Linear List ADT:
Stores a sequence of elements \(a_1, a_2, \ldots, a_n\). 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: head -> \(a_1\rightarrow a_2 \rightarrow \ldots \rightarrow a_n\)
  - Doubly: head -> [\(a_1\rightarrow a_2 \leftarrow a_3 \rightarrow \ldots \rightarrow a_n\)]

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

Doubling Reallocation:
- 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

Performance varies with implementation
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
  Basic Data Structures II
- Amortized analysis of dynamic stack

Proof:
- Break the full sequence after each reallocation → run
  \[ 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \quad 13 \quad 14 \quad 15 \quad 16 \quad 17 \]
- 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!

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 \( \leq 5 \cdot m \)]

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)
\[ n \rightarrow 5 \cdot n \]
**Squaring:** Square the size (or some other power)
\[ n \rightarrow n^2 \text{ or } n^{\ln 100} \]

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

Leave as exercise (spoiler alert!)
- Fixed increment \( \rightarrow \) no
- Fixed factor \( \rightarrow \) yes
- Squaring \( \rightarrow \) yes

**Dynamic Stack:**
- Showed doubling \( \Rightarrow \) Amortized \( O(1) \)
- 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) \( O(n \cdot m) \) space

Sparse matrix: Most entries are zero