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
  \[
  a_1, a_2, \ldots, a_n
  \]
- Linked allocation: linked list
  - Singly: head → \[ a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n \] → tail
  - Doubly: head → \[ a_1 \rightarrow a_2 \rightarrow \cdots \rightarrow a_n \] → tail

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
- \text{top} - push + pop

Queue: FIFO list: enqueue inserts at tail and dequeue deletes from head
- Double-ended queue
Cost model (Actual cost) 
Cheap: No reallocation → 1 unit
Expensive: Array of size \( n \) \( \geq 2n+1 \) is reallocated to size \( 2n \)

Dynamic (Sequential) Allocation
- When we overflow double
- Example: Stack

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c} \hline
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
| \hline
\end{array}
\]

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

\[
\frac{79}{17} \approx 4...
\]

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 \( \frac{T(m)}{m} \).

Thm: Starting from an empty stack, the amortized cost of our stack operations is at most \( \frac{5}{2} \)

[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.

Proof:
- Break the full sequence after each reallocation → run
\[
12345,67891011121314151617
\]
- 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 \( > 3.5n \) tokens
- Next reallocation costs \( 4n \), but we have enough saved!

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c|c|c} \hline
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
| \hline
\end{array}
\]

Actual Tokens:
\[
\begin{array}{c|c|c|c|c} \hline
+1 & +1 & +1 & +1 \\
\hline
+5 & +5 & +5 & +5 & = 20 \end{array}
\]
**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)
- \( n \rightarrow n^2 \) or \( n \rightarrow n^{1.57} \)

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)

\( 4 \rightarrow 16 \rightarrow 256 \rightarrow \ldots \)

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

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

**Node**:
- Idea: Store only non-zero entries linked by row and column

**Multilists**: Lists of lists

**Sparse Matrices**:
- An \( nxm \) matrix has \( n \cdot m \) entries and takes \( \text{(naively)} O(n \cdot m) \) space

**Sparse matrix**: Most entries are zero