Amortized Time

EECS 214, Fall 2017

Last time

We never said how much a single `union` or `find` operation costs

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

Last time

We never said how much a single `union` or `find` operation costs

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

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

Example: dynamic array

Dynamic Array ADT

Looks like: [3, 8, 2, 90, 5]

Signature:

- *get*(DynArray, Index): Element
- *set*(DynArray, Index, Element): Void
- *push*(DynArray, Element): Void
- *pop*(DynArray): Element
- *size*(DynArray): Natural

Laws:

- $\{a = [v_0, \ldots, v_k]\}\ get(a, i) = v_i$
- $\{a = [v_0, \ldots, v_k]\}\ set(a, i, v)\ \{a = [v_0, \ldots v_{i-1}, v, v_{i+1}, \ldots v_k]\}$
- $\{a = [v_0, \ldots, v_k]\}\ push(a, v)\ \{a = [v_0, \ldots, v_k, v]\}$
- $\{a = [v_0, \ldots, v_k]\}\ pop(a) = v_k\ \{a = [v_0, \ldots, v_{k-1}]\}$
- $\{a = [v_0, \ldots, v_k]\}\ size(a) = k + 1$

A naïve representation (1/2)

```
# A DynArrayOf<X> is dyn-array(VectorOf<X>)
defstruct dyn_array(data)
# Interpretation: the elements of `data` are the
# elements of the dynamic array

def da_get(a, i):
    a.data[i]

def da_set!(a, i, v):
    a.data[i] = v

def da_size(a):
    len(a.data)
```

A naïve representation (2/2)

```
def da_push!(a, v):
    def get_elem(i):
        if i < len(a.data): a.data[i]
        else: v
    a.data = [ get_elem(i) for i in len(a.data) + 1 ]

def da_pop!(a):
    let result = a.data[len(a.data) - 1]
    a.data = [ a.data[i] for i in len(a.data) - 1 ]
    result
```

Naïve representation complexities

- *get*/*set*/*size* are $\mathcal{O}(1)$
- *push*/*pop* are $\mathcal{O}(n)$!

Naïve representation complexities

- *get*/*set*/*size* are $\mathcal{O}(1)$
- *push*/*pop* are $\mathcal{O}(n)$!

How long does it take to build an n–element array by *push*es?

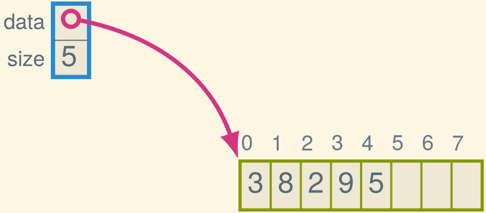
Naïve representation complexities

- *get*/*set*/*size* are $\mathcal{O}(1)$
- *push*/*pop* are $\mathcal{O}(n)$!

How long does it take to build an n–element array by *push*es?

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

A better idea: leave extra space in the array



Implementation (1/4)

```
# A DynArrayOf<X> is dyn_array(VectorOf<X>, Natural)
defstruct dyn_array(data, size)
# Interpretation: the first `size` elements of `data`
# are the elements of the array

def da_new(capacity): dyn_array([False; capacity], 0)

def da_size(a): a.size

def da_capacity(a): len(a.data)
```

Implementation (2/4)

```
def da_get(a, i):
    da_bounds_check!(a, i)
    a.data[i]

def da_set!(a, i, v):
    da_bounds_check!(a, i)
    a.data[i] = v
```

Implementation (2/4)

```
def da_get(a, i):
    da_bounds_check!(a, i)
    a.data[i]

def da_set!(a, i, v):
    da_bounds_check!(a, i)
    a.data[i] = v

def da_bounds_check!(a, i):
    if i >= a.size:
        error('dyn_array: out of bounds')
```

Implementation (3/4)

```
def da_pop!(a):
  a.size = a.size - 1
  let result = a.data[a.size]
  a.data[a.size] = False
  result
```

