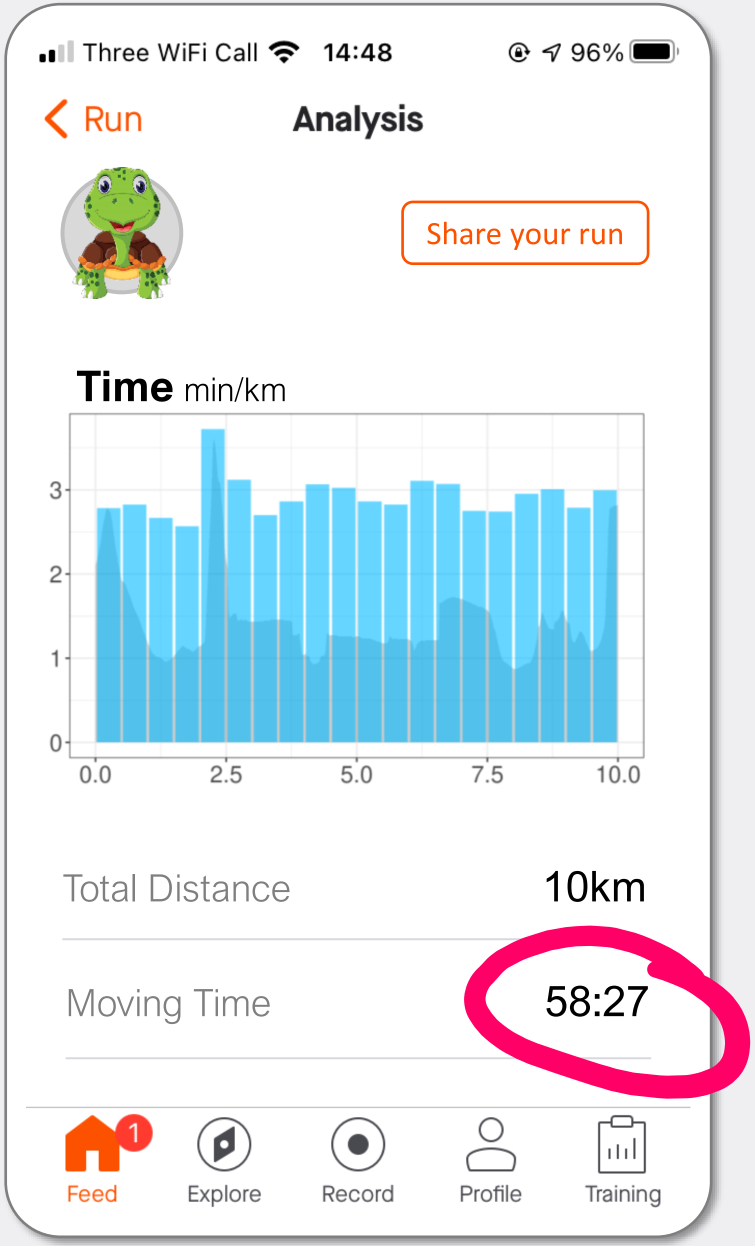
SECTION 7

Advanced data structures

SECTION 7.1

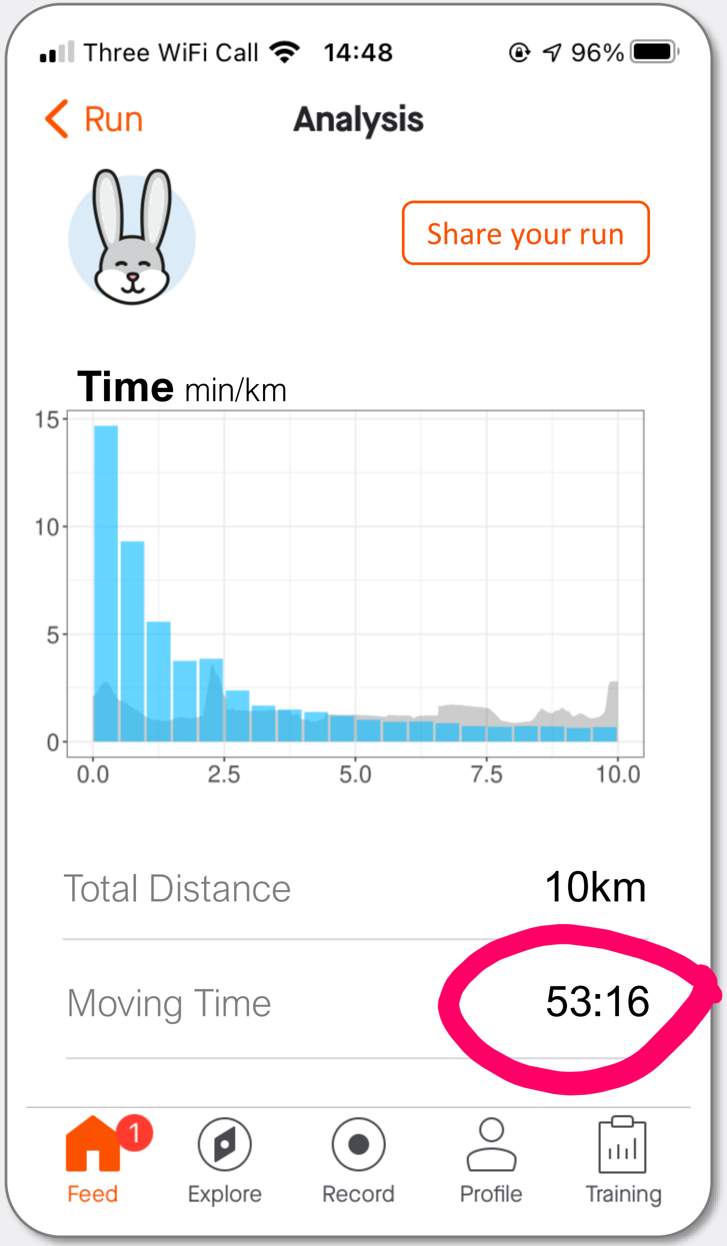
Aggregate analysis

I Raymond ſecit



Total Distance 10km

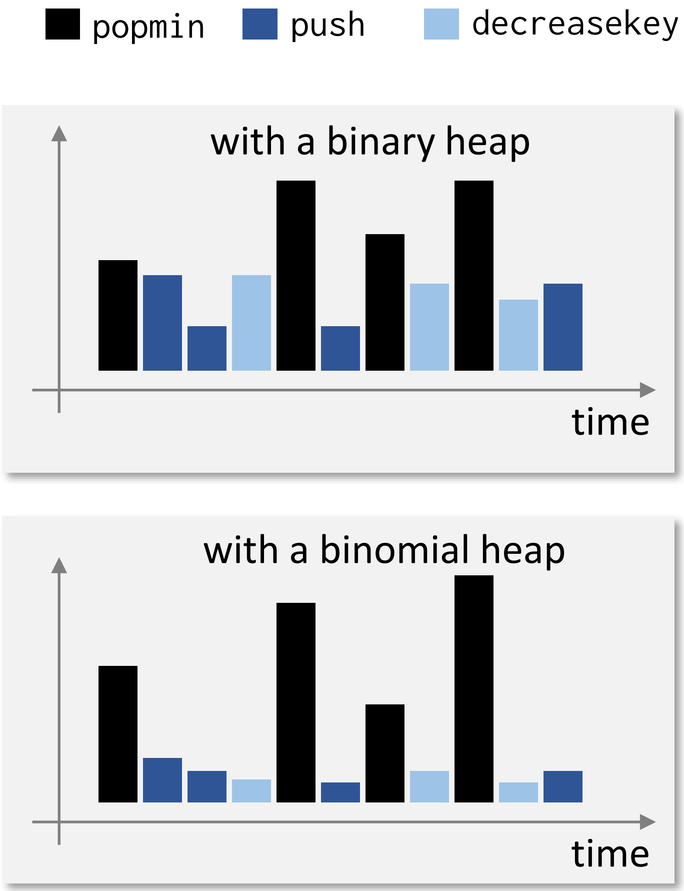
Moving Time 58:27



Total Distance 10km

Moving Time 53:16

Running time of each operation,
in a run of Dijkstra's algorithm



$$\text{total time} = O(V) \times c_{\text{popmin}} + O(E) \times c_{\text{push/dec.key}}$$

