Stacks and queues
(Weiss chapter 3)

Stacks

A *stack* stores a sequence of values

Main operations:

- *push(x)* – add value *x* to the stack
- *pop()* – remove the *most-recently-pushed* value from the stack

LIFO: *last in first out*

- Value removed by *pop* is always the one that was pushed most recently

Example:

- *push(1); push(2); pop(); push(3); pop(); pop()*
- *First pop returns 2, second pop returns 3, third pop returns 1*

Stacks

Analogy for LIFO: stack of plates

- Can only add or remove plates at the top!
- You always take off the most recent plate

Stacks

More stack operations:

- *is stack empty?* – is there anything on the stack?
- *top()* – return most-recently-pushed ("top") value without removing it

Example: balanced brackets

Given a string:

> “hello **(**hello is a greetng ***[****sic****]*** **{**“sic” is used when quoting a text that contains a typo **(**or archaic **[**and nowadays perhaps wrong**]** spelling**)** to show that the mistake was in the original text **(**and not introduced while copying the quote**)}) ”**

Check that all brackets match:

- Every opening bracket has a closing bracket
- Every closing bracket has an opening bracket
- Nested brackets match up: no “**([)]**”!

Algorithm

Maintain a *stack* of opened brackets

- Initially stack is empty
- Go through string one character at a time
- If we see an opening bracket, push it
- If we see a closing bracket, pop from the stack and check that it matches
  - e.g., if we see a "**)**", check that the popped value is a "**(**"
- When we get to the end of the string, check that the stack is empty

Algo

Check your understanding:
What has gone wrong
if each of the steps
written in bold fails?

Maintain a *stack* of o

- Initially stack is empty
- Go through string one character at a time
- If we see an opening bracket, push it
- If we see a closing bracket, **pop** from the stack and **check that it matches**

  (pop fails if stack is empty)

  – e.g., if we see a "**)**", check that the popped value is a "**(**"
- When we get to the end of the string, **check that the stack is empty**

More uses of stacks

The *call stack*, which is used by the processor to handle function calls

- When you call a function, the processor records what it was doing by pushing a record onto the call stack
- When a function returns, the processor pops a record off the call stack to see what it should carry on doing

Parsing in compilers

Lots of uses in algorithms!

Implementing stacks in Java

Idea: use a dynamic array!

- Push: add a new element to the end of the array
- Pop: remove element from the end of the array

Complexity: all operations have *amortised* O(1) complexity

- Means: n operations take $O(n)$ time
- Although a single operation may take O(n) time, an “expensive” operation is always balanced out by a lot of earlier “cheap” operations

Abstract data types

You should distinguish between:

- the *abstract data type (ADT)* (a stack)
- its *implementation* (e.g. a dynamic array)

Why?

- When you *use* a data structure you don't care how it's implemented
- Your code and design will be clearer if you use ADTs
- Most ADTs have many possible implementations

Queues

A *queue* also stores a sequence of values

Main operations:

- *enqueue(x)* – add value *x* to the queue
- *dequeue()* – remove *earliest-added* value

FIFO: *first in first out*

- Value dequeued is always the *oldest* one that's still in the queue

Much like a stack – but FIFO, not LIFO

Queues

Like a queue in real life!

- The first to enter the queue is the first to leave

Uses of queues

Controlling access to shared resources in an operating system, e.g. a printer queue

A queue of requests in a web server

- Generally, message queues are used to send information between processes in concurrent programs

Also appears in lots of algorithms

- (Stacks and queues both appear when an algorithm has to remember a list of things to do)

Implementing queues in Java

What's wrong with this idea?

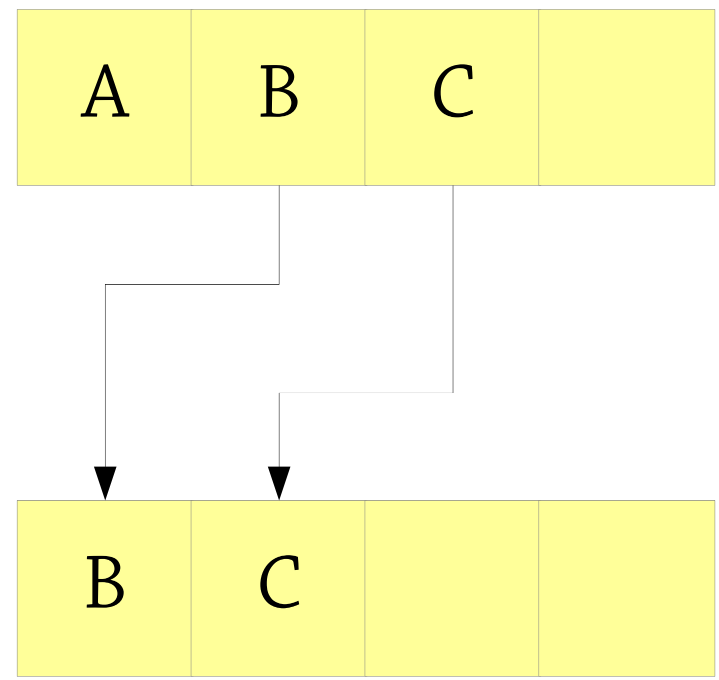
- Implement the queue as a dynamic array
- *enqueue(x)*: add *x* to the end of the dynamic array
- *dequeue()*: remove and return first element of array

To dequeue, we'd have to copy the entire rest of the array down one place... takes O(n) time

Dynamic arrays are no good

A queue containing A, B, C:

Dequeue removes A:



Moving the rest of the queue into place takes O(n) time!

