Amortized Analysis

Comp Sci 214, Fall 2022

The End is Near!

- Second midterm exam, Thu Dec 1st 11am, this room
  - Same format as first midterm, closed notes, no electronics
  - Not cumulative (but builds on earlier material)
  - Material until this Thursday is fair game
  - Will release a practice exam
- Final two lectures (Nov 22nd, 29th): cool, advanced topics!
  - Probabilitistic data structures
  - Persistent data structures
- Final project
  - Initial deadline next Tuesday
    - Highly recommend getting (at least!) `locate_all` working
    - Keep a backup of your last non-crashing version!
    - Submit that to get actual feedback
  - Second try and report due Tue Dec 6th (finals week)

Previously on 214

When we saw union-finds (WQUPC), we never said how much a single `union` or `find` costs

Instead, we said that m operations on n objects is $\mathcal{O}(m\ \alpha(n))$

Previously on 214

When we saw union-finds (WQUPC), we never said how much a single `union` or `find` costs

Instead, we said that *m* operations on *n* objects is $\mathcal{O}(m\ \alpha(n))$

This is because some long-running operations do *maintenance* that make other operations *faster*

- I.e., `find` doing path compression

This means we can get a more accurate (and cheaper!) cost analysis by considering operations in *aggregate*, rather than assuming the worst case for each.

Intuition: with *n* operations, they can't all be the worst!

Today

- Two data structures where worst case is too pessimistic.
- If we consider cost of operations in bulk, we can get better complexity bounds!
  $\rightarrow$ *Amortized analysis*

Dynamic Arrays

Recall: dynamic arrays

Briefly mentioned all the way back in lecture 2.

- Data structure that implements the sequence ADT.
- Elements stored in a vector with spare capacity.
- When inserting, use that spare capacity.
- When we run out of capacity, allocate a bigger vector.

Back then, I said growing by doubling was efficient.

- Now let's see why!
- (Same reasoning applies to hash tables too.)

Recall: dynamic arrays

Briefly mentioned all the way back in lecture 2.

- Data structure that implements the sequence ADT.
- Elements stored in a vector with spare capacity.
- When inserting, use that spare capacity.
- When we run out of capacity, allocate a bigger vector.

Back then, I said growing by doubling was efficient.

- Now let's see why!
- (Same reasoning applies to hash tables too.)

But first, let's see why growing by just one element is inefficient.

A naïve implementation (1/2)

```
class BadDynArray[T]:
    let data: VecC[T]

    def __init__(self):
        self.data = []

    def len(self):
        return self.data.len()

    def get(self, index):
        return self.data[index]

    def set(self, index, element):
        self.data[index] = element

    …
```

Store data in an array.
Keep no spare capacity.
Most ops just call array ops.

A naïve implementation (2/2)

Grow by 1.
Create a new vector.
Copy elements over.

```
class BadDynArray[T]:
    …

    def push_back(self, element):
        let new_data = [ None; self.len() + 1 ]
        for i, v in self.data:
            new_data[i] = v
        new_data[self.len()] = element
        self.data = new_data
```

Naïve representation complexities

- `get`/`set`/`len` are $\mathcal{O}(1)$
- `push_back` is $\mathcal{O}(n)$!

QUIZ: How long to build an n–element array, starting with an empty array and doing n `pushes`?

A: n **B**: n^2 **C**: n^3

Naïve representation complexities

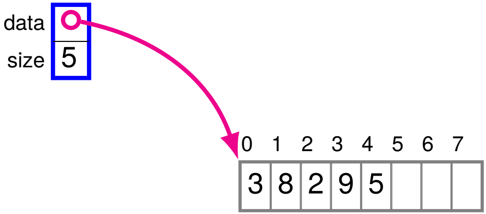
- `get`/`set`/`len` are $\mathcal{O}(1)$
- `push_back` is $\mathcal{O}(n)$!

QUIZ: How long to build an n–element array, starting with an empty array and doing n `pushes`?

A: n **B**: n^2 **C**: n^3

$$\sum_{i=1}^{n} \mathcal{O}(i) = \mathcal{O}(n^2)$$

A better idea: leave extra space in the array



\# elements in dynamic array $\neq$ vector length!
$\rightarrow$ store size separately.

When we add an element, there may be spare capacity.
$\rightarrow$ don't need to grow every time!

When we grow, grow enough to keep some spare capacity.
$\rightarrow$ double when we need to grow.

Implementation (1/3)

```
class DynArray[T]:
    let data: VecC[OrC(T, NoneC)] # could be unused space
    let size: nat?

    def __init__(self, initial_capacity: nat?):
        self.data = [None; initial_capacity]
        self.size = 0

    def len(self):
        return self.size

    def capacity(self) -> nat?:
        return self.data.len()

    …
```

Not much to see here.

Implementation (2/3)

```
class DynArray[T]:
    …

    def get(self, index):
        self._bounds_check(index)
        return self.data[index]

    def set(self, index, element):
        self._bounds_check(index)
        self.data[index] = element

    def _bounds_check(self, index):
        if index >= self.size:
            error('DynArray: out of bounds')

    …
```

Need to check that we’re not accessing unused spots!

Implementation (3/3)

```
class DynArray[T]:
    …

    def push_back(self, element):
        self._ensure_capacity(self.size + 1)
        self.data[self.size] = element
        self.size = self.size + 1

    def _ensure_capacity(self, cap):
        if self.capacity() < cap:
            cap = max(cap, 2 * self.capacity())
            let new_data = [ None; cap ]
            for i, v in self.data:
                new_data[i] = v
            self.data = new_data
```

Double when we need to grow.

Time complexities

- `get`/`set`/`len` are $\mathcal{O}(1)$
- `push_back` is still $\mathcal{O}(n)$ in the worst case!

QUIZ: How long to build an n–element array, starting with an empty array and doing n `push`es?

A: n **B**: n^2 **C**: n^3 **D**: It's complicated

Time complexities

- `get`/`set`/`len` are $\mathcal{O}(1)$
- `push_back` is still $\mathcal{O}(n)$ in the worst case!

QUIZ: How long to build an n–element array, starting with an empty array and doing n `pushes`?

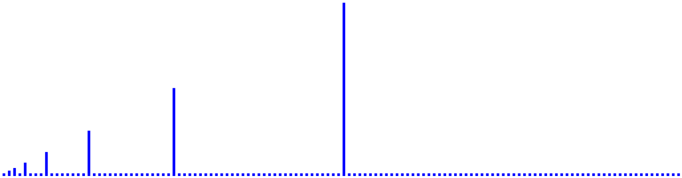
A: n **B**: n^2 **C**: n^3 **D**: It's complicated

$$\sum_{i=0}^{n} \mathcal{O}(i) = \mathcal{O}(n^2)?$$

That's if all pushes are the worst case. Will they be?

The peculiar thing about `push_back`

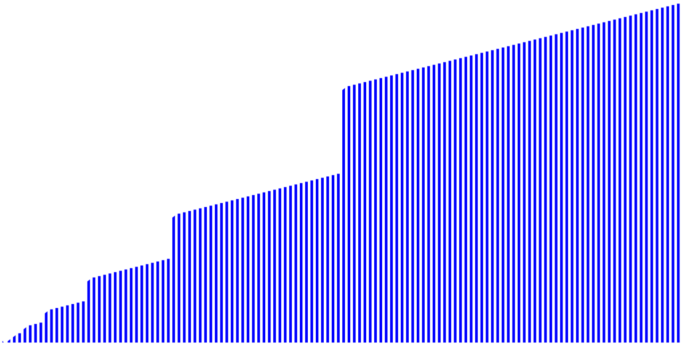
- Most of the time it's cheap
- But *occasionally*, we need to grow (which is expensive)