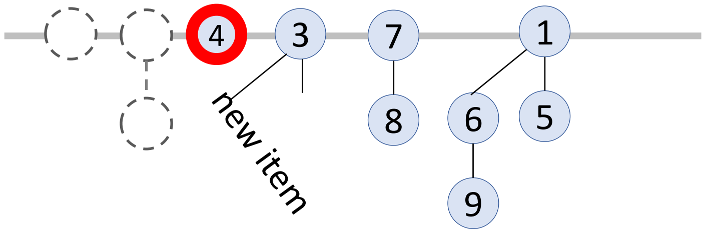
Don't worry about the worst-case cost of each individual operation.

Worry about the worst-case aggregate cost of a sequence of operations.

The worst case for a sequence of operations might not be as bad as the sum of per-op. worst cases.

(This is the hallmark of an advanced data structure.)

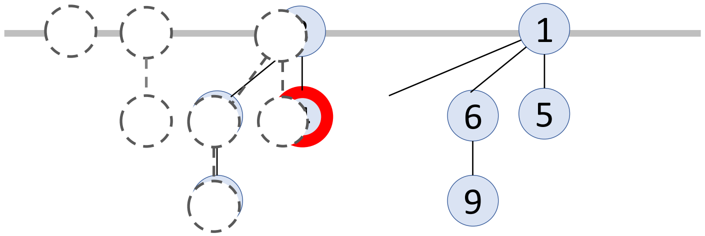
Adding an item
to a binomial heap



The worst case for a sequence of operations might not be as bad as the sum of per-op. worst cases.

(This is the hallmark of an advanced data structure.)

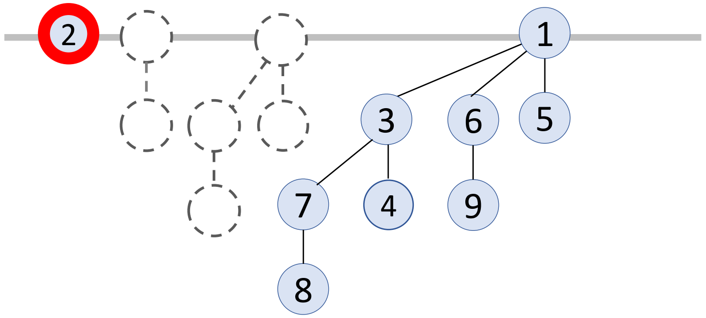
Adding an item
to a binomial heap



The worst case for a sequence of operations might not be as bad as the sum of per-op. worst cases.

(This is the hallmark of an advanced data structure.)

Adding a second item to a binomial heap



Worst-case cost of add is $O(\log n)$ n = # items in heap.

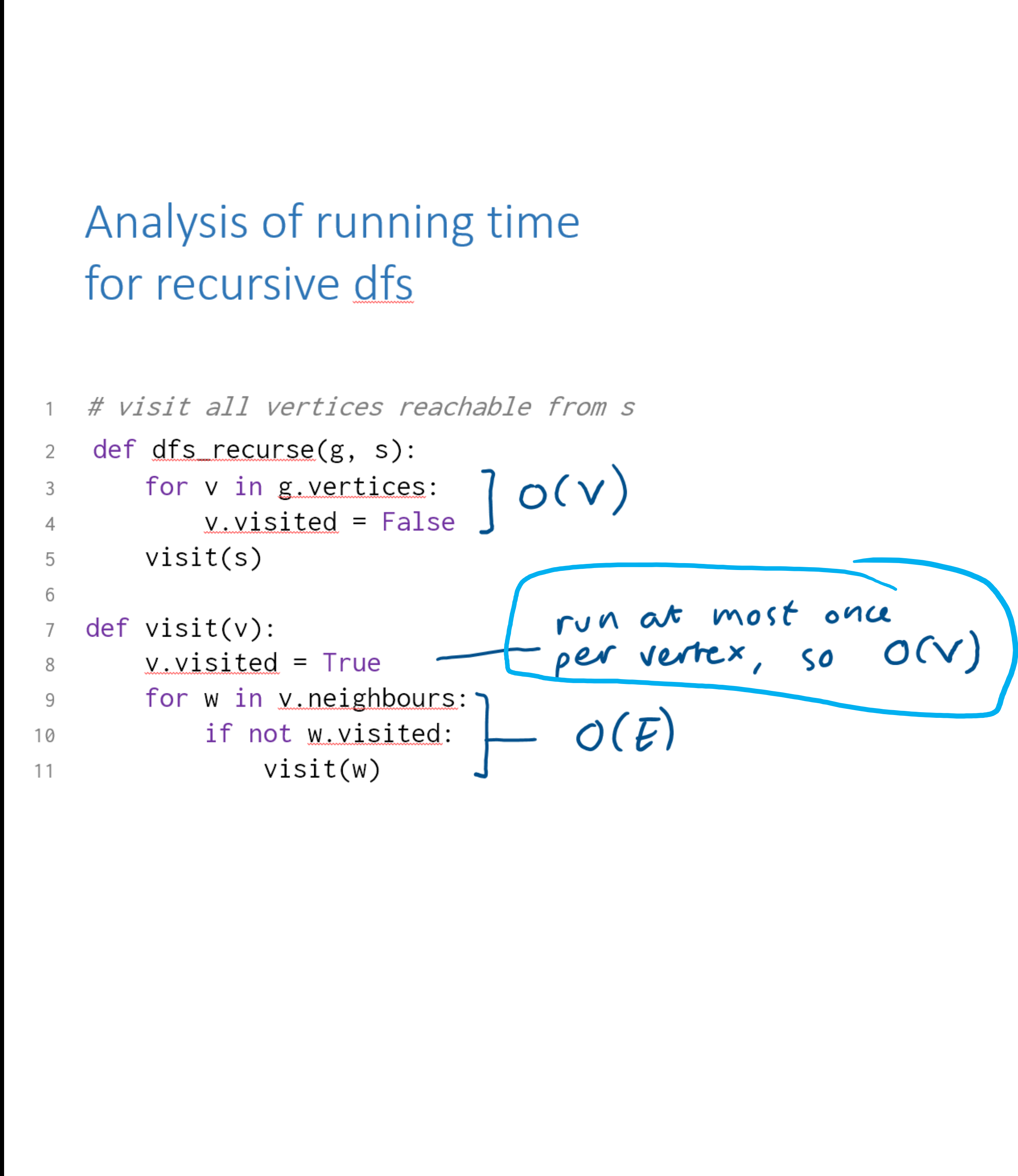
Worst-case cost of two adds is $O(1+\log n)$

The worst case for a sequence of operations might not be as bad as the sum of per-op. worst cases.

(This is the hallmark of an advanced data structure.)

How can we reason about aggregate costs?

