Linked List

Ayda Ghalkhanbaz

Fall 2023

Introduction

Implementing linked lists and benchmarking them is the main focus of this assignment. First we will explore what a linked list is and how it is implemented. Later in the discussion section, we will analyze the time taken to append two linked lists, and compare these results with the case of appending two arrays together.

Linked List

A linked list is a data structure that stores a collection of nodes (cells). In a singly linked list, each node contains two main attributes: a data and a pointer to the next node. A basic structure of a singly linked list is shown in figure 1. Due to its efficiency in memory usage, linked lists are very useful when the size of the collection has to change frequently.

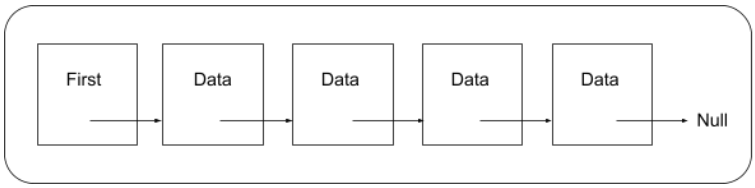


Figure 1: A basic illustration of a singly linked list structure

Method

The skeleton of the LinkedList class was already given in the assignment. This class contains a private helper class Node which creates a node with a data and a pointer next, and some methods such as add,find,remove and append.

The 'add' method accepts a value as an argument, creates a single node with that value and adds this node to the beginning of the list.

The 'find' method returns 'true' if the sought item is in the list, otherwise it returns 'false'.

The 'remove' method traverses the list until the target item is found, then makes its previous node point to its next node. In this way, the item to be removed will not be visible in the list.

The 'append' method simply adds a linked list at the end of another one. In order to do that, we will first traverse the entire first linked list, once the last node in the list is found, we adjust it so that the last node points to the first node of the second list. The code snippet below demonstrates the implementation.

```
public void append(LinkedList b){
    Node currNode = this.first;
    // loop until currNode holds the last node
    while(currNode.next != null){
        currNode = currNode.next;
    }
    // currNode points to the second list
    currNode = b.first;
    b.first = null;
    this.listSize += b.listSize;
}
```

Result

The benchmarking of this task included measuring the running time when appending a linked list with a growing length to a fixed length list and vice versa. The results are displayed in figure 2 respectively figure 3.

Figure 2. Results of appending a growing linked list to a fixed list

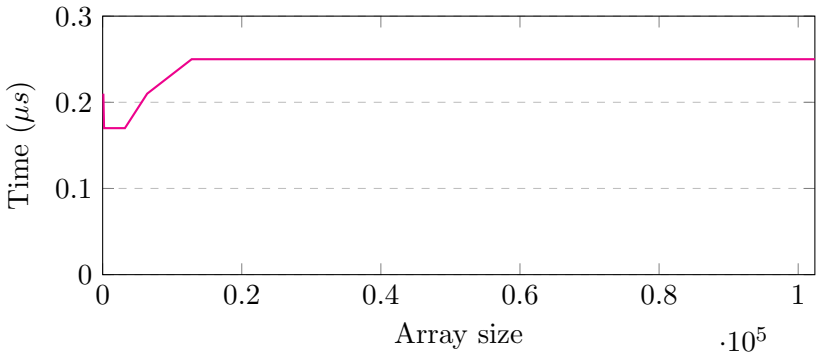
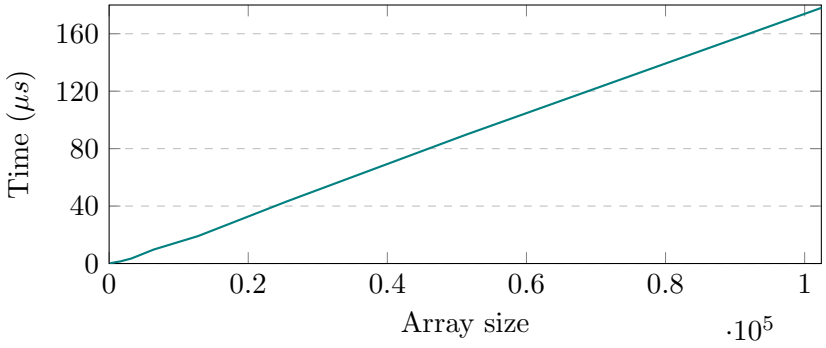