X axis: push #

Y axis: time it takes to do push #*x*

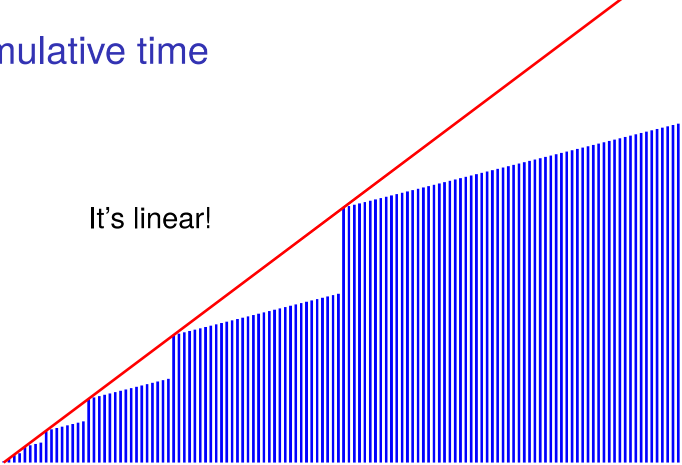
Cumulative time



X axis: push #

Y axis: *cumulative* time it takes to do pushes $0 \ldots x$

Cumulative time



X axis: push #

Y axis: *cumulative* time it takes to do pushes $0 \ldots x$

Wait, what?

- `push_back` has two “modes”
  - There is spare capacity: set array element, $\mathcal{O}(1)$
  - There is no spare capacity: double the size, $\mathcal{O}(n)$
  - As we go from 0 to n elements, mix of both modes
- When we double, get space for **many** cheap pushes!
  - If we currently have m elements, get m cheap pushes
  - Doublings get less and less frequent as array grows
- Now let’s work out the math of this

Dynamic array aggregate analysis

Suppose we create a new array and push n times. How can we show linear total time?

Dynamic array aggregate analysis

Suppose we create a new array and push n times. How can we show linear total time?

Let s_i be the size of the array at step i.

Let c_i be the cost of the ith push:

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is a power of 2 (i.e., we're full)} \\ 1 & \text{otherwise (i.e., we have spare capacity)} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
s_i	1	2	4	4	8	8	8	8	16	16	16	16	16	16	16	16	32
c_i	**1**	**2**	**3**	1	**5**	1	1	1	**9**	1	1	1	1	1	1	1	**17**

Adding it up

Let $d_i = c_i - 1$ (the cost of doubling, when we do double)

Then,

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} (1 + d_i)$$

Adding it up

Let $d_i = c_i - 1$ (the cost of doubling, when we do double)

Then,

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} (1 + d_i)$$

$$= n + \sum_{i=1}^{n} d_i$$

d_i is almost always 0.

Adding it up

Let $d_i = c_i - 1$ (the cost of doubling, when we do double)

Then,

$$\begin{aligned}
\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n}(1 + d_i) \\
&= n + \sum_{i=1}^{n} d_i && d_i \text{ is almost always } 0. \\
&= n + \sum_{i=0}^{\log_2 n} 2^i && \text{We double } \log_2 n \text{ times.}
\end{aligned}$$

Adding it up

Let $d_i = c_i - 1$ (the cost of doubling, when we do double)

Then,

$$\begin{aligned}
\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} (1 + d_i) \\
&= n + \sum_{i=1}^{n} d_i && d_i \text{ is almost always 0.} \\
&= n + \sum_{i=0}^{\log_2 n} 2^i && \text{We double } \log_2 n \text{ times.} \\
&= n + (1 + 2 + 4 + \ldots + \tfrac{n}{4} + \tfrac{n}{2} + n)
\end{aligned}$$

Adding it up

Let $d_i = c_i - 1$ (the cost of doubling, when we do double)

Then,

$$\begin{aligned}
\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} (1 + d_i) \\
&= n + \sum_{i=1}^{n} d_i && d_i \text{ is almost always 0.} \\
&= n + \sum_{i=0}^{\log_2 n} 2^i && \text{We double } \log_2 n \text{ times.} \\
&= n + (1 + 2 + 4 + \ldots + \tfrac{n}{4} + \tfrac{n}{2} + n) \\
&< 3n
\end{aligned}$$

Time complexities

- `get`/`set`/`size` are $\mathcal{O}(1)$
- `push_back` has $\mathcal{O}(1)$ *amortized cost*
  - If you do n of them, the *total* cost is $\mathcal{O}(n)$

Stop! Question time!

Before we move on to our second data structure...

Anything unclear so far?

Anything I missed?

Aside: Do we *have* to double?

- Doubling each time is efficient
- Adding 1 is not
- What about other options?

Aside: Do we *have* to double?

- Doubling each time is efficient
- Adding 1 is not
- What about other options?

- Any *multiplicative* growth works!
  - Tripling each time
  - Growing by 1.5$\times$
  - Even growing by just 1.0001$\times$!
  - All amortized $\mathcal{O}(1)$; just different constant factors
  - In practice, 1.5$\times$ is a common choice

- Adding a constant amount of space doesn't work
  - Not even adding 1,000,000 each time we grow!
  - That only gives us a *constant* number of cheap pushes to amortize a $\mathcal{O}(n)$ operation over!

Banker’s Queues

Banker's queues

- Another data structure that implements the queue ADT
- Will have queue operations with $\mathcal{O}(1)$ amortized cost
- Unlike other queues we've seen, easy to make *persistent*
  - More on persistent data structures in a later lecture

Banker's queues

- Another data structure that implements the queue ADT
- Will have queue operations with $\mathcal{O}(1)$ amortized cost
- Unlike other queues we've seen, easy to make *persistent*
  - More on persistent data structures in a later lecture

Key idea: use two stacks, *front* and *back*

- Enqueue into *back*, dequeue from *front*
- When *front* is empty, tranfer contents of *back* into it
- The elements of the queue go from top of front to bottom of front, then bottom of back to top of back

Banker's queue example

$\boxed{\langle\, 3, 4, 5 \,\langle}$ `{front:` $\boxed{\,|\, 3 \,\rangle}$`,` `back:` $\boxed{\,|\, 4, 5 \,\rangle}$`}`

Banker's queue example

$\boxed{\langle\, 3, 4, 5 \,\langle}$ {front: $\boxed{\,|\, 3 \,\rangle}$, back: $\boxed{\,|\, 4, 5 \,\rangle}$}

`q.dequeue()` $\Rightarrow$?

Banker's queue example

$\boxed{\langle\, 3, 4, 5 \,\langle}$ `{front:` $\boxed{|\, 3 \,\rangle}$, `back:` $\boxed{|\, 4, 5 \,\rangle}$`}`

`q.dequeue()` $\Rightarrow$ 3

$\boxed{\langle\, 4, 5 \,\langle}$ `{front:` $\boxed{|\, \rangle}$, `back:` $\boxed{|\, 4, 5 \,\rangle}$`}`

Banker's queue example

$\boxed{\langle 3, 4, 5 \langle}$ `{front:` $\boxed{| 3 \rangle}$`,` `back:` $\boxed{| 4, 5 \rangle}$`}`

`q.dequeue()` $\Rightarrow$ 3

$\boxed{\langle 4, 5 \langle}$ `{front:` $\boxed{| \rangle}$`,` `back:` $\boxed{| 4, 5 \rangle}$`}`

`q.enqueue(6)`

Banker's queue example

$\boxed{\langle 3, 4, 5 \langle}$ `{front:` $\boxed{| 3 \rangle}$, `back:` $\boxed{| 4, 5 \rangle}$`}`

`q.dequeue()` $\Rightarrow$ 3

$\boxed{\langle 4, 5 \langle}$ `{front:` $\boxed{| \ \rangle}$, `back:` $\boxed{| 4, 5 \rangle}$`}`

`q.enqueue(6)`

$\boxed{\langle 4, 5, 6 \langle}$ `{front:` $\boxed{| \ \rangle}$, `back:` $\boxed{| 4, 5, 6 \rangle}$`}`

Banker's queue example

$\boxed{\langle\, 3, 4, 5 \,\langle}$ `{front:` $\boxed{|\, 3 \,\rangle}$, `back:` $\boxed{|\, 4, 5 \,\rangle}$`}`

`q.dequeue()` $\Rightarrow$ 3

$\boxed{\langle\, 4, 5 \,\langle}$ `{front:` $\boxed{|\, \rangle}$, `back:` $\boxed{|\, 4, 5 \,\rangle}$`}`

`q.enqueue(6)`

$\boxed{\langle\, 4, 5, 6 \,\langle}$ `{front:` $\boxed{|\, \rangle}$, `back:` $\boxed{|\, 4, 5, 6 \,\rangle}$`}`

`q.dequeue()` $\Rightarrow$?

