Stacks and queues
(chapters 6.6, 15.1, 15.5)

So far...

Complexity analysis

- For recursive and iterative programs

Sorting algorithms

- Bubble, insertion, selection, quick, merge, (intro, dual-pivot quick, natural merge, Tim)

Binary search, dynamic arrays

Rest of course: lots of data structures!

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

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?
- *make new stack*
- *top()* – return most-recently-pushed (“top”) value without removing it

Stacks are used *everywhere*

- Example: *call stack* – whenever you call a function or method, the computer has to remember where to continue after the function returns – it does this by pushing where it had got to onto the call stack

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 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 “**([)]**”!

Maybe many sorts of brackets!

Algorithm

Maintain a *stack* of opened brackets

- Initially stack is empty
- Go through string one character at a time
- If we see a non-bracket character, skip it
- 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

Try it!

n

Maintain a ... ed brackets

- ...ck is empty
- ... string one c... racte...
- ...non-bracket ch... racte...
- If we se... n opening bracke..., push it
- If we see a closing bracket, pop from the stac...
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

Try it!

If it doesn't match, the structure of the brackets is wrong

If stack is empty on pop, there is a closing bracket without an opening one

If stack is not empty at end, an opening bracket is unclosed

Implementing stacks in Haskell

```
type Stack a = ???
push :: a → Stack a → Stack a
pop :: Stack a → Stack a
top :: Stack a → a
empty :: Stack a → Bool
```

[better API:

```
pop :: Stack a → Maybe (a, Stack a)]
```

Stacks are lists!

```
type Stack a = [a]
push :: a → Stack a → Stack a
push x xs = x:xs

pop :: Stack a → Stack a
pop (x:xs) = xs

top :: Stack a → a
top (x:xs) = x

empty :: Stack a → Bool
empty [] = True
empty (x:xs) = False
```

Stacks are lists!

```
type Stack a = [a]
push :: a → St    a
push x xs = x:x

pop :: Stack
pop (x:xs)

top :: Stack
top (x:xs) = x

empty :: Stack a → Bool
empty [] = True
empty (x:xs) = False
```

Note: in functional languages you usually don't bother with a special stack datatype, you just use lists instead

Implementing stacks in Java

Idea: use a dynamic array!

- Push: add a new element to the end of the array
- Pop: decrease size by 1
- Empty?: is size 0?

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

- We don't study amortised complexity in this course
- Means: although one push may take O(n) time (if the array needs to be copied), this happens rarely enough not to affect the complexity of the whole sequence of operations
- Formally means: n operations take $O(n)$ time

Stacks using dynamic arrays

`push(1); push(2)`

1	2

`top = 1`

`push(3)`

1	2	3	

`top = 2`

`pop()`

1	2	3	

`top = 1`, return 3

Queues

A queue is similar to a stack:

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

Difference: FIFO (*first in first out*)!

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

Used all over the place – not quite as often as stacks

- Example: controlling access to shared resources in an operating system, e.g. a printer queue

Queues

Analogy for FIFO: a queue!

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

Implementing queues in Java

One idea: a dynamic array as before

- *enqueue(x)*: add *x* to the end of the dynamic array
- *dequeue()*: return first element of array…
  …but how to remove it?

One option for *dequeue*:

- copy the contents of the array down one index…
  a[1] to a[0], a[2] to a[1], etc.

But this gives O(n) performance for *dequeue*! We want O(1).