- ❖ Just be clever and work hard
- ❖ Use an accounting trick called *amortized costs*



Analysis of running time for recursive dfs

```
# visit all vertices reachable from s
def dfs_recurse(g, s):
    for v in g.vertices:
        v.visited = False
    visit(s)

def visit(v):
    v.visited = True
    for w in v.neighbours:
        if not w.visited:
            visit(w)
```

$O(V)$

run at most once per vertex, so $O(V)$

$O(E)$

SECTION 7.2, 7.3

Amortized costs

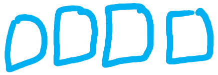
```
class MinList<T>:

    def append(T value):
        # append a new value

    def flush():
        # empty the list

    def foreach(f):
        # do f(x) for each item

    def T min():
        # return the smallest
        # (without removing it)
```



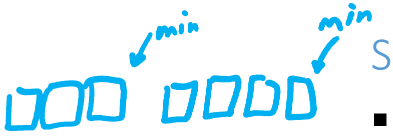
Stage 0

- Use a linked list
- `min` iterates over the entire list

append is O(1)
min is O(n)

Stage 1

- Use a linked list
- `min` caches its result, so that next time it only needs to iterate over newer values



append is O(1)
min is O(n)

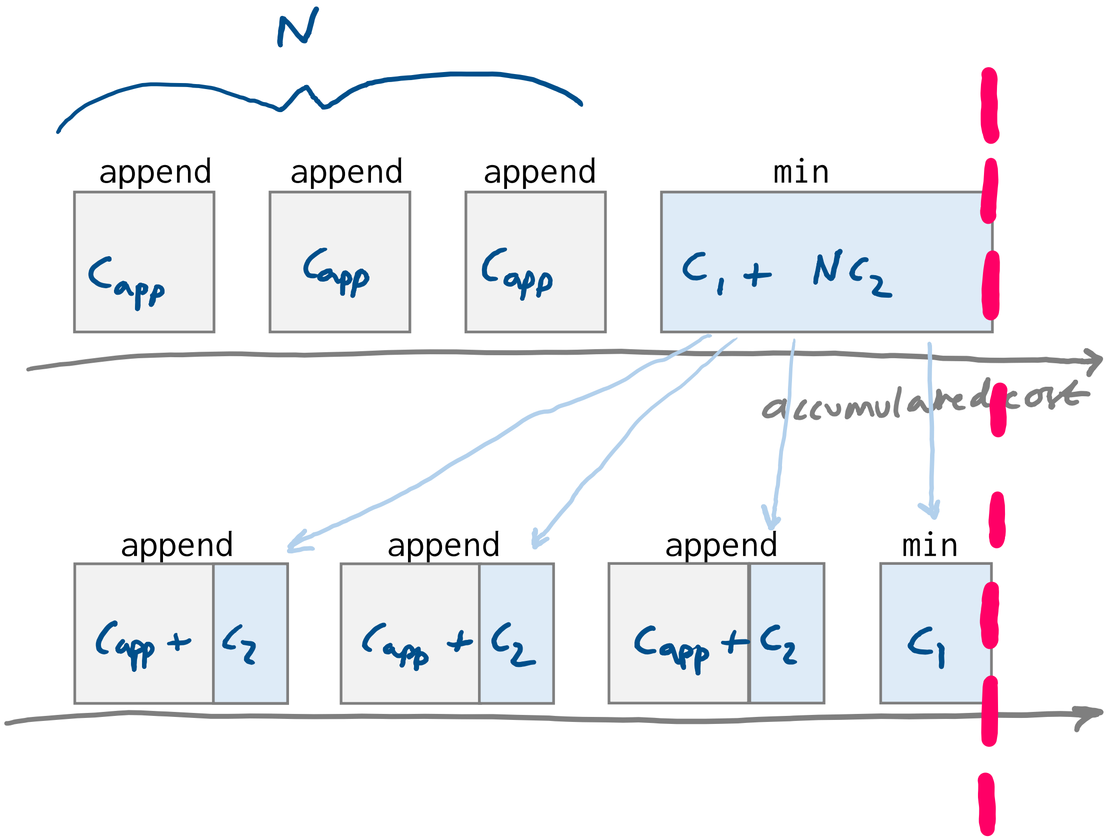
Stage 2

- Use a linked list
- Store the current minimum, and update it on every append

append is O(1)
min is O(1)