Bounded queues

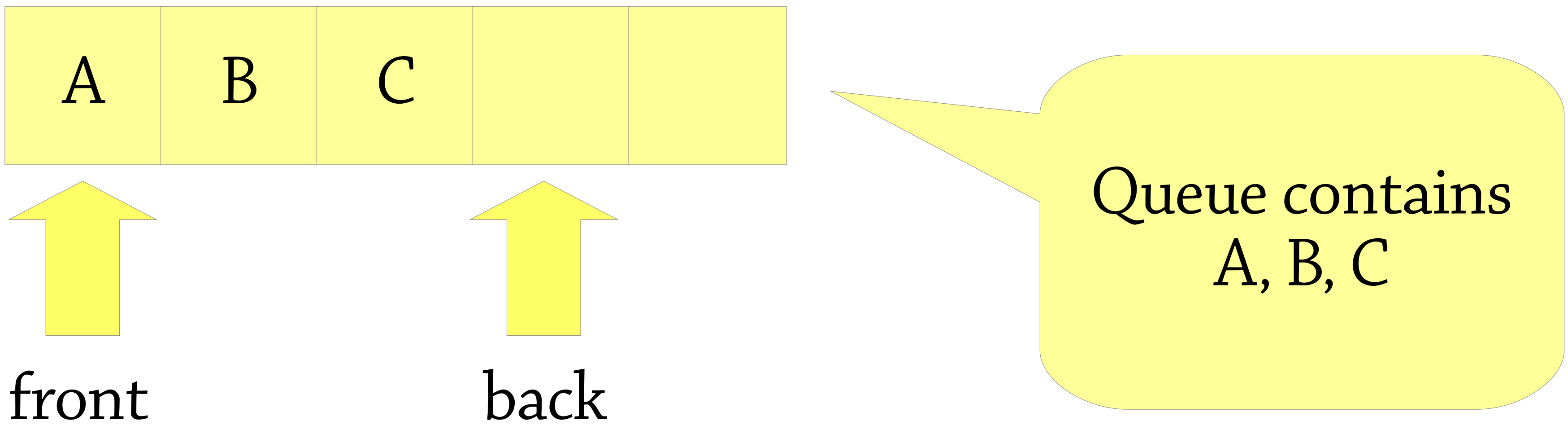
Let's solve a simpler problem first:
bounded queues

A bounded queue is a queue with a fixed capacity, e.g. 5

- The queue can't contain more than 5 elements at a time
- You typically choose the capacity when you create the queue