Attempt two

Implement a queue as an array…
…but keep *two* indices into the array:

- *rear*: the index where we enqueue elements
- *front*: the index where we dequeue elements
- Compare with stacks, where we had an array plus *one* index (the top of the stack)

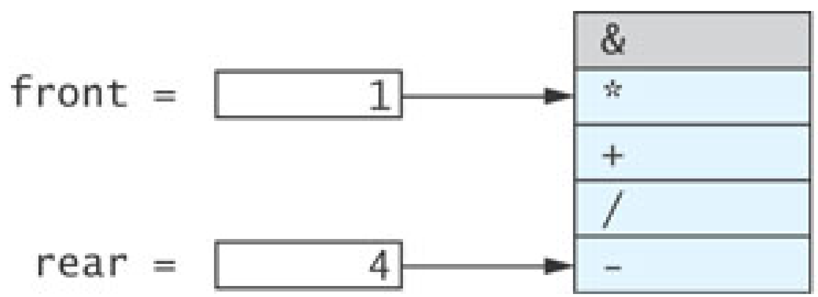
To enqueue an element, increment *rear* and put the new element there

To dequeue, take the element from *front* and increment *front*

An example queue

What values do we get if w
call de*queue* on this queue?

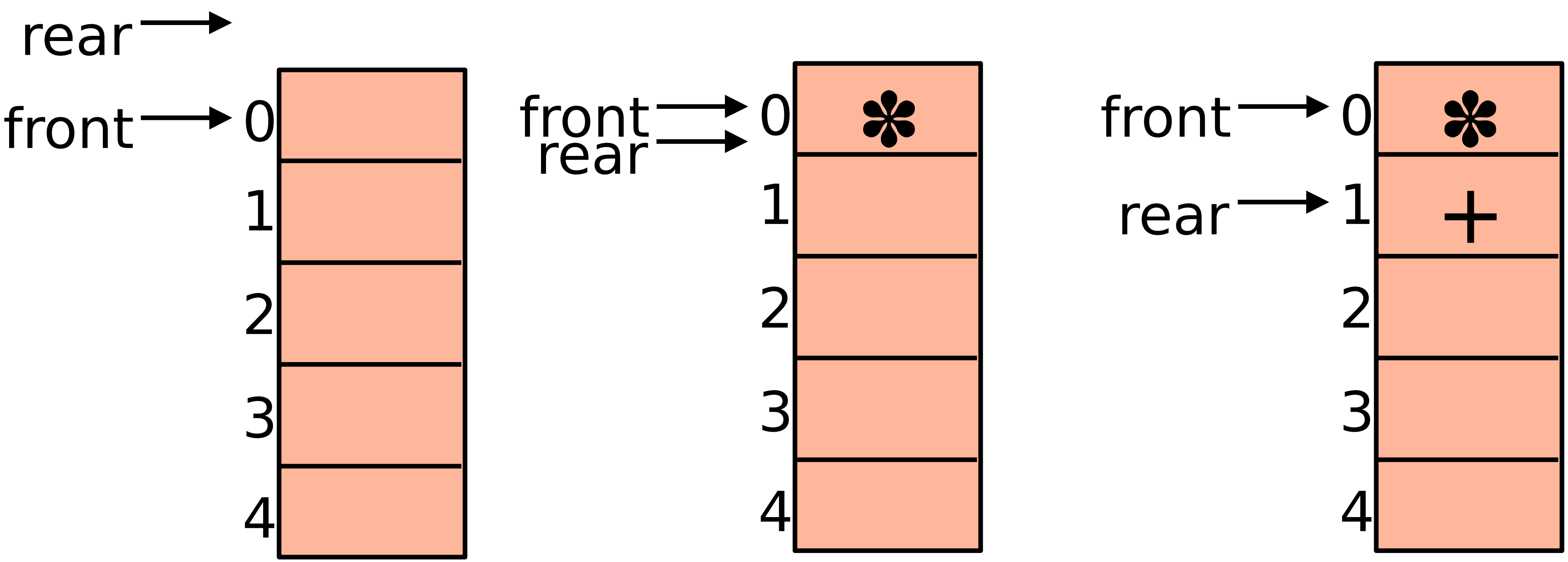
Queue contains everything between front and rear: `* + / -`



(Important first step in understanding a data structure: understand how its *representation* works. First step when designing a data structure too!)

A longer example

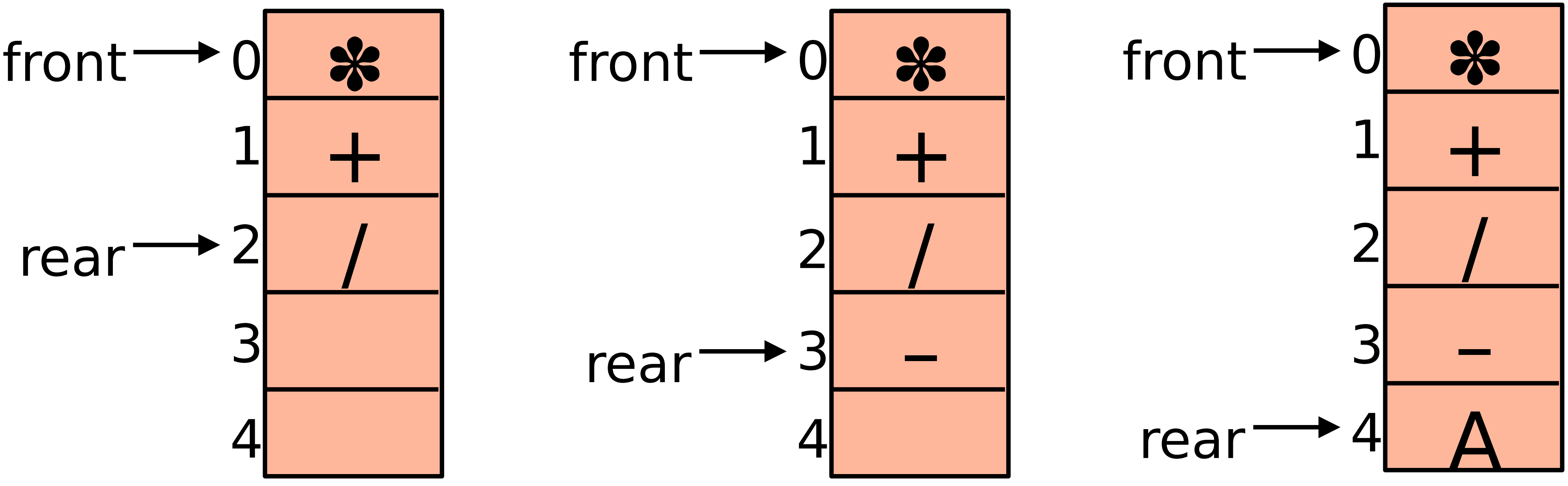
Enqueue *, +, /, -, A, dequeue two elements:



N.B.: initialise *rear* to -1 so that initially [front..rear] is empty!

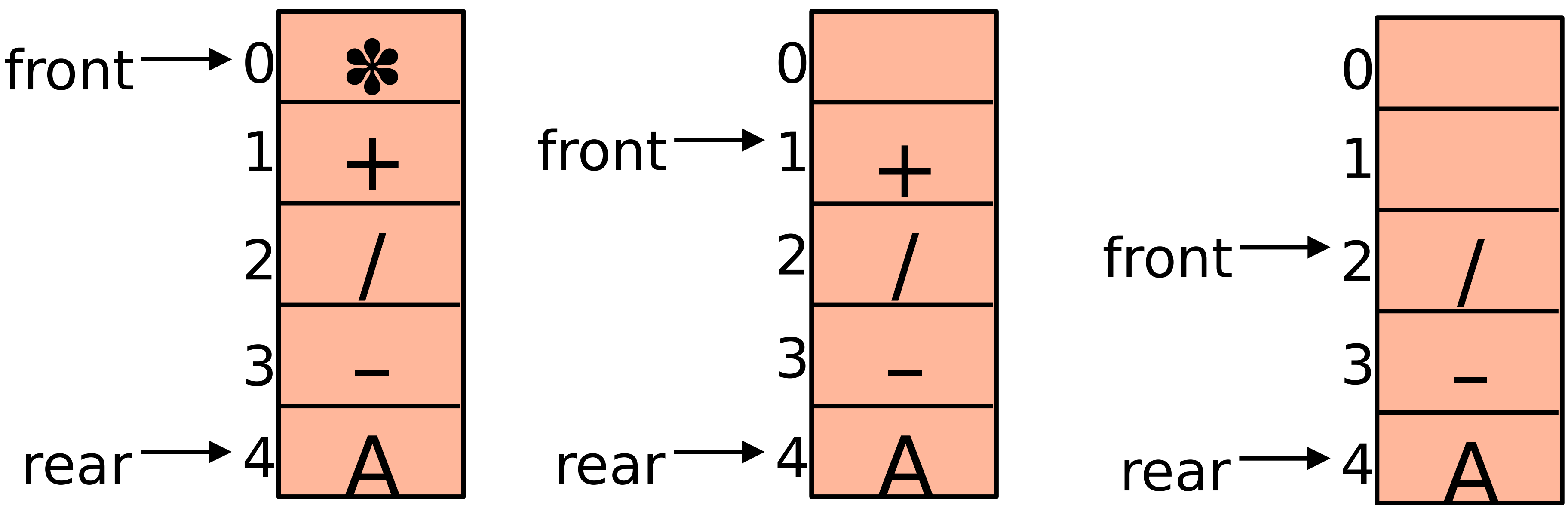
A longer example

Enqueue *, +, /, –, A, dequeue two elements:



A longer example

Enqueue *, +, /, −, A, dequeue two elements:



But: what happens if we want to enqueue another value now?

Queues as circular arrays

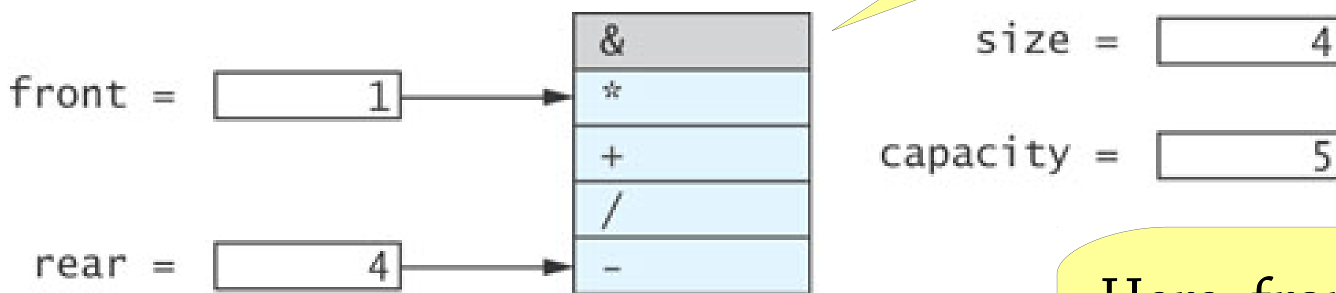
Problem: when *rear* reaches the end of the array, we can't enqueue anything else

