empty stack push(a) push(b) pop(c)

\[
\begin{align*}
\text{tos} & = -1 \\
\text{tos} & = 0 \\
\text{tos} & = 1 \\
\text{tos} & = 0
\end{align*}
\]

\[
\text{stack}[\text{tos}] = a \quad \text{stack}[\text{tos}] = b \quad \text{return stack}[\text{tos} + 1] = \text{stack}[1] = b
\]

\[
\begin{align*}
\text{isEmpty}(?) & \quad \text{return} \quad \text{tos} = \text{size} - 1 \\
\text{size}(\cdot) & \quad \text{return} \quad \text{tos} + 1
\end{align*}
\]

Running Times:
- push(e) \quad \text{O(1)}
- pop() \quad \text{O(1)}
- top() \quad \text{O(1)}
- isEmpty() \quad \text{O(1)}
- size() \quad \text{O(1)}

Doubly Linked List Implementation of Stack:
Just keep track of top-most node.
Store: Node called top, initially = null

\[
\begin{align*}
\text{newNode} & \quad \text{newNode} = \text{new Node(e)} \\
\text{newNode.prev} & \quad \text{newNode.prev} = \text{top} \\
\text{top} & \quad \text{top} = \text{newNode}
\end{align*}
\]
pop()

tmp = top.element

(top = top.prev)

return tmp.element

top()

return top.element

ex: Stack s = new Stack()

top = null

(Empty Stack)

s.push(a)  s.push(b)

s.pop()  s.top()

pop()  returns a

Run-Times:

push(t)  size?  O(1)

pop()  keep int variable size.  O(1)

top()  in push(), size++

isEmpty()  pop(): size--
Stacks were LIFO (Last In First Out)

In real world, many processes are FIFO - first in first out

Queue

Stack

ex: customer service phone calls

printer queues

Operations:

- `enqueue(e)`: insert e at back
- `dequeue()`: remove & return element at front.

`dequeue ← enqueue

front ↑ back

peek(): view element at front.

isEmpty():

size()
Array Implementation: Assume some (>1) elements in Queue

Store:

1) Array (Q) represents the queue (sufficiently large)
2) 3 integers:
   - front: index of front element initially = -1
   - back: back
   - currentSize: # of elements in queue = 0

enqueue (a)

   currentSize ++
   back ++
   Q[back] = a

decqueue ()

   currentSize --
   front ++
   return Q[front - 1]

Suppose queue looks like this:

```
   [---] c d e [---]
```

Empty slots: 'end of array'

Now:

enqueue(f)

```
   [---] c d e [---]
   front  back
```

enqueue(g)
enqueue(h) ?

Now can't add even though there is space at front.

SOLN: circular array- "wrap-around" array
end of array wraps around to beginning

Now: enqueue(h):

```
0 1 2 3 4 5 6  
| h | c | d | e | f | g |
```

W. length = 7

- back
- front

Problem? front will go to the end (after some dequeues)

after 4 dequeues:

```
| h | c | d | e | f | g |
```

- back
- front

After next dequeue():

```
| h | c | d | e | f | g |
```

front should point here

How to implement? Insert checks before front++ & back++

In enqueue():

- if (back == Q.length - 1)
  - back = 0

In dequeue():

- if (front == Q.length - 1)
  - front = 0
Linked List Implementation of Queues

Maintain:
- Node front: Pointer to front, initially = null
- Node back: "back, "
- int currentSize: number of elements = 0

\[ \text{a} \rightarrow \text{b} \rightarrow \text{c} \rightarrow \text{d} \rightarrow \text{null} \]

1. enqueue(e)
   - Node \( n = \text{new Node}(e) \)  
   - \( \text{back}\cdot\text{next} = n \)
   - \( \text{back} = n \)
   - \( \text{currentSize}++ \)

2. dequeue()
   - Node \( \text{tmp} = \text{front} \)
   - \( \text{front} = \text{front}\cdot\text{next} \)
   - \( \text{currentSize}-- \)
   - return \( \text{front}\cdot\text{element} \)

3. isEmpty() : return \( \text{currentSize} = 0 \)

4. size() : return \( \text{currentSize} \).