Linked Lists

Fatima Mohammad Ali

Fall 23/09/2023

Introduction

The purpose of this assignment was to explore a *singly linked list* by implementing different methods, such as *append*, and comparing it to array structure at last.

Linked List

A singly linked list is a list of *cells* which each holds a single reference to the next cell in the sequence. The list will solely have access to the first cell to navigate through the other cells.

To implement a list in this task, a class `LinkedList` with a private class `Cell` was initialized. Each cell was given an integer value (`int head`) and a pointer (`Cell tail`) to the next cell while the first cell (`Cell first`) was initialized in the `LinkedList` class. Several methods were added to build the list properties.

An `add(int item)` method which adds an integer as the first cell of the list was implemented. The method worked by initializing a cell object holding the given item value and its tail being null. If the list was empty, the new cell was set as `first`. If the list already had items (checks if `first` was not null), then the new cell would point to whatever first was pointing at (to not lose the link) and `first` would point to the new cell. Thus, a new cell was added to the beginning of the list.

To practise iterating through the list, `find(int item)` and `length()` methods were implemented. The main approach of respective method was to to start at the first cell, constantly check if the end of the list was reached (if the last pointer is a *null-pointer*, e.g. while(nxt != null)), and traverse to the next cell by using the `tail` reference. Each method was then enhanced by specific conditions and return values to carry out their respective function.

Similarly to the `add` method, a method to `remove(int item)` was implemented. When the relevant cell to be removed was found, it was removed/unlinked from the list by making the *previous* cell (initialized `Cell prev = null`) point to the *next* cell (skip over the adjacent cell). Thus, the

link to the previous and next cells were not lost and the cell in question was unlinked. Both previous and next cells were incremented accordingly with `prev` always being one step behind `nxt`, and the code was finalized by some relevant details.

That the methods all worked as they should was constantly double-checked by printing the list while coding.

Append a Singly Linked List

The main purpose of this assignment was to implement an *append* method. Appending a list to another works by iterating through the first list, find its null-pointer (last cell in the sequence), and make its tail point to the first cell of the second list. Thus, the first list would contain all cells and the second list would be empty (by setting it to null) at last.

Method

For this task, the `append(LinkedList list)` method was implemented as follows:

```
public void append(LinkedList list) {
    Cell nxt = this.first;  //start at the first cell
        //traverse the list to find the last cell
        while (nxt.tail != null) {
                nxt = nxt.tail;
        }
        //append list by setting its first ref. point to the last cell
        nxt.tail = list.first;
        list.first = null; //empty list
}
```

List *(a)* was appended to list *(b)* when called with `b.append(a)` in this assignment, meaning (b) would hold all cells while (a) would be empty afterwards.

Each list was made using a `makeList(int n)` method where n was the number of cells in a list (length) and each cell was given a sequential integer value.

Results

Benchmark results are given in table 1.

n (varied size)	a varying, b fixed	a fixed, b varying
200	4.4	2.3
400	3.4	1.6
800	5.0	2.7
1600	3.4	7.1
3200	3.4	11
6400	3.4	18
12800	3.4	37
25600	3.4	72

Table 1: Timestamps are shown in microseconds where list ***(a)*** was appended to list ***(b)*** in both cases. Fixed size was 1000 in both cases. Each timestamp is the minimum value of 100 tries (k=100).

Discussion

In this task, list (a) was appended to list (b). When (b) has a fixed size, a list of 1000 cells has to be run through to find the null-pointer where (a) should be appended. This time complexity is given by O(n) where n is the number of cells in list (b). When the last cell is found, it takes one operation to append *(a)*, regardless of the size of *(a)*. Hence, this operation is done in O(1). This could be confirmed by analysing table 1. When (b) is fixed and (a) is varying, all timestamps are approximately 3.4. This shows that regardless of the size of (a), O(n) would always be proportional to the size of (b), which is always set to 1000.

On the contrary, if (b) varied while (a) had a fixed size, there would be a linear increase of O(n) proportional to the growing size of (b). This could be confirmed by the values in table 1 where the timestamps in the last column increased with the increase of size (b).

Append Array vs Linked list

Since linked lists has similar purpose as arrays, it would be interesting to compare those structures by analysing their appending and allocation time.

Method

Appending two arrays consisted of allocating a new array and then copying all elements from both arrays to the new one. Both arrays were made by a `makeArray(int n)` method and the size of the new array was the sum of both arrays. To compare allocation time of both structures, the `makeList` and `makeArray` methods were set against each other for each n.

Results

In both cases, a was appended to b (e.g. `b.append(a)`).

n	LL b fixed	LL b varied	Arr b fixed	Arr b varied
400	3.9	1.6	0.7	0.5
800	3.9	2.7	0.9	0.8
1600	3.6	5.1	1.2	1.5
3200	4.1	9.4	1.8	2.7
6400	4.4	20	15	17
12800	3.5	36	7.0	11
25600	3.8	72	11	22

Table 2: Timestamps are shown in microseconds where ***(a)*** is appended to ***(b)*** in all cases. Fixed size was 1000 in all cases. Each timestamp is the minimum value of 1000 tries (k=1000).

n	make List	make Array
400	4.4	0.2
800	1.8	0.5
1600	19	2.6
3200	6.8	2.0
6400	14	3.9
12800	29	8.0
25600	62	17
51200	125	34

Table 3: The table shows the time it takes (in microseconds) to make a list and an array of size n. The minimum values of 1000 tries (k=1000) are given.

Discussion

To append an array to another one would require allocating a new bigger array and copying all elements to it. This would have time complexity $O(m + n)$ where m and n are the sizes of respective array, regardless of which array was appended. In table 2, it can be seen that appending an array with (b) fixed takes in general the same time as when (b) is varied (with insignificant deviations), which shows big-O notation $O(m + n)$.

Appending a linked list when (b) was varied (column 3 in table 2) required more time than appending arrays. This could be due the fact that

traversing (b) through each cell's reference to find the last cell requires more time than copying elements to a new array.

Furthermore, table 3 shows that it takes more time to allocate a linked list than an array of the same size. Allocating an array is done at compile-time and each element is accessed directly through its index (each index access is done in O(1)). A list however, has its cells linked to each other through their pointers. The pointers have to be updated during run-time by accessing a cell's address in memory and use its reference to link the next cell and create a sequence etc. This would be more time consuming than allocating an array.