CS 225 - Lecture 7

Scribe : Harsha Srimath Tirumala

1 Learning Goals

↪ Array List : Resize x2, Amortized Analysis

↪ Data structure tradeoffs : Extensions to lists

↪ Stack : LIFO data structure

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

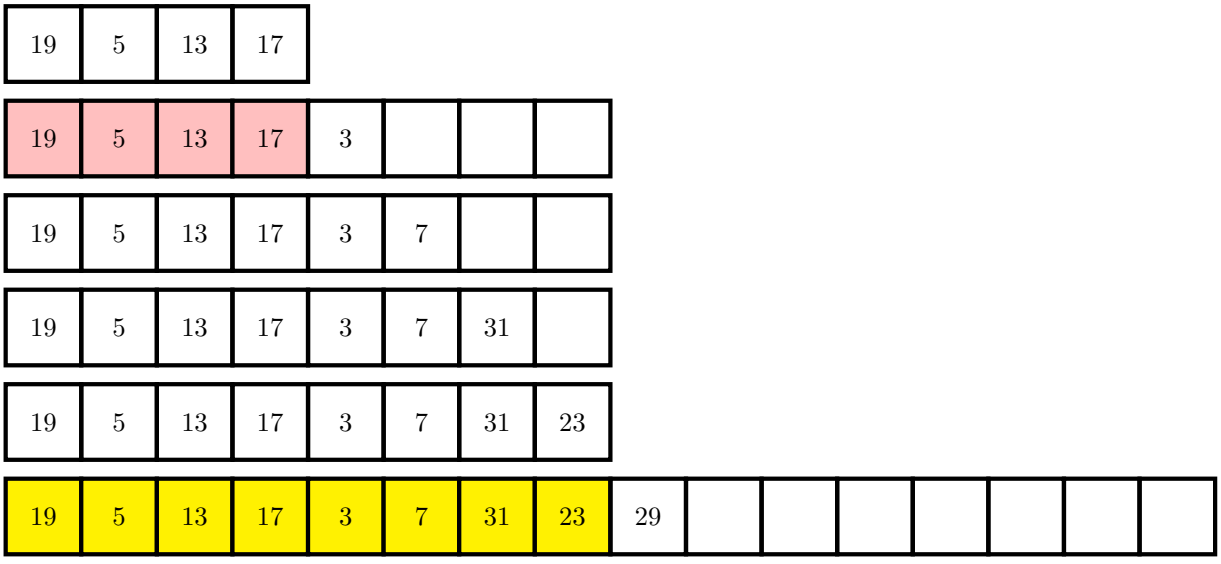
Array lists support efficient search of data when given access to the relevant location(s). However, array lists are inefficient for operations like modify which involve shifting subsequent elements.

3 Array Lists (at capacity)

In the previous lecture we discussed the resize +2 strategy - which incurred $\Theta(n)$ work per insert. We will now discuss a different strategy.

3.1 Resize x2

Consider the strategy of doubling the size of the array list when an element has to be inserted with the array at capacity. This can be implemented by allocating a new block of memory for a list of size double the current capacity and then copying all elements onto the new array list.
Illustrated below is how an array list of size 4 (at capacity) supports a new insertion of element 3 followed by insertion of 7, 31, 23 and 29.



Let $T(n)$ denote the time taken to populate an array list of size n. It is easy to see that :

$$T(n) = T(n/2) + (n/2) + (1 \times (n/2)) = T(n/2) + n$$

This is because it took $T(n/2)$ to populate the array list to $(n/2)$ elements; we then required copying $(n/2)$ elements into a new n sized array list before inserting element $((n/2)+1)$. All subsequent inserts thru element n only need $O(1)$ time each as the list is not at capacity.
Solving for $T(n)$:

$$T(n) = T(n/2) + n = \sum_{i=0}^{i=\lceil \log_2 n \rceil} \frac{n}{2^i} = n\left(\frac{1 - (\frac{1}{2})^{\log_2 n+1}}{1 - \frac{1}{2}}\right) = 2n\left(1 - \frac{1}{2n}\right) = 2n - 1$$

where $T(n)$ is the sum of a geometric series with first term n and common ration $\frac{1}{2}$. The total number of terms is $\log_2 n + 1$ as we need to double our list $\log_2 n$ times to get to capacity n (we have made the simplifying assumption that n is a power of 2).