Bounded queues

An array, plus two indices *back* and *front*

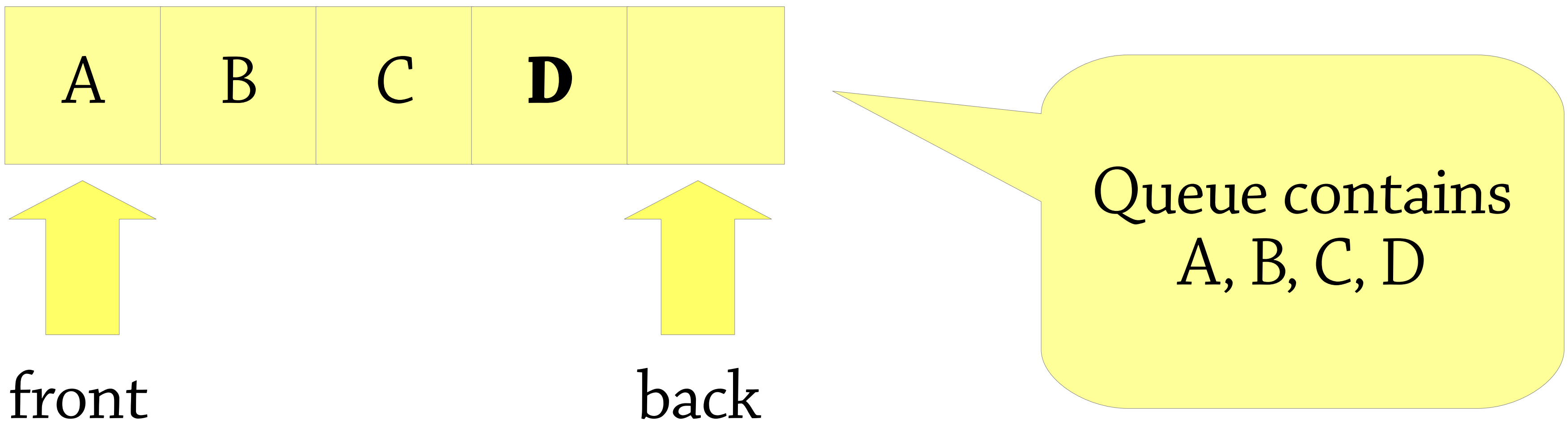


back: where we enqueue the next element

front: where we dequeue the next element

Bounded queues

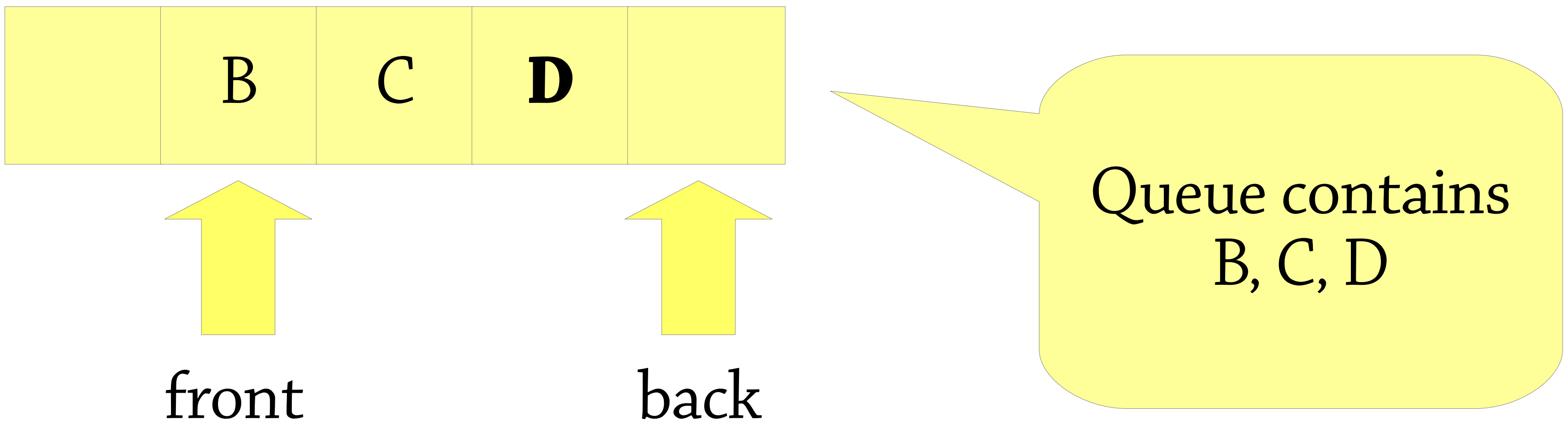
After enqueueing D



array[back] = D; back = back+1

Bounded queues

After dequeueing (to get A)



result = array[front]; front = front+1

Thinking formally about queues

What is the contents of one of our array-queues?

- Everything from index *front* to index *back-1*

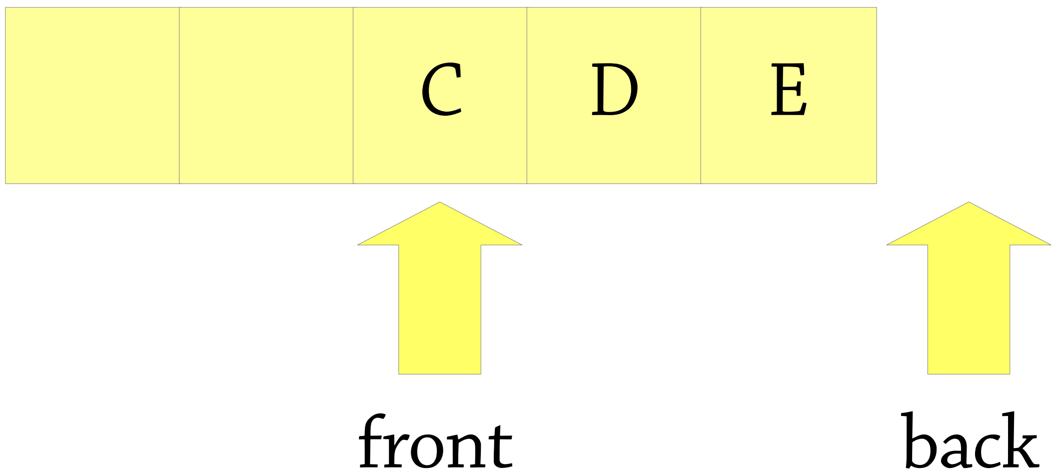
If we specify the *meaning* of the array like this, there is only one sensible way to implement *enqueue* and *dequeue*!

- Before dequeue:
  contents is *array[front], array[front+1], ..., array[back-1]*
- After dequeue: *array[front]* should be gone,
  contents is *array[front+1], ..., array[back-1]*
- Only good way to do this is *front = front + 1*!

Data structure design hint:
don't just think what everything should do!
Work out the *meaning* of the data structure too.

Bounded queues

After enqueueing E and dequeueing



What's the problem here?

Queues as circular buffers

Problem: when *back* reaches the end of the array, we can't enqueue anything else

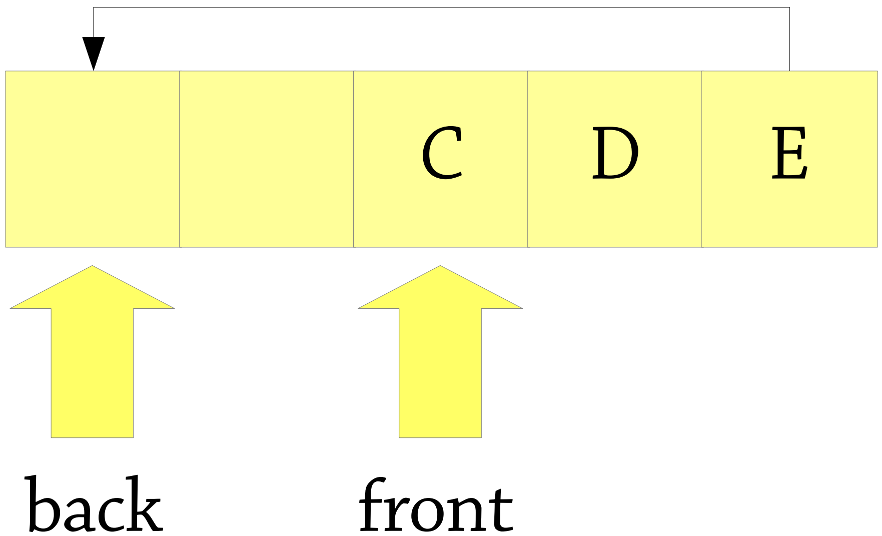
Idea: *circular buffer*

- When *back* reaches the end of the array, put the next element at index 0 – and set *back* to 0
- Next after that goes at index 1
- *front* wraps around in the same way

Use all the extra space that's left in the beginning of the array after we dequeue!

Bounded queues

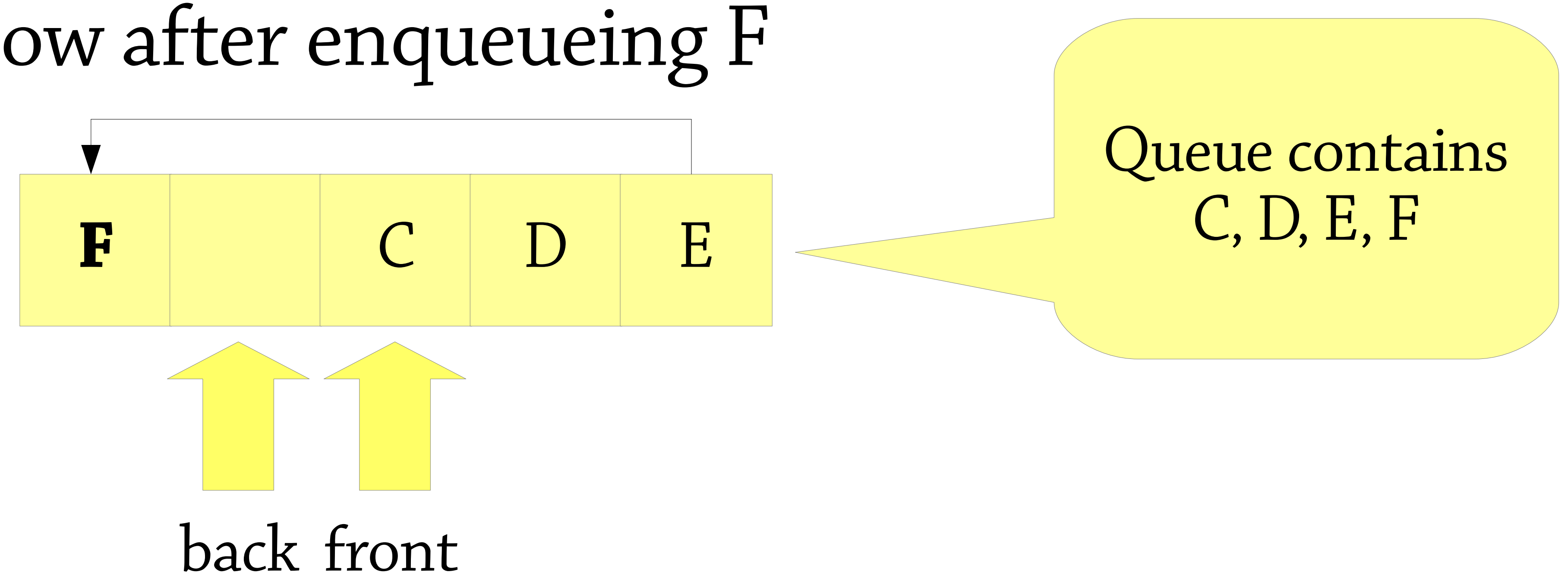
Try again – after enqueueing E



back wraps around to index 0

Bounded queues

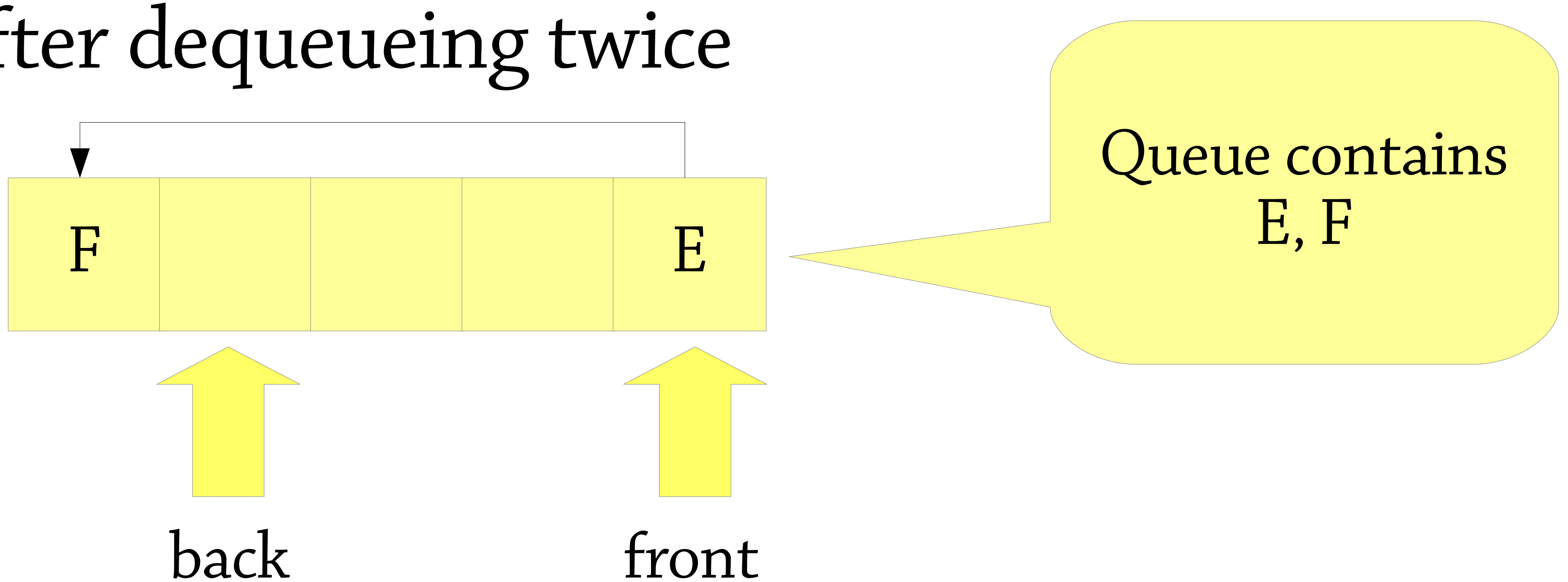
Now after enqueueing F



Meaning: queue contains everything from *front* to *back-1* still.
But wrapping around if *back* < *front*!
Exercise: phrase this precisely.

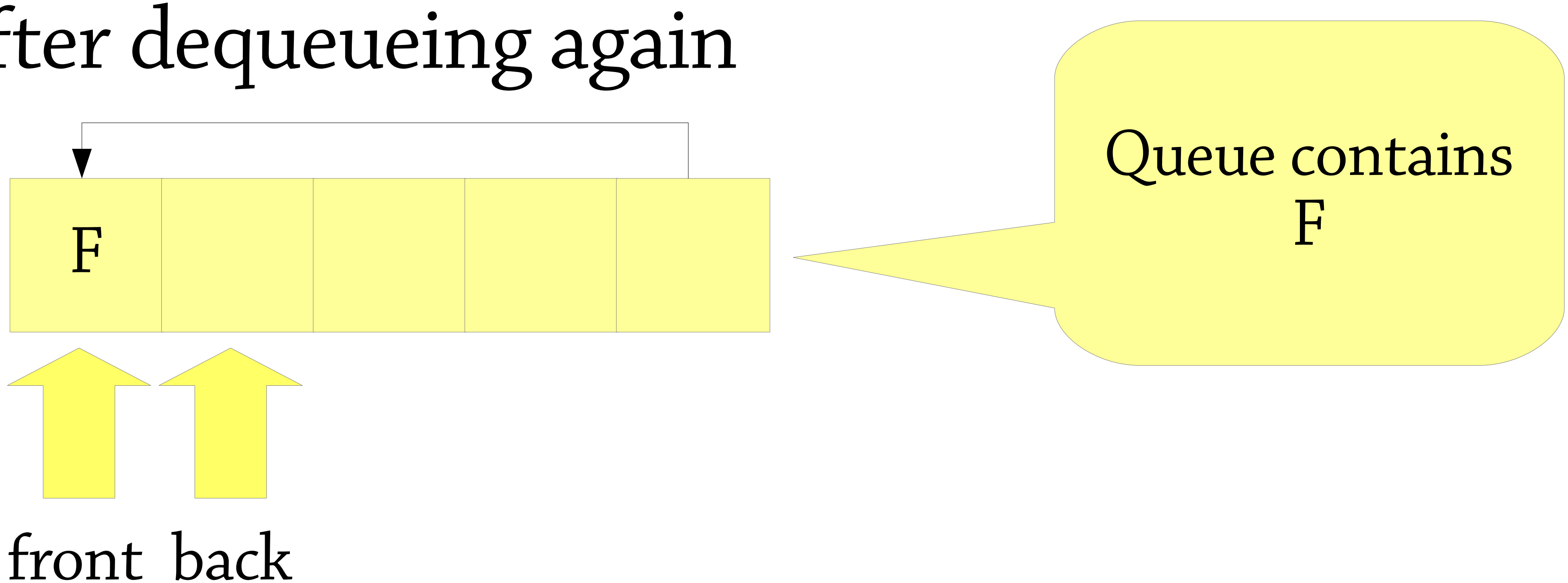
Bounded queues

After dequeueing twice



Bounded queues

After dequeueing again



front wraps around too!

Circular buffers

Basic idea: an array, plus two indices for the front and back of the queue

- These indices *wrap around* when reaching the end of the array, which is what makes it work

Exercise: what sequence of elements does a circular buffer represent?

The best bounded queue implementation!

Bounded queues

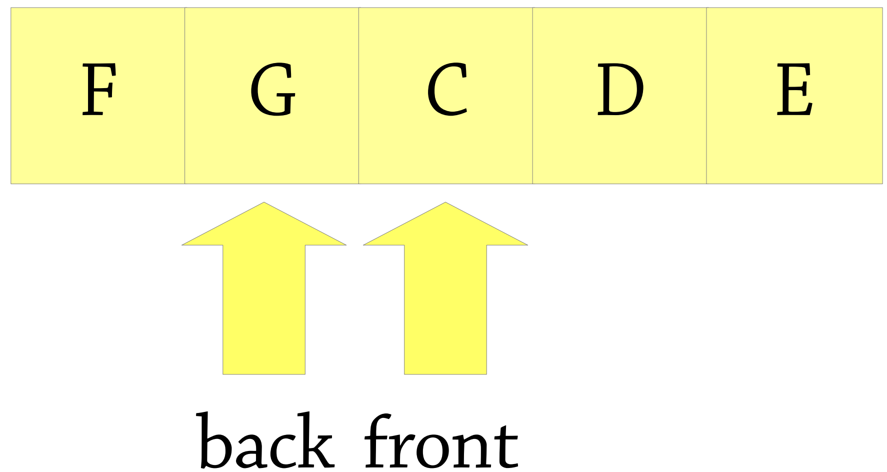
Circular buffers make a fine *bounded queue*

To make an unbounded queue, let's be inspired by dynamic arrays

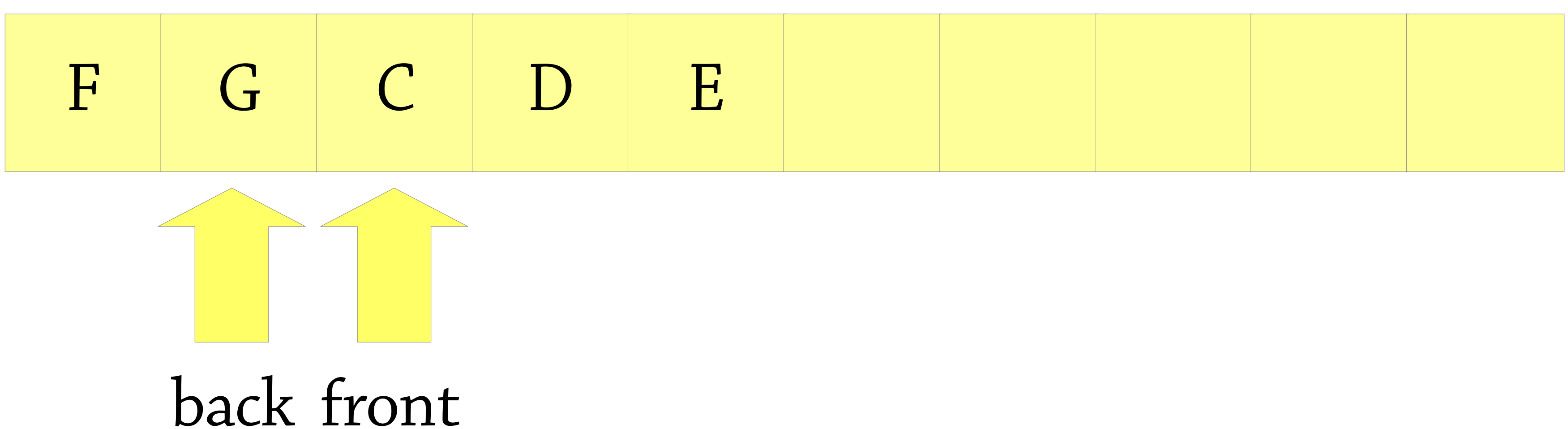
- Dynamic arrays: fixed-size array, double the size when it gets full
- Unbounded queues: bounded queue, double the capacity when it gets full

