INTRO TO DATA STRUCTURES

Elements of Data Structures

modeling	how real-world objects are encoded
operations	allowed functions to access and modify data structures
representation	mapping to memory
algorithms	how operations are performed

↳ this course covers both theoretical and practical approaches

- ↳ theoretical: algorithms and asymptotic analysis
- ↳ practical: implementation and efficiency in practice

Basic Data Structures

abstract data type	abstracts functional elements of structure from implementation

Linear Lists

↳ stores a sequence of elements $[a_1, a_2, \dots, a_n]$

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} element
- **delete(i)**: deletes i^{th} element
- **length()**: returns number of items

Implementations

sequential:

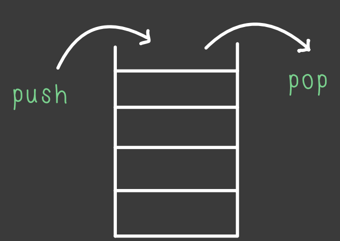
store items in array

linked allocation:

items stored w/ pointers

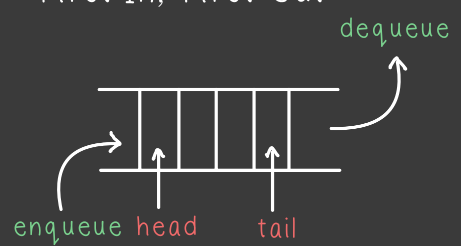
Stack

Last In, First Out



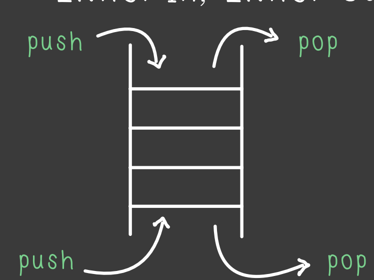
Queue

First In, First Out



Deque

Either In, Either Out



Dynamic Stacks

↳ when the array runs out of space:

  ↳ double reallocation: when an array of size n overflows...

    ↳ allocate a new array of size $2n$

    ↳ copy old array to new array

    ↳ remove old array

amortized cost	starting with an empty structure and assuming any sequence of m operations takes time $T(m)$, amortized cost is $T(m) / m$

Amortized Cost Analysis

Theorem

starting from an empty stack, the amortized cost of our stack operations is at most 5

Charging Argument

- for each request of push/pop we charge the user 5 work tokens
- we use 1 token to pay for the operation and put the other 4 in a bank account
- want to show that there is enough in the bank account to pay actual costs

Proof

- break the full sequence after each reallocation (run)
- at the start of a run there are $n + 1$ items in the stack and the array is size $2n$
- there are at least n operations before the end of the run
- during this time we collect at least $5n$ tokens
- next reallocation costs $4n$, but we have enough saved

↳ other reallocation strategies:

  ↳ fixed increment reallocation: allocate a new array of size $n + c$

  allocated stack size: $[c, 2c, 3c, \ldots, kc]$

  ↳ fixed factor reallocation: allocate a new array of size $c * n$

  allocated stack size: $[1, c, c^2, c^3, \ldots, c^k]$

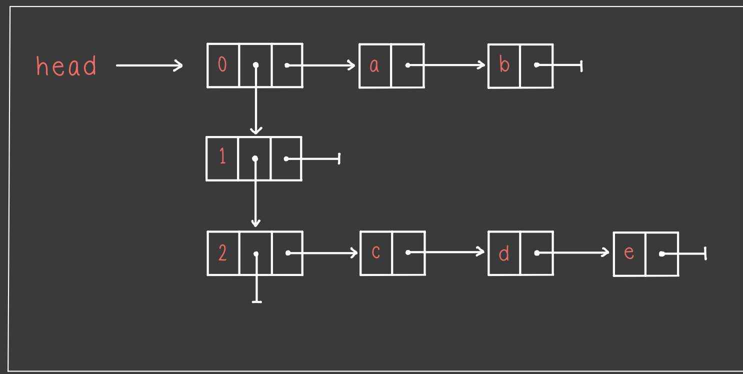
  ↳ exponential reallocation: allocate a new array of size n^c

  allocated stack size: $[1, c, c^{c^1}, c^{c^2}, \ldots, c^{c^k}]$

Multilists

multilist | a list of lists

- ↳ common example is Java's ArrayList
- ↳ more interesting example is Sparse Matrix



Sparse Matrix

- ↳ create 2n linked lists (one for each row and column)
- ↳ each entry of each list stores 5 values:
  - ↳ row index
  - ↳ column index
  - ↳ value
  - ↳ pointer to next row
  - ↳ pointer to next column

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

represent with nodes like: