CS 225 - Lecture 6

Scribe : Harsha Srimath Tirumala

1 Learning Goals

↪ Design choices for Data variables of array lists

↪ Review of array list implementations

↪ Amortized analysis

↪ Resize strategies for array lists at capacity : Resize +2

2 Linked Lists - Efficient *modify*, Inefficient *find*

Linked lists support efficient modifications of data when given access to the relevant location(s). However, linked lists are inefficient for operations like random access which require finding.

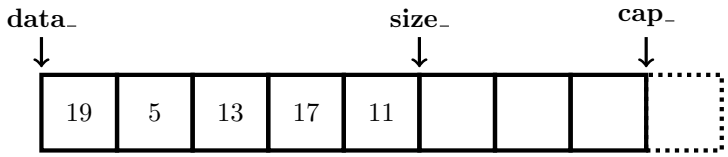
3 Array Lists

In array lists, an array is allocated as continuous memory. There are three data variables for efficient array usage:

↪ Data : Start position of an array (pointer to array start)

↪ Size : Current number of items in array (pointer to next avaialble space)

↪ Capacity : Total number of allocated spaces (pointer past end of array)



↪ **Pointer Arithmetic** - The design choice allows us to use pointer arithmetic directly on the variables, such as:

$$\#\text{ items in array} = \textbf{size}\_ - \textbf{data}\_ = 5$$

4 Array list Operations

↪ **Access random(*index*)** : $O(1)$ time as item can be directly accessed at $\textbf{data}\_ + index$

↪ **insertAtFront(*data*)** : $O(n)$ because all subsequent elements must be moved to the right.

↪ **insertAtBack(*data*)** : $O(1)$ when array is not full but $O(n)$ when array is at capacity as all elements need to be copied to a new, larger array list.

↪ **insert(*data, index*)** : $O(n)$ in the worst case as all subsequent data must be shifted.

Table 1: Array list (not at capacity)

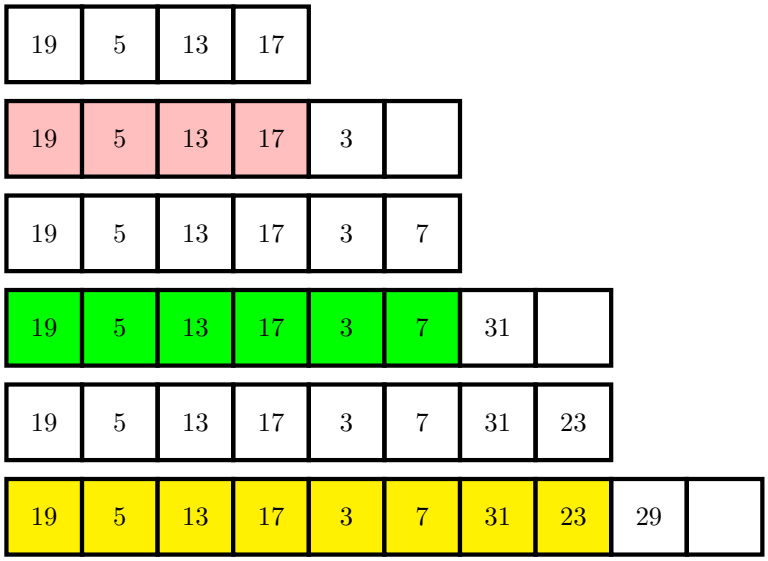
	@Front	@Back	@Index
Insert	$O(n)$	$O(1)$	$O(n)$
Delete	$O(n)$	$O(1)$	$O(n)$

5 Array list (at Capacity)

Since we don't own space past capacity, any subsequent addition requires allocation of new memory.

5.1 Resize : +2 elements

Consider the strategy of adding 2 elements when the array list is at capacity (as illustrated below). Note that using this strategy, the amount of work done for the first insert at capacity is $O(size\_)$ - as all the elements must be copied to the newly allocated array list. However, the next insert can be done in $O(1)$ - as the list is not at capacity anymore.



(In the above illustration, colored lists are newly allocated and colored nodes have been copied)
Let $T(n)$ denote the time taken to populate an array list of size n. It is easy to see that :

$$T(n) = T(n-2) + (n-2) + 1 + 1 = T(n-2) + n$$

This is because when the list was at capacity $(n-2)$, we had to copy $(n-2)$ elements into the new n sized array list and then insert elements $(n-1)$ and n. Solving for $T(n)$:

$$T(n) = \sum_{i=0}^{\frac{n}{2}} (n-2i) = \sum_{i=0}^{\frac{n}{2}} n - \sum_{i=0}^{\frac{n}{2}} 2i = \left(\frac{n}{2}+1\right) n - 2\frac{\left(\frac{n}{2}\right)\left(\frac{n}{2}+1\right)}{2} = \frac{n^2}{4} + \frac{n}{2} = \Theta(n^2)$$

This shows us that inserting n elements into an array list following the "Resize +2" strategy requires $\Theta(n^2)$ work. Observe that the worst case complexity of a particular insert is $\Theta(n)$ - which happens when an insert is called with the array list at capacity $\Theta(n)$. The *amortized* cost of each insert is $\frac{\frac{n^2}{4}+\frac{n}{2}}{n} = \frac{n}{4} + \frac{1}{2} = \Theta(n)$.
In the next lecture, we will discuss a more efficient resize strategy.