Whenever the queue gets full, allocate a new queue of double the capacity, and copy the old queue to the new queue

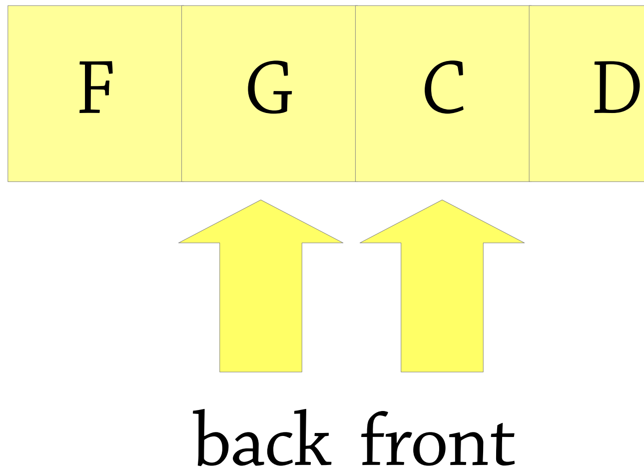
Reallocation, how not to do it



What's wrong with resizing like this?

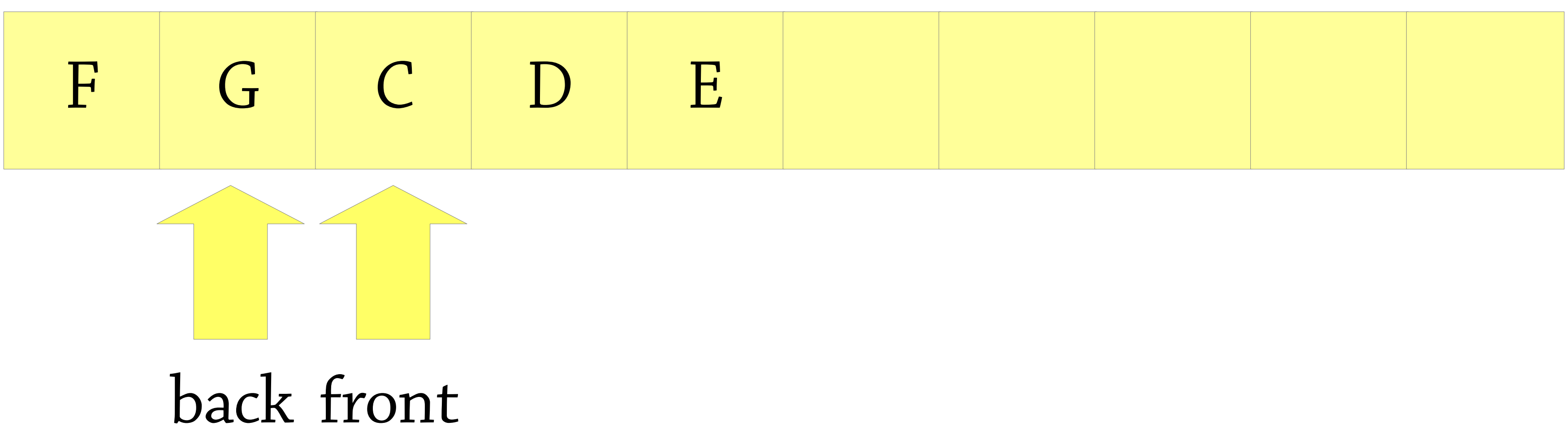


Reallocation, how ne

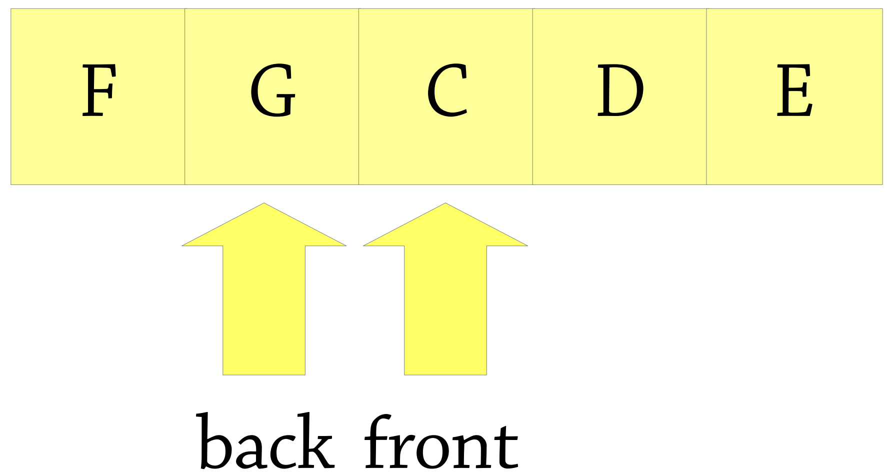


Queue contains
C, D, E,
five blank spaces,
F, G!

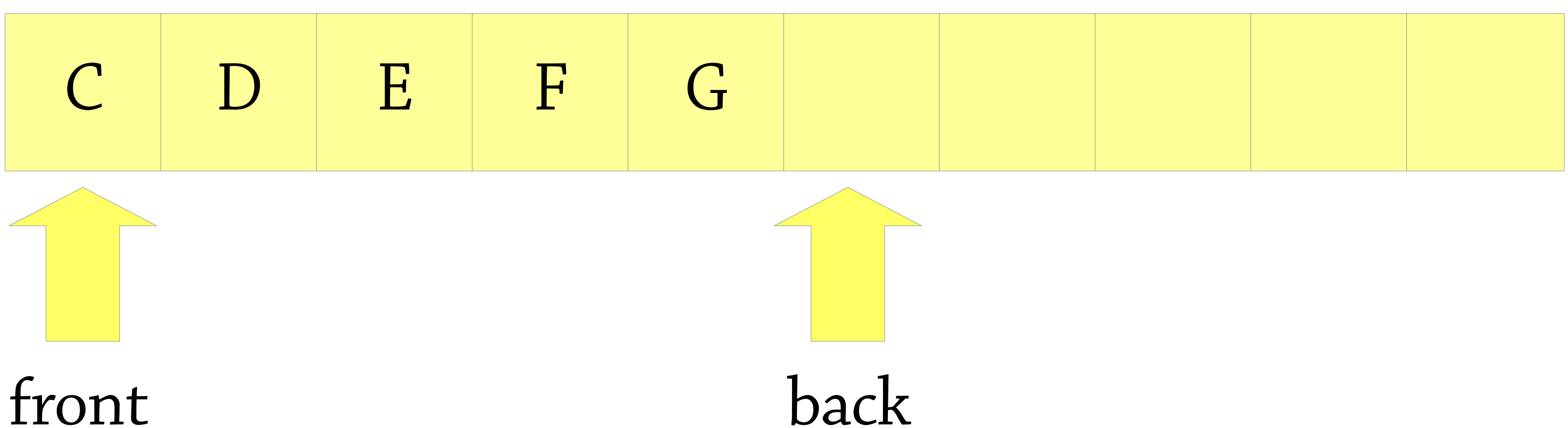
What's wrong with resizing like this?



Reallocation, how not to do it



Instead, repeatedly dequeue from the old queue and enqueue into the new queue:



Summary: queues as arrays

Maintain *front* and *back* indexes

- Enqueue elements at *back*, remove from *front*

Circular array

- *front* and *back* wrap around when they reach the end

Idea from dynamic arrays

- When the queue gets full, allocate a new one of twice the size
- Don't just resize the array – safer to use the queue operations to copy from the old queue to the new queue

Important implementation note!

- To tell when array is full, need an extra variable to hold the current *size* of the queue (exercise: why?)

Double-ended queues

So far we have seen:

- Queues – add elements to one end and remove them from the other end
- Stacks – add and remove elements from the same end

In a *deque*, you can add and remove elements from *both ends*

- *add to front, add to rear*
- *remove from front, remove from rear*

Good news – circular arrays support this easily

In practice

Your favourite programming language should have a library module for stacks, queues and deques

- Java: use `java.util.Deque<E>` – provides `addFirst/Last`, `removeFirst/Last` methods
- Note: Java also provides a `Stack` class, *but this is deprecated – don't use it*

Stacks, queues, deques – summary

All three extremely common

- Stacks: LIFO, queues: FIFO, deques: generalise both
- Often used to maintain a set of tasks to do later
- Common implementation: stacks are dynamic array, queues are circular buffers, O(1) *amortised* complexity

Data structure design hint: always think about what the representation of a data structure *means!*

- e.g. "what queue does this circular buffer represent?"
- This is the main design decision you have to make – it drives everything else
- This lets you design new data structures systematically
- And also understand existing ones