Stage 3

- `min` caches its result, the same as Stage 1
- ... but we argue it’s just as good as Stage 2



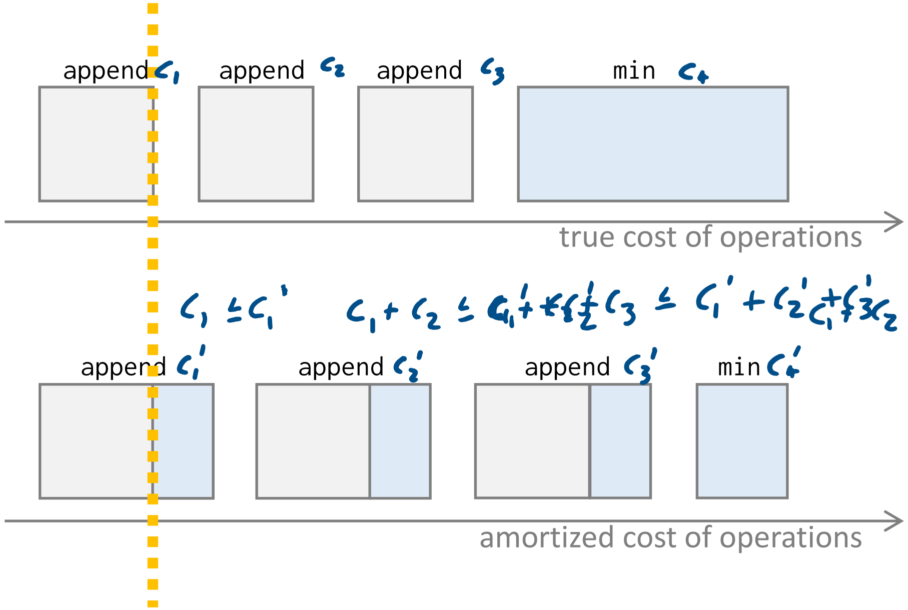
append append append min

append append append min

We get the same answer for aggregate cost whether we add true costs or "amortized" costs.

Stage 3

- `min` caches its result, the same as Stage 1
- ... but we argue it's just as good as Stage 2