Idea: *circular array*

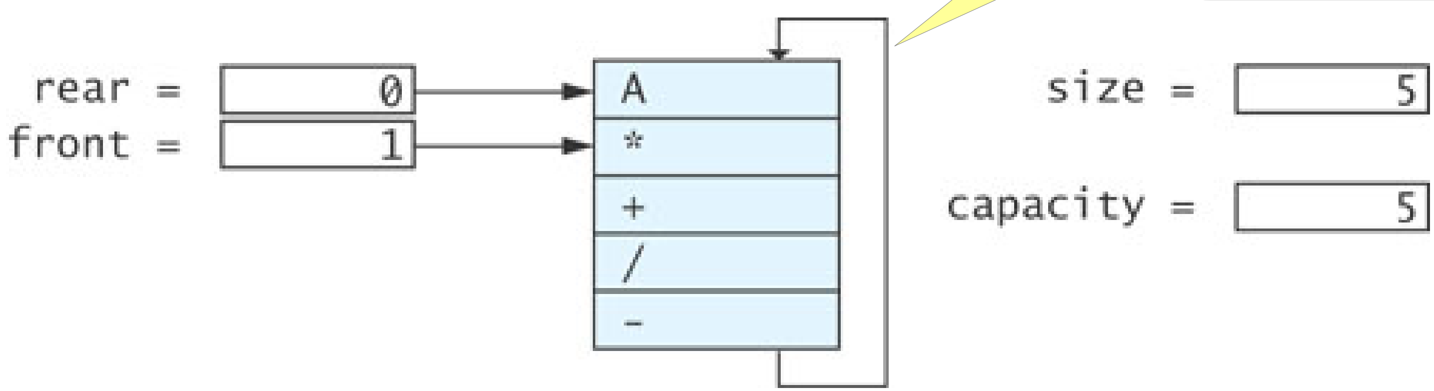
- When *rear* reaches the end of the array, put the next element at index 0 – and set *rear* to 0
- Next after that goes at index 1
- *front* wraps around in the same way

An example circular array

Queue contains everything between front and rear: `* + / -`

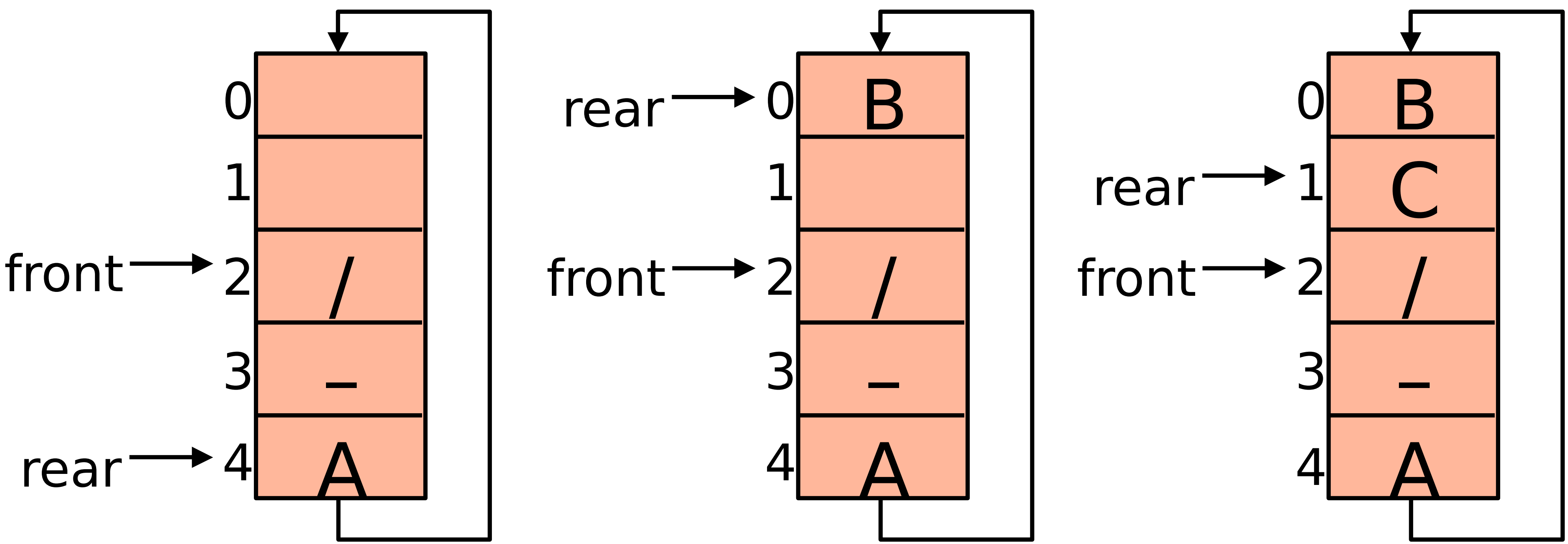


Here, front comes *after* rear! Array is circular: queue contains `* + / - A`



Continuing the example

Starting where we left off, enqueue B and C:



But: what happens if we want to enqueue another value now?

