The two-list queue, on paper

Programming Paradigms
Session 2 — Part A0, before any code is written

Sections 1–4: in class, about 20 minutes. Sections 5–7: homework.

We are about to build the same queue three times, in three languages. But everything that makes it work — and the one thing that breaks it — is decided *before* a language is chosen. Work through this with pen and paper; there is nothing here to run.

1. Why one list is not enough

Our only building block is the immutable *cons-list* from the lecture: `[]` is the empty list, and x `::` *rest* puts x in front of an existing list. Two facts, both from the lecture's sharing diagram:

- x `::` *rest* costs one step, and *shares* all of *rest*;
- reaching the *last* element of a list of length n costs n steps, and building a new list that differs only at the end costs n steps.

A queue is FIFO: `enqueue` adds at the back, `dequeue` removes from the front — the two operations act at *opposite ends*.

Exercise 1 (warm-up)

Suppose we represent the queue as a single list.

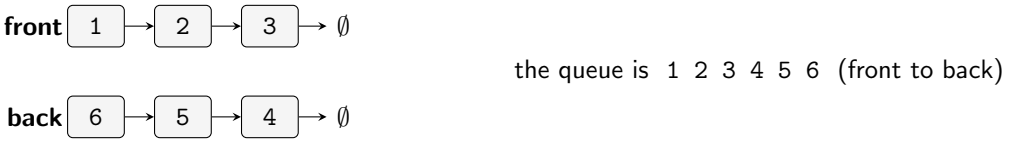
(a) Oldest element at the head. What does `dequeue` *cost? And* `enqueue`*?*

(b) Newest element at the head. Same two questions.

(c) Complete the sentence: with one cons-list, . . .

2. The idea: cut the list in two

Keep **two** lists. `front` holds the oldest elements, oldest at its head. `back` holds the newest elements, *newest* at its head — so `back` is stored backwards. Each list is then used only at its cheap end.



The picture is only a picture. The precise statement is the **abstraction function**: it says which abstract queue a concrete pair stands for.

$$\texttt{contents}(\texttt{front}, \texttt{back}) \;=\; \texttt{front} \texttt{ ++ } \texttt{reverse}(\texttt{back})$$

Exercise 2

Write out `contents` *for each pair.*

(a) (`[1,2]`, `[5,4,3]`)

(b) (`[7]`, `[]`)

(c) (`[]`, `[9,8]`)

Exercise 3

Give ***three different pairs*** *that all denote the same queue* `1 2`*. So the representation is not unique: many concrete pairs stand for one abstract queue. Why is that not a problem?*

3. The invariant

Of the three pairs in Exercise 3, we will forbid one. The rule:

> **Invariant (INV).** if `front` = `[]` then `back` = `[]`
> equivalently: a non-empty queue always has a non-empty `front`

Why impose a rule that rules out perfectly meaningful states? Because of what it buys. Consider a queue with an empty front:

$$q_0 \;=\; (\texttt{[]},\ \texttt{[4,2,1,9,5]}) \qquad \text{whose contents is} \qquad \texttt{5 9 1 2 4}.$$

The oldest element, `5`, sits at the *far end* of `back`. Suppose we allow this state and dequeue five times.

Without the invariant. Each `dequeue` must walk to the end of `back` and rebuild it without its last cell. On a back of length k that costs about k steps, and `back` shrinks by one each time:

$$5 + 4 + 3 + 2 + 1 \;=\; 15 \text{ steps.}$$

With the invariant. The state q_0 is illegal, so it never exists: whoever built it must first reverse `back`, giving (`[5,9,1,2,4]`, `[]`) at a cost of 5 steps. After that every `dequeue` takes the head of `front`, one step each:

$$5 \text{ (one reversal)} \;+\; 5 \times 1 \;=\; 10 \text{ steps.}$$

n elements waiting	without INV	with INV
5	15	10
100	5050	200
general	$n(n+1)/2 = \Theta(n^2)$	$2n = \Theta(n)$

For five elements the gain is small. The point is the last row: without the invariant the cost of draining the queue is *quadratic*, because every single `dequeue` pays to search `back` again. The invariant makes the structure pay that price **once**, and the elements then come out one step at a time. It also keeps `peek` honest: with INV, reading the front element never needs to look at `back` at all.

Exercise 4

Which of these satisfy INV? For the legal ones, say what `peek` *returns and how many steps it takes.*

(`[]`, `[]`) (`[1]`, `[]`) (`[]`, `[2]`) (`[1]`, `[3,2]`) (`[]`, `[3,2]`)

4. The operations

One helper does all the work. Read it as: *never hand out a queue that breaks INV.*

$$\texttt{make}(\texttt{front}, \texttt{back}) = \begin{cases} (\texttt{reverse}(\texttt{back}),\ \texttt{[]}) & \text{if } \texttt{front} = \texttt{[]} \\ (\texttt{front},\ \texttt{back}) & \text{otherwise} \end{cases}$$

$$\texttt{enqueue}(x, (\texttt{f}, \texttt{b})) = \texttt{make}(\texttt{f},\ x \texttt{ :: } \texttt{b})$$

$$\texttt{dequeue}((x \texttt{ :: } \texttt{f}',\ \texttt{b})) = \big(x,\ \texttt{make}(\texttt{f}',\ \texttt{b})\big)$$

$$\texttt{dequeue}((\texttt{[]},\ \texttt{b})) = \text{undefined}$$

Neither operation modifies anything, both return a *new* pair, and **every** queue in the whole program is produced by `make`.

"Undefined" is a design decision, not an oversight. `dequeue` is a *partial* operation: on the empty queue there is no element to return, so there is nothing honest to write. On paper we may leave it undefined; an implementation must choose how to say it, and the three languages of this lab choose differently:

- **OCaml** puts it in the type — `dequeue : 'a t -> ('a * 'a t) option` — and returns `None`;
- **Python** raises `IndexError`;
- **Java** throws `NoSuchElementException`.

Only the first is *checked*: an OCaml caller cannot forget the empty case, because the type will not let the value out until the case is handled. In the other two, nothing in the signature gives warning; the discovery is made at run time. Part E of the lab compares exactly this.

Exercise 5

Hand-execute, starting from the empty queue (`[]`, `[]`)*. Fill one row per operation. In the last column write* `yes` *if* `make` *performed a reversal, and how many elements it moved.*

operation	`front` after	`back` after	returned	reversal?
`enqueue 1`				
`enqueue 2`				
`enqueue 3`				
`dequeue`				
`dequeue`				
`enqueue 4`				
`dequeue`				
`dequeue`				

HOMEWORK from here on

Sections 5–7 and Exercises 6–9: the proofs and the cost argument. Nothing to hand in.

5. Two claims, and their proofs (homework)

Claim A — the invariant really holds

Claim. Every queue value a program can obtain satisfies INV.

Proof (informal, by induction on the number of operations). The empty queue (`[]`, `[]`) satisfies INV. For the step, observe that the *only* way to obtain a queue is as the result of `make`, so it suffices to show:

`make`(`f`, `b`) satisfies INV — for ***any*** *`f` and `b`, legal or not.*

Two cases, straight from the definition:

- `f` = `[]`. The result is (`reverse`(`b`), `[]`). Its back is `[]`, so INV holds whatever the front is.
- `f` ≠ `[]`. The result is (`f`, `b`) with non-empty front, so INV's premise is false and INV holds vacuously. □

That is the whole point of a smart constructor: the invariant is checked in *one* place, and every other line of the program inherits it for free.

Exercise 6

Suppose a student defines `enqueue`$(x, (\texttt{f}, \texttt{b})) = (\texttt{f},\ x :: \texttt{b})$ *— correct-looking, and it skips* `make`*. Which step of the proof above fails? Give a concrete two-operation sequence that reaches an illegal state, and say what goes wrong afterwards.*

Claim B — it really is a FIFO queue

Claim. $\texttt{contents}(\texttt{enqueue}(x, q)) = \texttt{contents}(q)\ \texttt{++}\ [x]$.

Proof. We use one fact about lists, worth checking on a three-element example: $\texttt{reverse}(x :: \texttt{b}) = \texttt{reverse}(\texttt{b})\ \texttt{++}\ [x]$. Let $q = (\texttt{f}, \texttt{b})$, legal by Claim A.