Banker's queue example

$\boxed{\langle\, 3, 4, 5 \,\langle}$ {front: $\boxed{|\, 3 \,\rangle}$, back: $\boxed{|\, 4, 5 \,\rangle}$}

q.dequeue() $\Rightarrow$ 3

$\boxed{\langle\, 4, 5 \,\langle}$ {front: $\boxed{|\, \rangle}$, back: $\boxed{|\, 4, 5 \,\rangle}$}

q.enqueue(6)

$\boxed{\langle\, 4, 5, 6 \,\langle}$ {front: $\boxed{|\, \rangle}$, back: $\boxed{|\, 4, 5, 6 \,\rangle}$}

q.dequeue() $\Rightarrow$?

$\boxed{\langle\, 4, 5, 6 \,\langle}$ {front: $\boxed{|\, \rangle}$, back: $\boxed{|\, 4, 5, 6 \,\rangle}$}

Banker's queue example

$\boxed{\langle\, 3, 4, 5 \,\langle}$ {front: $\boxed{\,|\, 3 \,\rangle}$, back: $\boxed{\,|\, 4, 5 \,\rangle}$}

q.dequeue() $\Rightarrow$ 3

$\boxed{\langle\, 4, 5 \,\langle}$ {front: $\boxed{\,|\, \rangle}$, back: $\boxed{\,|\, 4, 5 \,\rangle}$}

q.enqueue(6)

$\boxed{\langle\, 4, 5, 6 \,\langle}$ {front: $\boxed{\,|\, \rangle}$, back: $\boxed{\,|\, 4, 5, 6 \,\rangle}$}

q.dequeue() $\Rightarrow$?

$\boxed{\langle\, 4, 5, 6 \,\langle}$ {front: $\boxed{\,|\, 6 \,\rangle}$, back: $\boxed{\,|\, 4, 5 \,\rangle}$}

Banker's queue example

$\boxed{\langle 3, 4, 5 \langle}$ `{front:` $\boxed{| 3 \rangle}$, `back:` $\boxed{| 4, 5 \rangle}$`}`

`q.dequeue()` $\Rightarrow$ 3

$\boxed{\langle 4, 5 \langle}$ `{front:` $\boxed{| \; \rangle}$, `back:` $\boxed{| 4, 5 \rangle}$`}`

`q.enqueue(6)`

$\boxed{\langle 4, 5, 6 \langle}$ `{front:` $\boxed{| \; \rangle}$, `back:` $\boxed{| 4, 5, 6 \rangle}$`}`

`q.dequeue()` $\Rightarrow$?

$\boxed{\langle 4, 5, 6 \langle}$ `{front:` $\boxed{| 6, 5 \rangle}$, `back:` $\boxed{| 4 \rangle}$`}`

Banker's queue example

$\boxed{\langle\, 3, 4, 5 \,\langle}$ `{front:` $\boxed{|\, 3 \,\rangle}$, `back:` $\boxed{|\, 4, 5 \,\rangle}$}

`q.dequeue()` $\Rightarrow$ 3

$\boxed{\langle\, 4, 5 \,\langle}$ `{front:` $\boxed{|\, \rangle}$, `back:` $\boxed{|\, 4, 5 \,\rangle}$}

`q.enqueue(6)`

$\boxed{\langle\, 4, 5, 6 \,\langle}$ `{front:` $\boxed{|\, \rangle}$, `back:` $\boxed{|\, 4, 5, 6 \,\rangle}$}

`q.dequeue()` $\Rightarrow$?

