Different data structures

Leo Halfar

September 2023

Introduction

This report will discuss the implementation of a linked list as well as appending linked lists and arrays and the difference in time consumption. Furthermore it will discuss the cost when creating a linked list compared to allocating space for an array.

A linked list

The linked list included a sequence of cells. Each cell had a value as well as a reference to the next cell in the sequence. In order to get information from the list four methods had to be created, add, length, remove and find. A cell first was created in order to always have access to the first cell in the list.

Add

The add function, as the name would suggest added a cell to the sequence. The function was called with an integer that was to be added to the list. First a cell current was created and set to the first cell. An if statement then checked if the current cell was empty meaning the list was empty. If this was the case a new cell was created with the integer as its value and the next reference null. The first cell was subsequently set to this new cell.

If this wasn't the case the current cell was set to the next cell in the list using current.next. This was done until the cell after current was null using a while loop. A new cell with the integer add was called with as its value was then created and the current cell's next reference was set to the new cell.

Remove

The function remove was to remove a cell in the sequence. First of a cell current was set to the first cell. The function then checked if the list was empty or if it only contained one cell. If it was empty the function simply

returned since there was nothing to return. If the list contained a single cell the first cell was simply set to a null cell. In the case that neither of these if statement where true the current cell was set to the current.next cell until the value of current.next was equal to the value that was set to be removed. When this occurred the cell current.next was set to the current.next.next cell.

Length

The length function returned the length of the list in the form of an integer. This was simply done by setting a cell current to the first cell and creating a counter i. A while loop then incremented i as long as the current cell wasn't null. Each time i was incremented current was set to current.next. When the while loop finished the function returned i.

Find

The find method checked if a value could be found within the list and returned a boolean true or false depending on it's finding. This was implemented by first setting a cell current to the first cell and creating a boolean trueorfalse and setting it to false. With a while loop the function checked if the current cells value was equal to the value that was being searched for. If that was the case trueorfalse was set to true and a break was used to exit the while loop and return the boolean. If not the current cell was set to current.next and that cells value was checked. This was done until the value was found or the current value was equal to null meaning we had reached the end of the sequence at which point the while loop would be exited and false would be returned.

Append

The append function appended one sequence b to the end of another sequence a. This was achieved by using a while loop to get to the last cell of list a and setting the cells next reference to the first cell in list b. A function that could append an array b to the end of an array a was also created. This was implemented by creating a new array with the size of the two arrays combined. Two for loops then added the elements from the two original arrays into the new array.

Push and Pop

The next assignment was to implement a stack data structure with the push and pop method by changing our implementation of the remove function. The add function already worked like a push method and therefor didn't

have to undergo any changes. The remove function was changed to always remove the last cell in the sequence. This was done by first setting a cell current to the first cell. There after using if statements the function checked if the list was empty or if it only contained one cell. If it was empty the function returned and if it only contained one cell the first cell was returned and subsequently set to a null cell. If neither of these statements where true a while loop set the current cell to current.next until current.next.next was equal to null. This meant that the current cell was the second to last cell in the sequence. The value of the current.next cell was returned and the current.next cell was set to current.next.next meaning the last cell in the sequence was returned and removed from the stack.

Benchmarks

The benchmarks for appending arrays and sequences was done by first creating arrays of different sizes using a function created in a previous assignment and measuring the time it took to append two arrays using the system.nanotime methods. Using the provided method for creating a linked list, sequences of different sizes was created and the time consumption when appending a sequence on to another one was measured. The benchmarks for measuring the time consumption when allocating space to an array or creating a linked list was achieved by simply allocating space for arrays of different sizes and creating linked lists of varying sizes and measuring the time consumption.

Result

A	**B**	**Linkedlist**	**Array**
size	size		
1000	1000	11	37
1000	10000	11	190
1000	100000	11	1700
10000	1000	110	190
100000	1000	920	1700

Table 1: Time consumption when appending two linked lists and arrays in microseconds.

Size	Array	Linkedlist
1000	1700	38000
10000	15000	430000

Table 2: Time consumption when allocating memory for an array and creating a linked list in nanoseconds.

When looking at the results for appending a sequence b onto a sequence a we can see that when b is constant we get a time complexity of O(n) which is to be expected. When a is constant we get a constant result which also was expected. Since we only have to move through list a and set it's last cell to the first cell in list b changing the list b when a is constant shouldn't affect the execution time. The execution time of appending an array onto another array will always depend on the total amount of elements in the arrays since all of them have to be put in to the new, larger array. Because of this appending list a and b will always be faster than appending array c and d as long as b is smaller than c and d combined. As we can see in the result the the time consumption for creating a linked list was much greater than allocating space for an array.

Discussion

One thing that was not measured during this experiment was the difference in execution time between an array and linked list implementation for a stack data structure. Here there is a couple different elements to take into account. The push operation will be faster in the array based stack since it doesn't have to go through all the elements as long as the size of the stack doesn't have to change. The same goes for the pop operation, since in order to remove the last element in the linked list based stack we have to go through each cell until we reach the last where the array based stack simply removes the last element. All this should indicate that a linked list based stack should be faster for a dynamic stack and a array based stack should be faster for a static stack. When analyzing the results for the append operation when the b list was static one can see that the results doesn't entirely coincide with the time complexity O(n). Could we have gotten a more accurate result by doing more measurements for each list size or perhaps including more list sizes. Perhaps, however since the content of the list shouldn't affect the result in any way the probability of getting a vastly different result is slim to none.

conclusion

In conclusion, this experiment discusses the performance characteristics of linked lists and arrays in various scenarios. Linked lists excel in dynamic data structures, while arrays are more suitable for static data structures.