Lecture 2: Data Structures and Dynamic Arrays

[Previous Lesson]

Interface (API/ADT) vs Data Structure

Interface tells you what to do
Data Structure tells you how to do it

Interface

- Specification
- What data you can store
- What operations are supported and what they mean.
- Problem

Data Structure:

- Representation
- How to Store it
- Algorithms
- Solution

Two Main Interfaces:

- Set
- Sequence

Two Main DS Approaches:

- Arrays
- Pointer based

Static Sequence Interface:

Number of items doesn't change. They are subject to these operations.

- Build(x) Make new DS for items in x
- Len( ) Return n
- Iter-seq( ) output the items in sequence order
- Get-at(i) return x, index i
- Set-at(i, x) set x to x
- Get_first/last( )
- Set_first/last( )

Solution (natural solution): Static Array:

- O(i) per get-at/set-at/len
- O(n) per build/iteration

Memory allocation Model:
Allocate an array of size in $\Theta(n)$ time
Space = O(time)

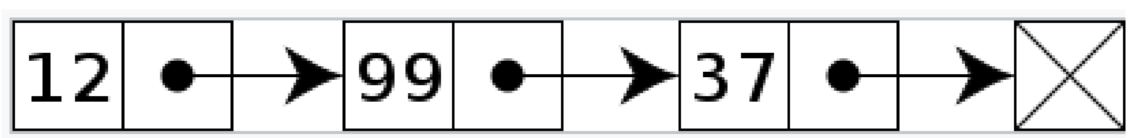
Key: Word RAM model of computation

- Memory = array of w-bit words
- ‘Array” - consecutive chunk of memory
- array[i] = memory[address(array) + i]
- Array access is constant time O(i)
  Assume $w \geq \log n$

Dynamic Sequence Interface; static sequence, plus:

- Inset_at(i, x)make x the new x_i, shifting others
- Delete_at(i) shift
- Insert/delete_first/last (x)/( )

Linked Lists:
Stored items in a bunch of nodes, each node has an item and the next field.



A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list.

Dynamic Sequence Operations
Static Array (nothing dynamic)

- Insert/delete = at( ) cost
- $\Theta(n)$ time
  1. Shifting
  2. Allocation/copying

Linked List (bad at random access, good at working at ends, good at being dynamic)

- Insert/delete_first( ): O(i)time
- get/set_at need $\Theta(i)$ time
- Worst case, $\Theta(n)$ time

Data Structure Augmentation: Adding extra information to data structure (maintenance)

Dynamic Arrays (Python "lists")

- Relax constraint size(array) = n (which is the number of items in the sequence)
- Enforce size = $\Theta(n)$ & $\geq$ n
- Maintain $A[i] = x_i$
- Insert_last(x): add to end unless n = size
- If n = size, allocate new array of 2 * size
- n insert_last( ) from empty array, resize at n = 1, 2, 4, 8, 16….

  Resize cost = $\Theta(1 + 2 + 4 + 8 + 16\ldots) + n$

Amortization:

Operation takes T(n) amortized time if any k operations take $\leq$ k * T(n) time (averaging over operation sequence)