Figure 3. Results of appending a fixed size linked list to a growing list



Discussion

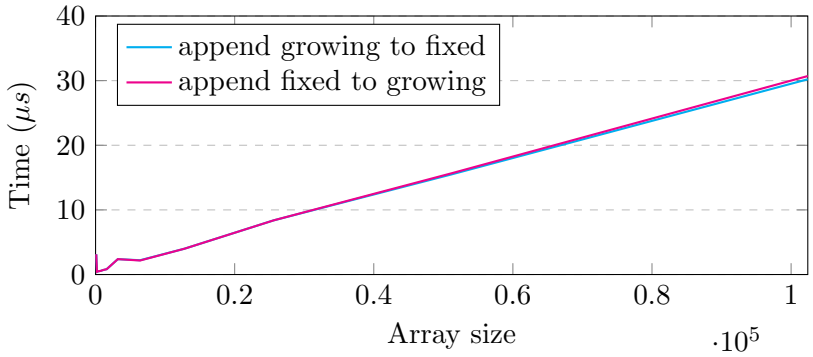
The outcomes of this benchmark which are shown in figure 2 and 3 are quite accurate with what we could expect in theory. As explained above, the 'append' method traverses a linked list and adds another list at the end of the first one. If the first list has a fixed size, it really doesn't matter whether we append a list at the end of it with size one or a one million; the required time remains consistent. In other words, we only need to traverse the fixed list and this will take the same amount of time every time we try to append a list to it. As illustrated in figure 2, we can observe that the running time is indeed constant. With the observation from the benchmark and the theory behind linked lists, we can verify that the time complexity of appending a linked list to another with a fixed length is $\mathcal{O}(1)$.

On the other hand, the running time varies when appending a linked list to another with a growing length as shown in figure 3. Every time we want to append a list to the dynamic list, we have to traverse the list, that is, if the list has a size of 10 then it will take less time than a list with a size of one million. This behaviour leads to a time complexity of $\mathcal{O}(n)$ where 'n' represents the length of the list.

Compared to an array

In this task we benchmark linked lists against arrays using an equivalent implementation of the 'append' method. When appending an array to another, we allocate a new array with a size that is large enough to contain the values from both arrays. The benchmarking of appending arrays was done with the same principle similar to the previous task and the results are demonstrated in figure 4.

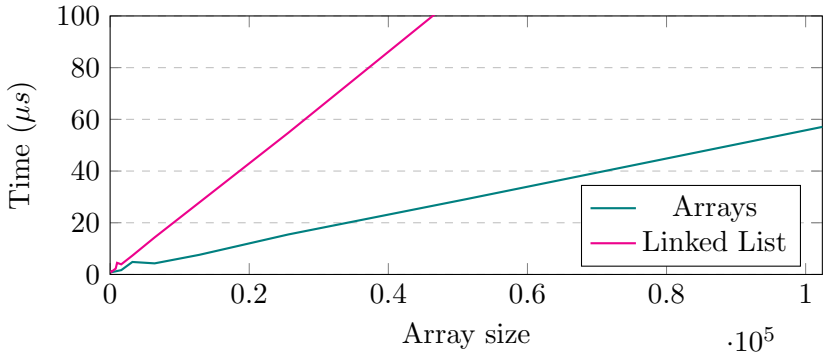
Figure 4. Appending two arrays



As we can see in figure 4, the running time for both cases are quite the same and the reason is the logic behind the implementation. In order to append an array to another, we need to allocate a large array and copy the values from both arrays, i.e. it doesn't matter which of the growing or fixed arrays is appended to the other one. In both cases we do the same operations on the same size.

As the final task in this assignment, we have generalised the benchmark and in addition to measure the time for appending two lists/arrays, we have also included the execution time for creating dynamic and fixed arrays/linked lists. The results of benchmarking are displayed in figure 5.

Figure 5. Creating and appending a two arrays/linked lists



According to the results shown in figure 5, we observe that, in general, creating and appending two arrays is faster than performing equivalent operations on linked lists. It is also noteworthy that the results for linked list in figure 5, corresponds to appending a dynamic linked list to a fixed-size list. Interestingly, the total execution time for appending a fixed list to a dynamic list was approximately twice as slow as the previous case.