Linked list

Laasyasree Desu

Fall 2023

1 Introduction

This assignment involved implementing a linked list and different methods to apply to it such as adding an element, removing, finding the length of the list and finding a specific element in the list. Additionally another method "append" was implemented where two linked lists were put together. Furthermore, the linked list was then compared and bench-marked to an array with the same functions. Lastly the linked list was implemented using a stack.

2 Linked list

A linked list, similar to an array is a data structure that stores elements. But where arrays work through indexes, linked lists work by references instead. A linked list is a sequence of cells with elements and pointers where the first cell is tracked, and all other cells point to consequent element. The last cells pointer is a null pointer as there is no other element after. The code for initiation the creating a linked list was partially given. The additional parts of code to create the list was done through a for loop that would run n number of times (n being the length of the list passed as a parameter). For every iteration the current elements pointer was pointed to the next element added and the current element was updated. An example of the code can be seen below:

2.1 Methods

After creating the linked list, different methods of altering and using the list were also implemented. The methods included adding an element/cell, deletion, finding the length of the list and finding a specific element.

2.1.1 Adding

This method involved adding an element (cell) to the beginning of the list. This was done by setting the pointer of the new cell to point to the first element of the list and then updating so that the new cell is now first.

2.1.2 Deleting

For the deleting method, a for loop was created to iterate through the list. Using an if statement, if the element being removed is present in the list then the previous cell would be accessed to be able to update the pointer to point to the cell after it (skipping the cell being removed).

2.1.3 Finding

Similar to the deleting method a for loop was created to iterate through the list and an if statement would match the element in question to the current element for every loop. However this time if the element was found the method returned true. Otherwise at the end of the loop it returned false.

2.1.4 Length

To find the length of the linked list a for loop was created to go through the list and a counter was incremented for every iteration. At the end of the loop the counter would be returned as the length of the list.

2.2 Append

An additional method implemented was the method append, where two linked lists were joined to create a new list. This method was then bench-marked twice to measure the time taken - the first time to join a list of varying sizes to a fixed size one and the second time to for the opposite. This was implemented by first constructing the method with a parameter (for the second linked list). A while loop would then iterate through the first linked list to reach the end (null pointer) and would instead point it to the first of the second linked list. An example of this code can be seen below:

2.2.1 Results

Fixed list size	Variable list	Time taken (ns)
1000	100	1100
1000	200	1100
1000	400	1100
1000	800	1100
1000	1600	1100

Table 1: Average time taken to append a linked list of different sizes to a fixed list

Looking at the table above it can be seen that for all linked list sizes the time taken remained the same. Due to the constant time, it can be concluded

that the complexity is O(1), which implies that the time amounts to the same regardless of how big "n" is. This can also be seen in the code for the algorithm. To be append the lists, only the first is traversed fully and the null pointer at the end is updated, hence the time being independent of the size of the second list.

Fixed list size	Variable list	Time taken (ns)
1000	100	100
1000	200	200
1000	400	500
1000	800	1300
1000	1600	2300

Table 2: Average time taken to append a linked list of fixed size to list of various sizes

From observing the results above it can be seen that the time taken to append the two lists changed with the size of the list. A linear relationship can be observed, leading to the conclusion that the time complexity is O(n). This emphasises that the run-time depends on the size "n" (in this case the linked list). This was also seen in the implementation, as the whole of the first linked list would be traversed and the sizes of the first list were varied.

2.3 Comparison

The second task of this assignment was to compare the time efficiency of a linked list with an array. This was done in two parts, the first comparison was to see the time taken to build a linked versus an array. The second comparison was to implement the append method on both data structures and measure the time.

2.3.1 Implementing array

The append method for two arrays was first implemented by taking the sum of the two arrays in question. This sum was then used to allocate space for the new array, the length of the new array was equal to the sum. The two arrays were then joined by copying all elements onto the new bigger array.

Array size	Time taken (ns)
100	60
200	100
400	300
800	500
1600	800

Table 3: Average time taken to append two arrays

looking at the results above the append method implemented on two arrays, one of fixed size and one of variable has a linear relationship, as the time taken is growing relative to the arrays size. This is also seen in the method as the the size of the new array would be affected by the size of the individual arrays. This gives the linear time complexity O(n). Although the linked list and the array both have linear time complexities, the results show the array to be slightly more efficient. However, smaller fixed sizes of linked lists appended to any other sized ones could be the most efficient as the only the smaller list would be traversed, while the second linked list could be of any size.

2.4 Stack

In a previous assignment a stack was implemented using arrays. The last task in this assignment also involved implementing the stack, however this time using linked lists to be able to compare them. The biggest difference between the two implementations would be in memory allocation. Linked lists would only use the memory that is needed, leading to no wastage. A dynamic stack on the other hand, depending on the conditions for resizing, would have wasted memory until enough elements are pushed onto it to warrant a resizing. Furthermore to be able to resize, the elements in the array would all have to be copied onto a bigger or smaller array. However this only happens for there is a need to change the size, while in linked lists, to be able to push and pop the whole linked list would need to traversed, affecting the run-time of the operations. Therefore the different implementations would be advantageous in different circumstances, particularly to with memory or run-time.