list is appended. Traversing through the list is the factor that decides the runtime for the method. Appending the new list is a small operation and is therefore negligible.

3 Linked list using arrays

An array stores elements contiguously, meaning each element is located next to the previous element. By mapping an element's index directly to its memory location, array searches are very efficient ($O(1)$ time complexity), allowing for fast random access. Arrays offer a better performance for accessing elements by index but are fixed in size and resizing requires creating a new array. For this part of the assignment, a linked list was implemented using arrays, also called *Arraylist*. It dynamically seizes the array and allows flexibility in adding and removing elements. For the next benchmark a static fixed-sized array with 1000 elements and a varying-size array(n), are copied and together put into a larger-sized array. The following results were obtained from the benchmark:

Iterations	**Size(n)**	**Static Size**	**Average Runtime**
1000000	100	1000	102
1000000	200	1000	197
1000000	400	1000	411
1000000	800	1000	835
1000000	1600	1000	1701
1000000	3200	1000	3321

Table 3: Appending arrays, measured in ns

The results confirm the time complexity to be linear $O(n)$. The graph below gives an illustrated picture of the efficiency compared to previous collected data.

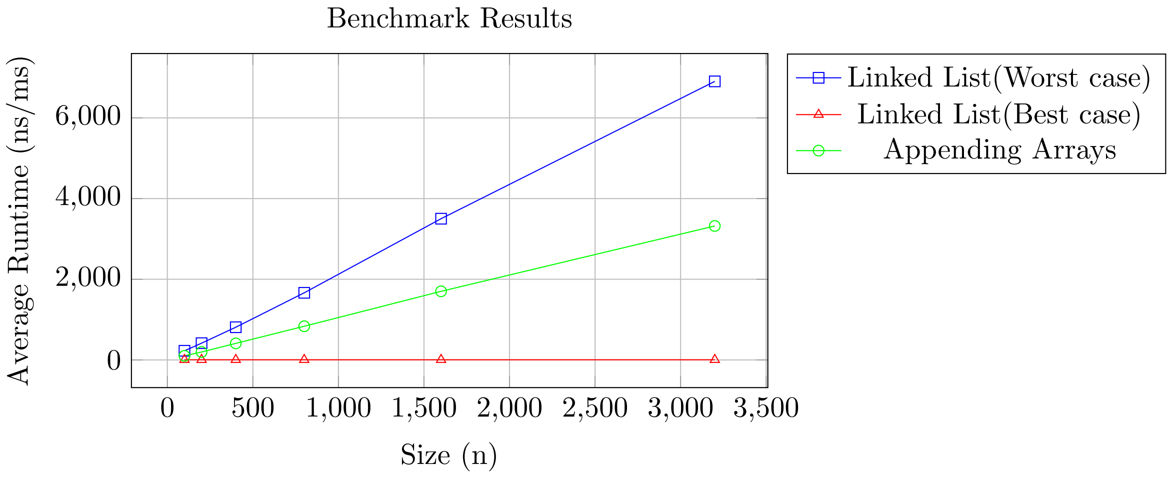

Figure 1: Benchmarking Linked List and Array Operations

4 Stack

Creating a stack using a linked list implementation offers several advantages, especially when it comes to dealing with dynamic data. A linked list construct allows the stack to grow and shrink dynamically and therefore no need to explicitly define an initial size or fear exceeding capacity. Memory is only allocated as needed, this reduces wasted space compared to an array which allocates more space to avoid frequent resizing. Both insertion(push) and deletion(pop) operations happen at constant time $O(1)$. This is because operations only involve adjusting pointers and elements no longer need to be shifted like in an array-based structure.