Bounded queues

Our idea of circular arrays works… *until the array gets full*

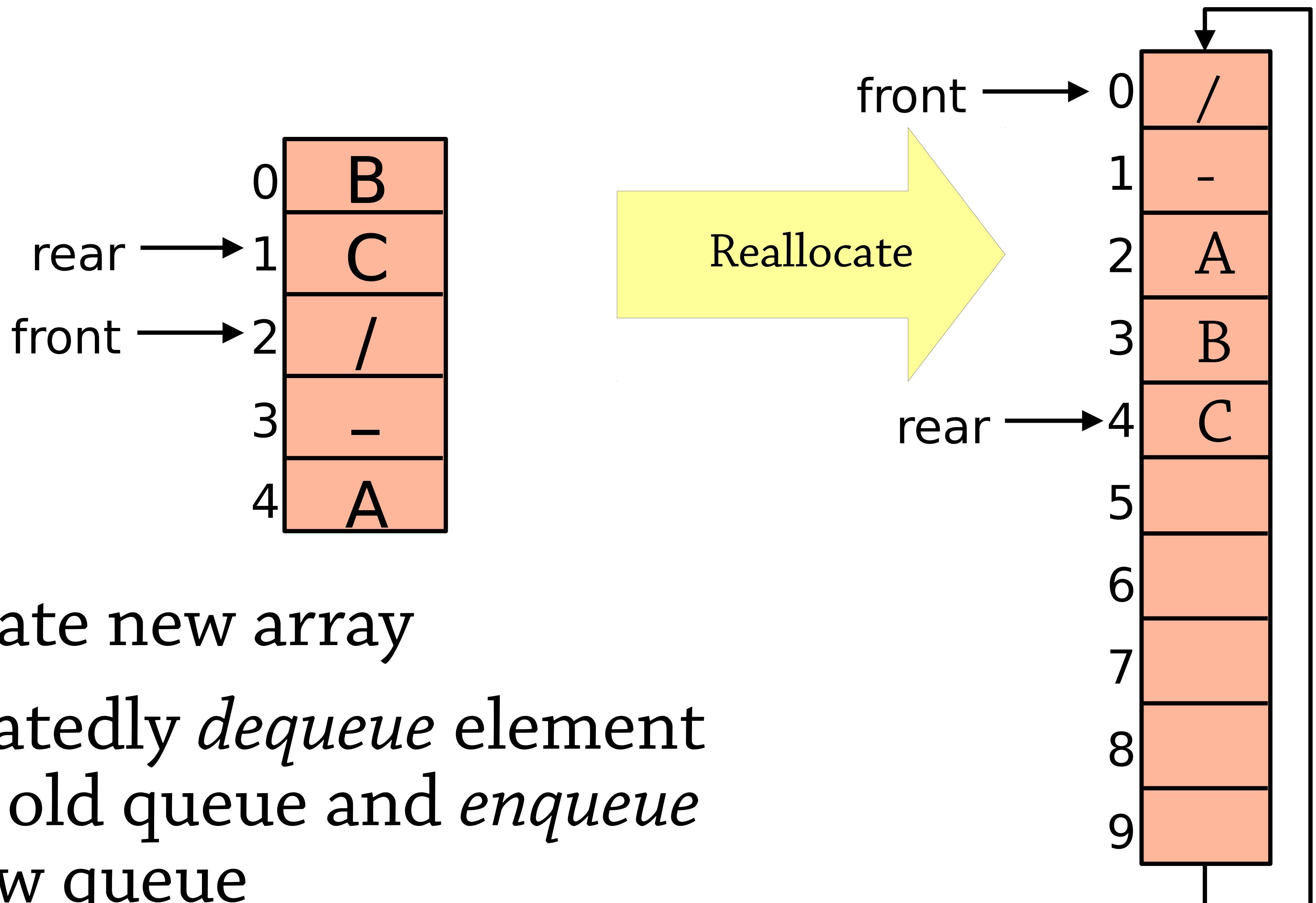
What we have so far is a *bounded queue*