$\boxed{\langle\, 4, 5, 6 \,\langle}$ `{front:` $\boxed{|\, 6, 5, 4 \,\rangle}$, `back:` $\boxed{|\, \rangle}$}

Banker's queue example

$\boxed{\langle 3, 4, 5 \langle}$ `{front:` $\boxed{|\, 3 \rangle}$, `back:` $\boxed{|\, 4, 5 \rangle}$`}`

`q.dequeue()` $\Rightarrow$ 3

$\boxed{\langle 4, 5 \langle}$ `{front:` $\boxed{|\, \rangle}$, `back:` $\boxed{|\, 4, 5 \rangle}$`}`

`q.enqueue(6)`

$\boxed{\langle 4, 5, 6 \langle}$ `{front:` $\boxed{|\, \rangle}$, `back:` $\boxed{|\, 4, 5, 6 \rangle}$`}`

`q.dequeue()` $\Rightarrow$ 4

$\boxed{\langle 4, 5, 6 \langle}$ `{front:` $\boxed{|\, 6, 5, 4 \rangle}$, `back:` $\boxed{|\, \rangle}$`}`

$\boxed{\langle 5, 6 \langle}$ `{front:` $\boxed{|\, 6, 5 \rangle}$, `back:` $\boxed{|\, \rangle}$`}`

Banker's queue example

$\boxed{\langle\, 3, 4, 5 \,\langle}$ {front: $\boxed{|\, 3 \,\rangle}$, back: $\boxed{|\, 4, 5 \,\rangle}$}

q.dequeue() $\Rightarrow$ 3

$\boxed{\langle\, 4, 5 \,\langle}$ {front: $\boxed{|\,\rangle}$, back: $\boxed{|\, 4, 5 \,\rangle}$}

q.enqueue(6)

$\boxed{\langle\, 4, 5, 6 \,\langle}$ {front: $\boxed{|\,\rangle}$, back: $\boxed{|\, 4, 5, 6 \,\rangle}$}

q.dequeue() $\Rightarrow$ 4

$\boxed{\langle\, 4, 5, 6 \,\langle}$ {front: $\boxed{|\, 6, 5, 4 \,\rangle}$, back: $\boxed{|\,\rangle}$}

$\boxed{\langle\, 5, 6 \,\langle}$ {front: $\boxed{|\, 6, 5 \,\rangle}$, back: $\boxed{|\,\rangle}$}

q.enqueue(7)

Banker's queue example

$\boxed{\langle\, 3, 4, 5 \,\langle}$	`{front:` $\boxed{\,|\, 3 \,\rangle}$,	`back:` $\boxed{\,|\, 4, 5 \,\rangle}$`}`
`q.dequeue()` $\Rightarrow$ 3		
$\boxed{\langle\, 4, 5 \,\langle}$	`{front:` $\boxed{\,|\, \rangle}$,	`back:` $\boxed{\,|\, 4, 5 \,\rangle}$`}`
`q.enqueue(6)`		
$\boxed{\langle\, 4, 5, 6 \,\langle}$	`{front:` $\boxed{\,|\, \rangle}$,	`back:` $\boxed{\,|\, 4, 5, 6 \,\rangle}$`}`
`q.dequeue()` $\Rightarrow$ 4		
$\boxed{\langle\, 4, 5, 6 \,\langle}$	`{front:` $\boxed{\,|\, 6, 5, 4 \,\rangle}$,	`back:` $\boxed{\,|\, \rangle}$`}`
$\boxed{\langle\, 5, 6 \,\langle}$	`{front:` $\boxed{\,|\, 6, 5 \,\rangle}$,	`back:` $\boxed{\,|\, \rangle}$`}`
`q.enqueue(7)`		
$\boxed{\langle\, 5, 6, 7 \,\langle}$	`{front:` $\boxed{\,|\, 6, 5 \,\rangle}$,	`back:` $\boxed{\,|\, 7 \,\rangle}$`}`

Banker’s queue implementation (1/2)

```
class BankersQueue[T] (QUEUE):
    let front
    let back
    # Interpretation: the queue is the elements of
    # `front` in pop order followed by `back` in reverse

    def __init__(self):
        self.front = Stack()
        self.back = Stack()

    def len(self):
        return self.front.len() + self.back.len()

    def empty?(self):
        return self.front.empty?() and self.back.empty?()

    …
```

Banker’s queue implementation (2/2)

```
class BankersQueue[T] (QUEUE):
    …

    def enqueue(self, element: T) -> NoneC:
        self.back.push(element)
```

Banker's queue implementation (2/2)

```
class BankersQueue[T] (QUEUE):
    …

    def enqueue(self, element: T) -> NoneC:
        self.back.push(element)

    def dequeue(self) -> T:
        # refill `front` if needed
        if self.front.empty?():
            if self.back.empty?():
                error('BankersQueue.dequeue: empty')
            while not self.back.empty?():
                # reverses order! top of `back` will
                # end up on bottom of `front`
                self.front.push(self.back.pop())
        return self.front.pop()
```

We have both fast case and slow case $\rightarrow$ amortized analysis!

