Stacks were LIFO (Last In First Out)

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

Queue

Stack

- enqueue(e): insert e at back
- dequeue(): remove return element at front
- peek(): view element at front
- isEmpty()
- size()
Array Implementation - Assume some (>1) elements in Queue.

Data Fields:

Object[] Q = new Object[10000];  (sufficiently large)
3 integers front, index of front element initially = -1
   back  "  back  "  = -1
   currentSize  "  of elements in queue  = 0

Methods:

enqueue (Object e) // Assume non-empty:

   currentSize++
   back++
   Q[back] = e

decqueue ()

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

size () : return currentSize
is_empty () : return currentSize = 0.

Suppose queue looks like this:

--- c d e | |

Empty slots "end up array." 

Now:

enqueue(f) --- c d e | f |

enqueue(g)  front  back
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):

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>.</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
<td>g</td>
</tr>
</tbody>
</table>

Q, length = 7

↑
↑
back front
↑
↑
back
①
②

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

after 4 dequeues: → h ... [ ] [ ] [ ] [ ] [ ] [ ]

back front

member, e assume we won't rerun at end of race, so e don't worry about rer-writng in existing element

After next dequeue():
front should point here "back/front"

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

At top of enqueue():
if (back == Q.length-1)
    back = 0
    or back = c
else...
In dequeue():
if (front == Q.length-1)
    front = 0
    currentSize--
    return Q[Q.length-1]
else...
Linked List Implementation of Queues:

Data fields:

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

1. enqueue (Object e)
   - Node n = new Node (e)
   - back.next = n
   - back = n
   - currentSize ++

2. dequeue()
   - Node tmp = front
   - front = front.next
   - currentSize --
   - return tmp.element
   - tmp = null

3. isEmpty(): return currentSize == 0

4. size(): return currentSize.
Recursion (Review) - programming technique where solution to the problem depends on solutions to sub-problems.

2 Parts:
- Base Case - solved without recursion
- Recursive Step - makes progress toward base case

1. Summing values from 1 to n.

Problem: Summing from 1 to n.

Sub-problem: """" to values < n.

\[ \sum_{i=1}^{n} i = 1 + 2 + 3 + 4 + \ldots + n \]

\[ = n + n - 1 + n - 2 + \ldots + 1 \]

Write in terms of smaller versions of the same problem (summing)?

\[ \sum_{i=1}^{n} i = n + \left( \sum_{i=1}^{n-1} i \right) \text{ recursion} = \]

\[ = n + (n - 1) + \left( \sum_{i=1}^{n-2} i \right) \text{ recursion} \]

We can stop expressing as a smaller problem when we get to a point where we can solve the problem directly. Which value of \( n \)? \( n=1 \rightarrow \) Base Case.
Base Case: For $n=1$: $\sum_{i=1}^{n} i = 1 + \bigcirc$ no recursion!

Write as a recursive function:

```c
int sum(n)
{
  if (n == 1)
    return 1
  else
    return (n + sum(n-1));
}
```

ex: $\text{sum(5)} = 15$

5 + $\text{sum(4)} = 5 + 10 = 15$

4 + $\text{sum(3)} = 4 + 6 = 10$

3 + $\text{sum(2)} = 3 + 3 = 6$

2 + $\text{sum(1)} = 2 + 1 = 3$

Base Case? $n=1$ (or $n=0$). ⇒ then $n! = 1$