- Queue with a fixed capacity, decided when you create the queue
- O(1) enqueue, dequeue

To turn it into an unbounded queue, let's use the idea of a dynamic array:

- When the queue gets full, double its size

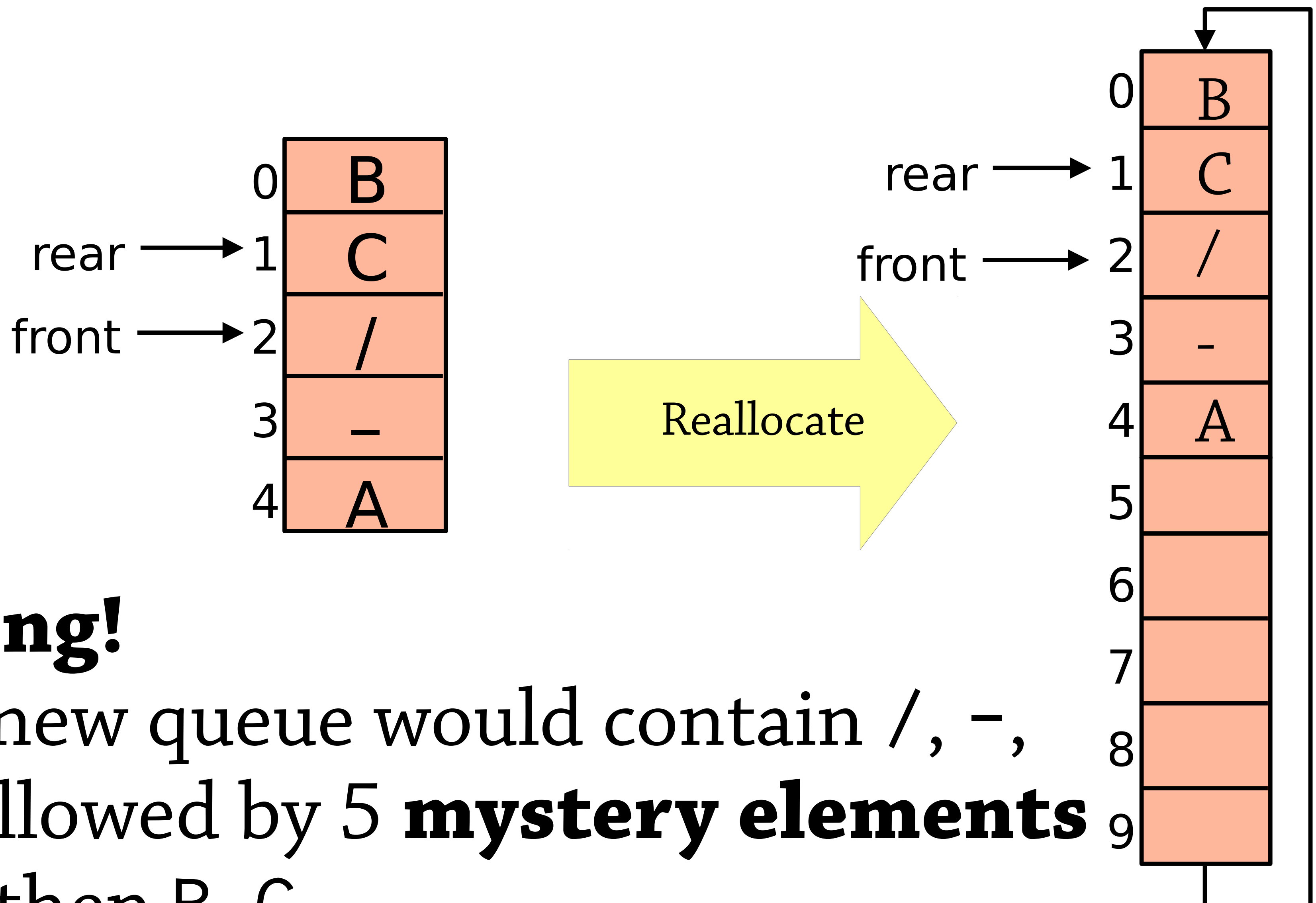
Reallocation



Allocate new array

Repeatedly *dequeue* element from old queue and *enqueue* in new queue (don't just copy array! Why?)

Reallocation, how not to do it



Wrong!

The new queue would contain /, –, A, followed by 5 **mystery elements** and then B, C.

Summary: queues as arrays

Maintain *front* and *rear* indexes

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

Circular array

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

Idea from dynamic arrays

- When the array gets full, allocate a new one of twice the size

Important implementation note!

- To tell when array is full, need an extra variable to hold the current *size* of the queue

Queues in Haskell

```
type Queue a = ???
enqueue :: a → Queue a → Queue a
dequeue :: Queue a → (a, Queue a)
empty :: Queue a → Bool
```

[better API:

```
dequeue :: Queue a → Maybe (a, Queue a)]
```

One possibility: using a list

```
type Queue a = [a]
enqueue :: a → Queue a → Queue a
enqueue x xs = xs ++ [x]

dequeue :: Queue a → (a, Queue a)
dequeue (x:xs) = (x, xs)

empty :: Queue a → Bool
empty [] = True
empty (x:xs) = False
```

But enqueue takes O(n) time!

An analogy

Back to the "stack of plates" analogy!

I am washing plates, you are putting them away

- You want to put the dishes away in the same order I wash them – FIFO, a queue of plates

Idea: we both have a stack of plates

- Me: plates I've washed
- You: plates you're going to put away

If you run out of plates, you take my stack of washed plates. But – the oldest plates are at the bottom! So first *turn the stack upside down!*

Two stacks of plates

Represent a queue as a *pair* of lists/stacks