Stop! Question time!

Before we move on to the analysis...

Anything unclear so far?
Anything I missed?

Banker's queue analysis

We assign a "balance" (as in, bank account balance) to each data structure state:

$$B(q) = q\texttt{.back.len()}$$

Operations may deposit to or withdraw from the balance.

Balance of a new queue is 0, and balance is never negative.

Banker's queue analysis

We assign a "balance" (as in, bank account balance) to each data structure state:

$$B(q) = q.\texttt{back.len()}$$

Operations may deposit to or withdraw from the balance.

Balance of a new queue is 0, and balance is never negative.

The amortized cost of an operation will be

$$c + B(q') - B(q)$$

where c is the actual cost of the operation,
q is the state before the operation, and q' is after.
Balance change is $+$ for deposits, $-$ for withdrawals

Key idea: some operations will spend savings to get an amortized cost lower than their actual cost.

Actual costs

Actual cost of enqueue: 1

Actual costs

Actual cost of enqueue: 1

Actual cost of cheap dequeue (front isn't empty): 1

Actual costs

Actual cost of enqueue: 1

Actual cost of cheap dequeue (front isn't empty): 1

Actual cost of expensive dequeue (with reversal):

- Cost of the reversal (the number of elements reversed): n
- Plus the cost of a cheap dequeue: 1
- Total: $n + 1$

Amortized cost of enqueue

- Actual cost of enqueue is 1
- Increases the length of the back by 1, hence $B(q') - B(q) = 1$
  - I.e., *deposit* of 1

So amortized cost is $1 + 1 = 2$. That's $\mathcal{O}(1)$.

Amortized cost of cheap dequeue

- Actual cost of cheap dequeue is 1
- No change in balance; no change to `back`

So amortized cost is 1. Also $\mathcal{O}(1)$.

Amortized cost of expensive dequeue

Let n be `q.back.len()`, the length of the back stack. Then:

- Actual cost is $n + 1$
- $B(q) = n$ (before reversal)
- $B(q') = 0$ (after reversal)
- *Withdrawal* of n

So amortized cost is $(n + 1) + (0 - n) = 1$.

Look at that! $\mathcal{O}(1)$!

Wait, what just happened?

operation	single operation*	amortized
enqueue	$\mathcal{O}(1)$	$\mathcal{O}(1)$
dequeue	$\mathcal{O}(n)$	$\mathcal{O}(1)$

We found a way (*B*) to account for cost across operations.

- No fraud, though! All costs are accounted for somewhere.
- We're just shifting costs from expensive ops to cheap ones.

* worst-case

Wait, what just happened?

operation	single operation*	amortized
enqueue	$\mathcal{O}(1)$	$\mathcal{O}(1)$
dequeue	$\mathcal{O}(n)$	$\mathcal{O}(1)$

We found a way (B) to account for cost across operations.

- No fraud, though! All costs are accounted for somewhere.
- We're just shifting costs from expensive ops to cheap ones.

Insight: dequeue can only be expensive if we did a lot of (cheap) enqueues before!

We're "saving up" time on cheap operations, and "spending" it on the expensive ones. So everything works out constant.

* worst-case

Amortized data structures

data structure	operation	single operation*	amortized
dynamic array	`push_back`	$\mathcal{O}(n)$	$\mathcal{O}(1)$
banker's queue	`enqueue`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
	`dequeue`	$\mathcal{O}(n)$	$\mathcal{O}(1)$
WQUPC	`union`	$\mathcal{O}(\log n)$	$\mathcal{O}(\alpha(n))^{**}$
	`find`	$\mathcal{O}(\log n)$	$\mathcal{O}(\alpha(n))^{**}$

* worst-case

** AKA "I can't believe it's not constant!"

If we have time: playing with our dynamic array and/or banker’s queue

Next time: The relational model