Linear List ADT:
Stores a sequence of elements \( \langle a_1, a_2, \ldots, a_n \rangle \). Operations:
- \( \text{init}() \): create an empty list
- \( \text{get}(i) \): returns \( a_i \)
- \( \text{set}(i, x) \): sets \( i \)th element to \( x \)
- \( \text{insert}(i, x) \): inserts \( x \) prior to \( i \)th
  (moving others back)
- \( \text{delete}(i) \): deletes \( i \)th item
  (moving others up)
- \( \text{length}() \): returns num. of items

Implementations:
- Sequential: Store items in an array
- Linked allocation: linked list
  - Singly: \( \text{head} \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \text{null} \)
  - Doubly: \( \text{head} \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \text{null} \)

Performance varies with implementation

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

Doubling Re-allocation:
When array of size \( n \) overflows
- allocate new array size \( 2n \)
- copy old to new
- remove old array

Basic Data Structures I
- ADTs
- Lists, Stacks, Queues
- Sequential Allocation

Dynamic Lists + Sequential Allocation: What to do when your array runs out of space?
- Deque ("deck"): Can insert or delete from either end

Stack: All access from one side
- \( \text{push} \) + \( \text{pop} \) (LIFO)
- \( \text{null} \)

Queue: FIFO list: \( \text{enqueue} \) inserts at tail and \( \text{dequeue} \) deletes from head
Cost model (Actual cost)
Cheap: No reallocation → 1 unit
Expensive: Array of size \( n \) is reallocated to size \( 2n \)

Dynamic (Sequential) Allocation
- When we overflow, double
  Eg. Stack
  \[
  \begin{array}{c}
  \text{Top} \\
  \text{9} \xrightarrow{+11} \text{3} \\
  \text{7} \xrightarrow{+23!} \text{11} \\
  \text{23} \\
  \end{array}
  \]

Basic Data Structures II
- Amortized analysis of dynamic stack

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. [i.e. any seq. of \( m \) ops has cost \( \leq 5m \)]

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.
Fixed Increment: Increase by a fixed constant
\( n \rightarrow n + 100 \)

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

Squared: Square the size (or some other power)
\( n \rightarrow n^2 \) or \( n \rightarrow n^{1.57} \)

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

Dynamic Stack:
- Showed doubling \( \Rightarrow \) Amortized \( \mathcal{O}(1) \)
- Other strategies?

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

Multilists: Lists of lists

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

Sparse matrix: Most entries are zero
Announcements: 9/1

Prog Assign 0 - Almost done
→ Rules - Can't use ArrayList
→ Challenge → 1 loop → Realloc
↓
Extra credit → No loops