- `(xs, ys)`
- To enqueue an element, add it to `ys`
- To dequeue, remove an element from `xs`
- If `xs` is empty, replace it with the reverse of `ys`

Formally, the queue `(xs, ys)` represents the sequence `xs ++ reverse ys`

- For example, `([1,2,3], [6,5,4])` represents the queue `1 2 3 4 5 6`

Complexity of this

The bit that is not constant time is when we replace `xs` with `reverse ys`

- This takes O(n) time if `length ys` = n
- But it moves n elements to `xs`
- Hence the enqueues and dequeues for those n elements will take O(n) time
- Alternatively: `reverse` takes O(1) time per element moved, and each element is moved once before being dequeued

We can do n operations in O(n) time

- O(1) *amortised* time per operation

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

- For the functional version, have to be a bit careful to get the right complexity – see exercise

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
- For using as a queue, provides `add = addFirst`, `remove = removeLast`
- For using as a stack, provides `push = addFirst`, `pop = removeFirst`
- Note: Java also provides a `Stack` class, *but this is deprecated – don't use it*
- Haskell: instead of a stack, just use a list
- For queues and deques, use `Data.Sequence` – a general-purpose sequence data type

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
- Imperative language: circular arrays, O(1) complexity
- Functional language: stacks are lists, deques can be implemented as a pair of lists with O(1) *amortised* complexity – not quite as efficient

Data structure design hint: always think about what a data structure *represents!*

- In this case, “if I have a stack or queue implemented in such and such way, what sequence of values is it supposed to contain?”
- See “An example circular array” and “Two stacks of plates” slides
- It gives you a handle on *why* the data structure works