STACK / QUEUE IMPLEMENTATIONS

Run-time Analysis

Revisit: Array or Linked List?

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>STACK:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>push</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>find_kth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUEUE:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>enqueue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dequeue</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert_at_kth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEQUE (double-ended queue):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert_front</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove_front</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert_back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remove_kth</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Circular Array**

Accessing next element:
- `index = (index + 1) % max_length`

Accessing previous element:
- `index = (index – 1 + max_length) % max_length`

---

**Two Stacks in a Single Array**

Where to place them?

Three stacks?

Two Queues in an Array?
Stack With a "Max" Operation

- How to return the max element?
  - Search?
  - Remember?

Exponentiation: For Loop

- What is the complexity of the following code?

```java
public static long pow (long x, int n) {
    if (n == 0)
        return 1;
    long prod = x;
    for (int i=0; i < n; i++)
        prod = prod*x;
    return prod;
}
```
Recursive (Efficient) Exponentiation

- What does the following code do?
  ```java
  public static long pow(long x, int n) {
    if (n == 0)
      return 1;
    if (n == 1)
      return x;
    if (isEven(n))
      return pow(x*x, n/2);
    else
      return pow(x*x, n/2) * x;
  }
  ```

- On PIAZZA: What is its complexity?

O(.): Impact of Constants

- If one implementation is O(log n) and another is O(n)
  - Which is faster?

- Could depend on:
  - Constant factors:
    - How many assignments, additions, accesses for each n?
      - e.g., T(n) = 900,000n vs. T(n) = 2n²
    - Size of n:
      - If n is small, constant additive factors may be important
        - e.g., T(n) = 900,000 + log n vs. T(n) = 2 + n

- O(.) ⇒ there is some \( n_0 \), such that O(n) faster for \( n > n_0 \)
Summary: Run-time vs. O(.)

- O(.):
  - Helps reason about performance as a function of n

- Run-time:
  - Important for small n
    - Or not-so-small n, depending on the constant factors
  - Cross-over point $n_0$ changes as technology advances
    - CPU and memory speed
    - New efficient compilers
    - Clever programming

- Demo: Tester.java