This shows us that inserting n elements into an array list using the resize x2 strategy requires $\Theta(n)$ work. The worst case cost of insert is still $\Theta(n)$ - as evidenced by the insertion of the $\left(\frac{n}{2}+1\right)$-th element which requires copying the first $\frac{n}{2}$ elements to the newly allocated array list. The *amortized* cost of insert is now $\frac{2n-1}{n} \approx 2$ - which is the big saving this strategy gives us.

Table 1: Resize +2 vs Resize x2

	Worst-case	Amortized
Resize +2	$O(n)$	$\Theta(n)$
Resize x2	$O(n)$	$\Theta(1)$

Note - We have calculated the precise amortized cost (Θ bound) of insert - not just an upper bound.

4 Linked list vs Array list

Table 2: List Implementations

	Linked List	Array List
Random Access	$O(n)$	$O(1)$
Insert after *given* element	$O(1)$	$O(n)$
Delete after *given* element	$O(1)$	$O(n)$
Insert at *arbitrary* location	$O(n)$	$O(n)$
Remove at *arbitrary* location	$O(n)$	$O(n)$
Search for an input value	$O(n)$	$O(n)$
Insert/Remove *front*	$O(1)$	$O(n)$
Insert/Remove *back*	$O(n)$	$O(1)^*$(Amortized)

5 Tradeoffs

We will frequently ask ourselves if there are any tradeoffs that lead to meaningful savings with respect to data structures.

5.1 Size of a Linked List

Given a linked list, we need to scan through the list from head to the null pointer to establish the size of the linked list. This requires $O(n)$ work. We can instead have an unsigned int *size* to store the size of the linked list and edit it appropriately after each operation. Depending on the context, this can be a useful tradeoff.

5.2 Sorted Linked list

A sorted linked list has limited benefits as random access is expensive. It may be of use in cases such as an in order traversal that prints elements in a sorted order.

5.3 Sorted Array list

A sorted array list is quite useful in supporting a quicker search. Binary search for sorted array list has a complexity of $O(\log_2 n)$ - as opposed to searching an unsorted array list. The downside? Insert is more expensive as it must be structured (and we can't just insert at the end).

5.4 Doubly Linked list

A linked list where each node stores two pointers - to the next as well as the previous element. This supports bi-directional scanning. Since each node stores two pointers, the storage requirement is an additional $O(n)$ pointers.

5.5 Utopia : $O(1)$ Insert, $O(1)$ remove

Is it possible to have a data structure that supports efficient insert and remove operations? Well, the ability to efficiently do random access gets in the way of this goal. Why is that? It's because if we require efficient random access (say $O(1)$ time), then we need to do some bookkeeping post every insert/remove operation to continue to support random access. The amount of work required can be significant (for example, $\Omega(n)$ in the case of array lists). We will now discuss a data structure that makes this tradeoff to support $O(1)$ Insert, $O(1)$ remove.

6 Stack

↪ A stack stores an ordered collection of items. Stacks support only two operations - *Push* and *Pop* :

- *Push* : Put an item on top of the stack.
- *Pop* : Remove the top item of the stack (and return it)

Stack has only one pointer - top_. Since elements can only be removed from the top, stacks follow the Last In First Out model. Stacks do NOT support random access.

↪ C++ has a built-in stack implemented using a vector or deque. Array list implementation supports $O(1)$ insert and $O(1)$ remove - as long as array is not at capacity.

↪ Stacks thru linked lists - Stacks can be simulated using linked lists where *push* is *InsertFront*, *Pop* is *Remove-Front* and the head pointer simulates the behavior of *top* of stack. Linked list implementation entails $O(1)$ insert and $O(1)$ remove.