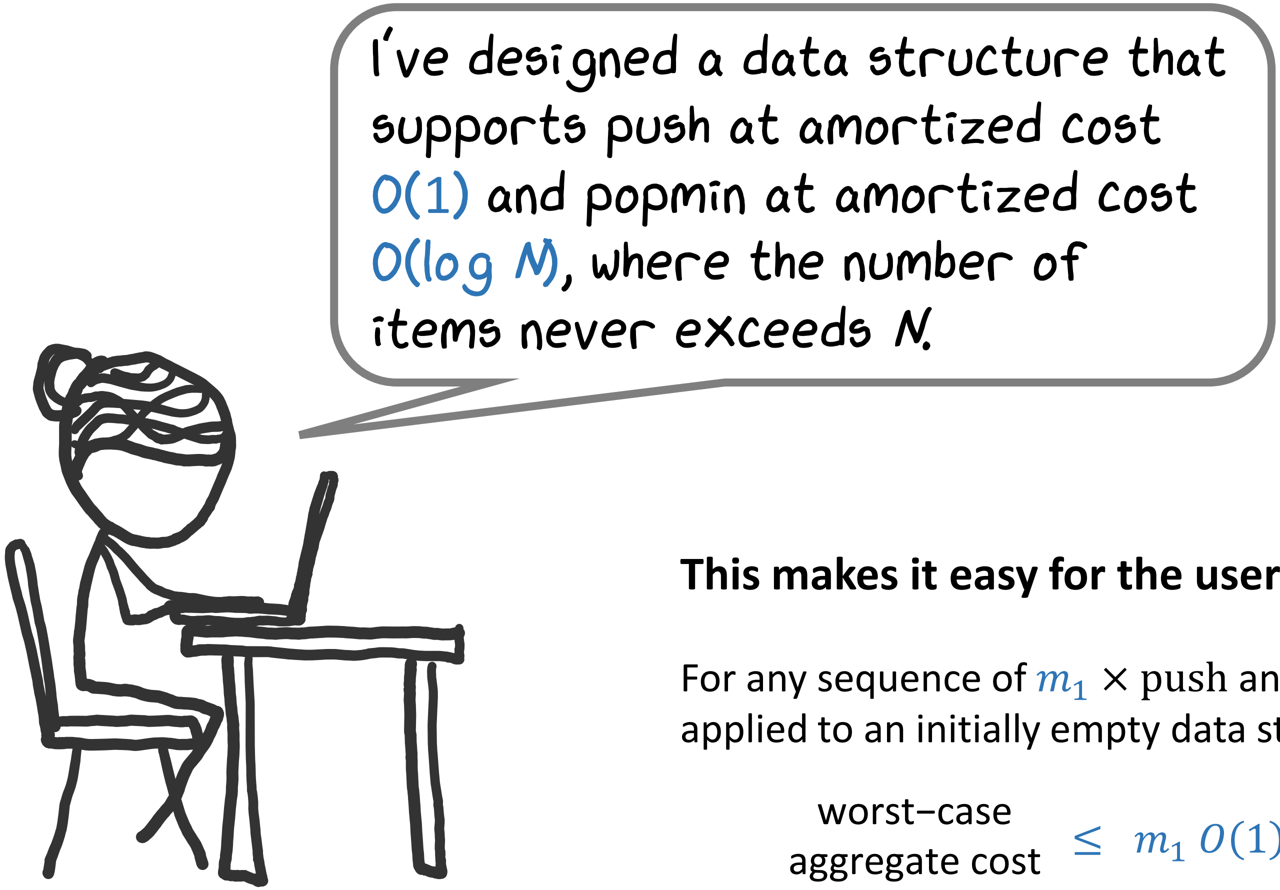
$c_1 \le c_1'$ $c_1 + c_2 \le c_1' + c_2'$ $c_1 + c_2 + c_3 \le c_1' + c_2' + c_3'$ $c_1 + c_2 + c_3 + c_4 \le c_1' + c_2' + c_3' + c_4'$

FUNDAMENTAL INEQUALITY OF AMORTIZATION

Let there be a sequence of m operations, applied to an initially-empty data structure, whose true costs are $c_1, c_2, \dots, c_m$. Suppose someone invents $c_1', c_2', \dots, c_m'$. These are called **amortized costs** if

$$c_1 + \cdots + c_j \le c_1' + \cdots + c_j' \quad \text{for all} \quad j \le m$$

aggregate true cost of a sequence of ops $\le$ agg. amortized cost of those operations] for ANY sequence of ops.


I've designed a data structure that supports push at amortized cost O(1) and popmin at amortized cost O(log N), where the number of items never exceeds N.

This makes it easy for the user to reason about aggregate costs.

For any sequence of $m_1 \times$ push and $m_2 \times$ popmin,
applied to an initially empty data structure,

$$\text{worst-case aggregate cost} \leq m_1\, O(1) + m_2\, O(\log N) = O(m_1 + m_2 \log N)$$

i.e. there exist N_0 and $\kappa > 0$ such that,
for any $N \geq N_0$, and for any sequence of of $m_1 \times$ push and $m_2 \times$ popmin
on a data structure that starts empty and always has $\leq N$ elements,

$$\text{worst-case aggregate cost} \leq \kappa(m_1 + m_2 \log N)$$

SECTION 7.4

How on earth are we meant to come up with useful amortized costs?

SECTION 7.5

Please review the Binary and Binomial heaps, before Wednesday’s lecture.