Basic Data Structures I

- ADTs
- Lists, Stacks, Queues
- Sequential Allocation

Abstract Data Type (ADT)

- Abstracts the functional elements of a data structure (math) from its implementation (algorithm / programming)

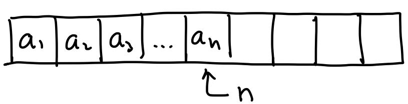
Linear List ADT:

Stores a sequence of elements $\langle a_1, a_2, \dots, a_n \rangle$. Operations:

- init() - create an empty list
- get(i) - returns a_i
- set(i, x) - sets i^{th} element to x
- insert(i, x) - inserts x prior to i^{th} (moving others back)
- delete(i) - deletes i^{th} item (moving others up)
- length() - returns num. of items

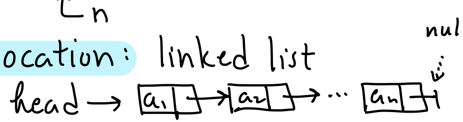
Implementations:

Sequential: Store items in an array

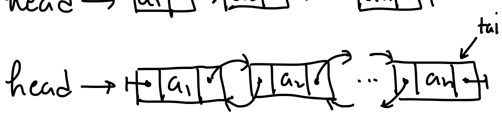


Linked allocation: linked list

Singly:



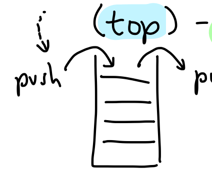
Doubly:

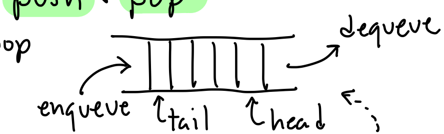


Performance varies with implementation

Stack:

All access from one side (top) - push + pop





Queue:

FIFO list: enqueue inserts at tail and dequeue deletes from head

Doubling Reallocation:

When array of size n overflows

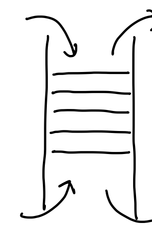
- allocate new array size 2n
- copy old to new
- remove old array

Dynamic Lists + Sequential Allocation:

What to do when your array runs out of space?

Deque ("deck"):

Can insert or delete from either end

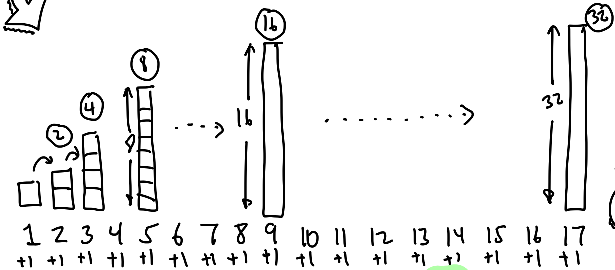


Basic Data Structures II
- Amortized analysis of dynamic stack

Cost model (Actual cost)

Cheap: No reallocation → 1 unit

Expensive: Array of size n is reallocated to size 2n } → 2n+1



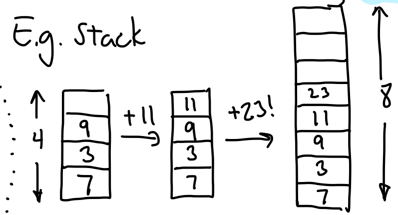
Total = 17+(2+4+8+16+32) = 79

Amortized Cost:
Starting from an empty structure, suppose that any sequence of m ops takes time T(m). The amortized cost is T(m)/m.

Thm: Starting from an empty stack, the amortized cost of our stack operations is at most 5.
[ie. any seq. of m ops has cost ≤ 5·m]

Dynamic (Sequential) Allocation
- When we overflow, double

E.g. Stack

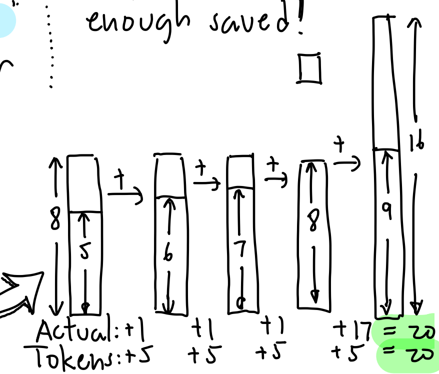


Charging Argument:
- Each request of push/pop we charge user 5 "work tokens"
- We use 1 token to pay for the operation + put other 4 in bank account.
- Will show there is enough in bank account to pay actual costs.

Proof:
- Break the full sequence after each reallocation → run

1 2 | 3 | 4 5 | 6 7 8 9 | 10 11 ... 16 17 |

- At start of a run there are n+1 items in stack and array size is 2n
- There are at least n ops before the end of run
- During this time we collect at least 5n tokens
  - → 1 for each op
  - → 4 for deposit
- Next reallocation costs 4n, but we have enough saved! □



Basic Data Structures III
- Dynamic Stack - Wrap-up
- Multilists + Sparse Matrices

Fixed Increment: Increase by a fixed constant
$n \longrightarrow n + 100$

Fixed factor: Increase by a fixed constant factor (not nec. 2)
$n \longrightarrow 5 \cdot n$

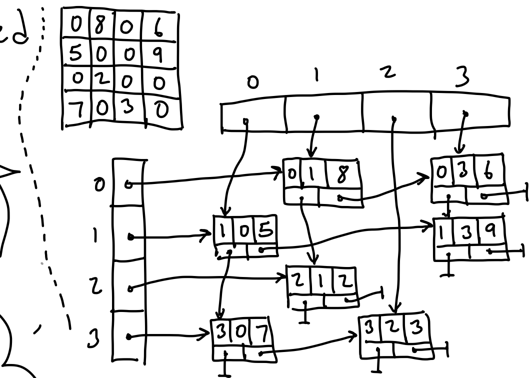
Squaring: Square the size (or some other power)
$n \rightarrow n^2$ or $n \rightarrow \lceil n^{1.5} \rceil$

Which of these provide $O(1)$ amortized cost per operation?

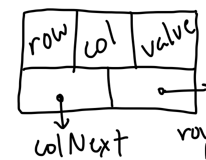
Leave as exercise ☹

(Spoiler alert!)
- Fixed increment → no
- Fixed factor → yes
- Squaring → ?? (depends on cost model)

Dynamic Stack:
- Showed doubling ⇒ Amortized $O(1)$
- Other strategies?

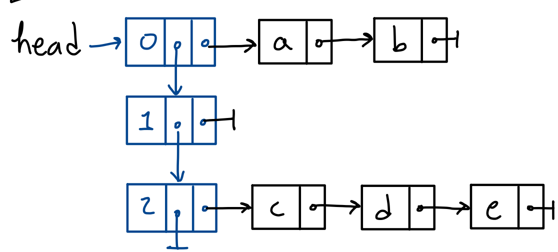


Node:

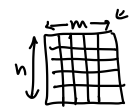


Idea: Store only nonzero entries linked by row and column

Multilists: Lists of lists



Sparse Matrices:
An $n \times m$ matrix has $n \cdot m$ entries and takes (naively) $O(n \cdot m)$ space



Sparse matrix: Most entries are zero