Linear List ADT:
Stores a sequence of elements \( \langle a_1, a_2, \ldots, a_n \rangle \). 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
  - Store items in an array
  - \( a_1, a_2, \ldots, a_n \)
- **Linked allocation**: linked list
  - **Singly**: \( \text{head} \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \text{null} \)
  - **Doubly**: \( \text{head} \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow \text{tail} \)

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
  - \( \text{push} + \text{pop} \)
  - \( \text{LIFO} \)
  - `push`
  - `pop`
  - `head`

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

Dynamic (Sequential) Allocation
- When we overflow, double
  - $2n \times m$

Proof:
- Break the full sequence after each reallocation $\rightarrow$ run
- 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 $\frac{3}{2}n$ 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

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 $\leq 5 \cdot m$]
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 \( n \rightarrow n^{1.5} \)

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

Leave as exercise (Spoiler alert!)
- Fixed increment \( \rightarrow \) no
- Fixed factor \( \rightarrow \) yes
- Squaring \( \rightarrow \) ?? (depends on cost model)

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 (naively) \( O(n \cdot m) \) space
- Sparse matrix: Most entries are zero
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