- $\texttt{f} \neq \texttt{[]}$: then `make` returns $(\texttt{f},\ x :: \texttt{b})$ and

$$\texttt{contents} = \texttt{f}\ \texttt{++}\ \texttt{reverse}(x :: \texttt{b}) = \texttt{f}\ \texttt{++}\ \texttt{reverse}(\texttt{b})\ \texttt{++}\ [x] = \texttt{contents}(q)\ \texttt{++}\ [x].\ \checkmark$$

- $\texttt{f} = \texttt{[]}$: by INV, $\texttt{b} = \texttt{[]}$ too, so q is the empty queue and $\texttt{contents}(q) = \emptyset$. Now $\texttt{make}(\texttt{[]}, [x]) = ([x], \texttt{[]})$, whose contents is $[x] = \emptyset\ \texttt{++}\ [x]$. ✓ □

Exercise 7

Do the `dequeue` *case yourself. Show: if* `contents`$(q) = x :: rest$ *then* `dequeue`(q) *returns* x*, and the new queue has contents* $rest$*. Two cases again — and state clearly where INV is used.*

6. Why the $O(n)$ reversal is affordable (homework)

`dequeue` usually costs one step, but sometimes triggers a reversal of the whole `back` list. That looks like $O(n)$. Here is why a *sequence* of operations is still cheap. Count one step per cons and one step per element moved by a reversal, and pay as follows:

> **The coin argument.** Charge every `enqueue` **two** coins: one pays for its cons immediately, and one is *placed on the element itself* and saved. When a reversal moves an element out of `back`, that element pays for its own move with the coin sitting on it.
>
> An element is put into `back` once, moved to `front` at most once, and never goes back. So no coin is ever spent twice, and n operations cost at most $2n$ coins: **amortized** $O(1)$.

Exercise 8

Return to the table of Exercise 5. Count the actual *steps (one per cons, one per element moved). Compare the running total with* $2\times$ *(number of operations so far). At which row is the gap between the two largest, and what is sitting in the structure at that moment?*

Optional, for the curious. That last sentence is a whole method in disguise. Put $\Phi(\texttt{f}, \texttt{b}) = |\texttt{b}|$ and define *amortized cost* = actual cost + $\Delta\Phi$. Then `enqueue` costs $1 + 1 = 2$; a `dequeue` without reversal costs $1 + 0 = 1$; and a `dequeue` that reverses k elements costs $(k + 1) - k = 1$. All constant. Coins are the *banker's method*, Φ is the *physicist's method*; both are set out in Okasaki [1, §5.1], and this queue is his §5.2.

7. The assumption behind the argument (homework)

The coin argument rests on one sentence that we never actually proved:

> *"An element is moved to* `front` *at most once, and never goes back."*

This holds if the queue is used in the way one normally imagines: one current version, each version dequeued once, older versions discarded. But our queue is **persistent**. Nothing forces a program to discard the old versions.

Exercise 9 (⋆)

Let q be a legal queue with `front` $= [a]$ *and* `back` $= [b_n, \ldots, b_1]$*, that is, n elements waiting in the back.*

(a) What does one `dequeue`(q) *cost, in steps?*

(b) Now evaluate `dequeue`(q) again — *on the same q, not on the queue it returned. What does the second call cost?*

(c) Do this k times. What is the total cost? What did the coin argument predict?

(d) In one sentence: which assumption does persistence break, and why?

Keep the answer to 9(d). Part D of the lab (`bench.py`) runs exactly this experiment, and shows an "amortized $O(1)$" operation costing $O(n)$ every single time.

The repair is not to give up persistence. It is to make the reversal happen *at most once ever*, by delaying it and remembering the result — lazy evaluation plus memoisation. We will have those tools in **session 5**.

Reference

[1] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998. Chapter 5, *Fundamentals of Amortization*: §5.1 gives the banker's and physicist's methods in general, §5.2 is this queue. Chapter 6 is the lazy repair promised above. The book grew out of the author's 1996 PhD thesis at Carnegie Mellon. A copy is in the course `material/` directory.