Linear List ADT:
Stores a sequence of elements \( <a_1, a_2, ..., a_n> \). 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 (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: `head`\( \rightarrow a_1 \rightarrow a_2 \rightarrow ... \rightarrow a_\text{null} \)
  - Doubly: `head`\( \rightarrow a_1 \rightarrow a_2 \rightarrow ... \rightarrow a_n \rightarrow \text{null} \)

Performance varies with implementation

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

Doubling 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 I
- ADTs
  - Lists, Stacks, Queues
- Sequential Allocation

Stack: All access from one side
- `top` - push + pop
- LIFO
  - push
  - pop

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

Donald Knuth
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
  - Example: Stack

\[
\begin{array}{c|c|c}
\text{Seq} & \text{Top} & \text{Size} \\
\hline
9 & 11 & 3 \\
7 & 11 & 3 \\
7 & 1 & 3 \\
\end{array}
\]

Proof:
- Break the full sequence after each reallocation → run
  - 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 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 \( 3n \) tokens
  - 1 for each op
  - 4 for deposit
- Next reallocation costs \( 4n \), but we have enough saved!

Basic Data Structures II
- Amortized analysis of dynamic 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.

Amortized Cost: Starting from an empty structure, suppose that any sequence of \( m \) ops takes time \( T(m) \).
- The amortized cost is \( T(m)/m \).
- \( t_1 + t_2 + \cdots + t_m = T(m)/m \)

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 \( \leq 5m \)]
Announcements: Tue 1/31
- Programming Assignment 0 out
  (see handouts page) → Due Feb 8, 11:59 pm
- Java Eclipse set up?
  - see Project page
- Office hours
  - soon