Implementation (4/4)

```
def da_push!(a, v):
    da_ensure_capacity!(a, a.size + 1)
    a.data[a.size] = v
    a.size = a.size + 1
```

Implementation (4/4)

```
def da_push!(a, v):
    da_ensure_capacity!(a, a.size + 1)
    a.data[a.size] = v
    a.size = a.size + 1

def da_ensure_capacity!(a, cap):
    if da_capacity(a) < cap:
        let new_size = max(cap, 2 * da_capacity(a))
        let new_data = [ False; new_size ]
        for i, v in a.data:
            new_data[i] = v
        a.data = new_data
```

Time complexities

- *get*/*set*/*size* are $\mathcal{O}(1)$
- *pop* is $\mathcal{O}(1)$
- *push* is $\mathcal{O}(n)$ still

Time complexities

- *get*/*set*/*size* are $\mathcal{O}(1)$
- *pop* is $\mathcal{O}(1)$
- *push* is $\mathcal{O}(n)$ still

How long does it take to build an n–element array by *push*es?

Time complexities

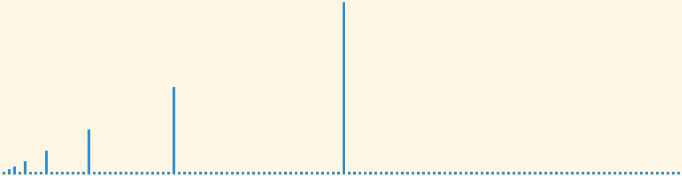
- *get*/*set*/*size* are $\mathcal{O}(1)$
- *pop* is $\mathcal{O}(1)$
- *push* is $\mathcal{O}(n)$ still

How long does it take to build an n–element array by *push*es?

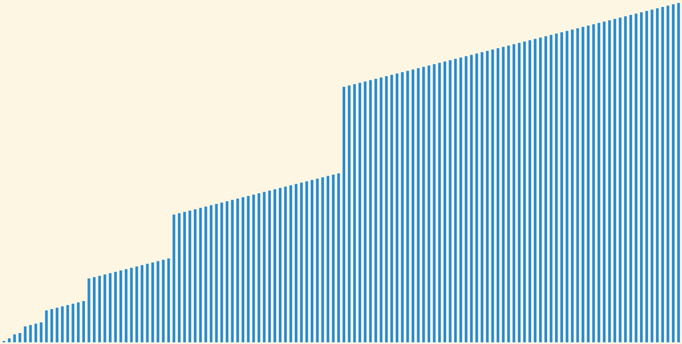
$$\sum_{i=0}^{n} \mathcal{O}(i) = \mathcal{O}(n^2)$$

The peculiar thing about *push*

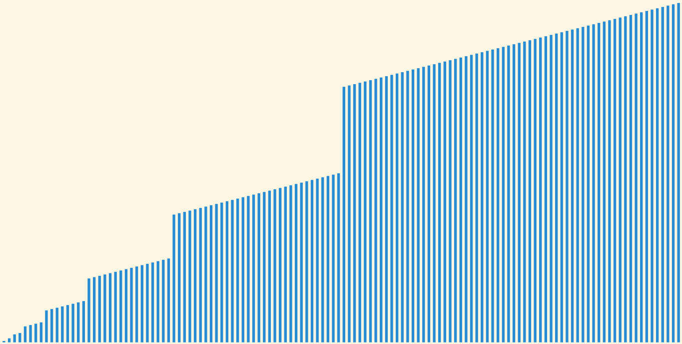
- Most of the time it's cheap
- Only occasionally do we need to grow (which is expensive):



Cumulative time



Cumulative time



It's linear!

Dynamic array aggregate analysis

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

Dynamic array aggregate analysis

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

Let c_i be the cost of the ith insertion:

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is a power of 2} \\ 1 & \text{otherwise} \end{cases}$$

Dynamic array aggregate analysis

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

Let c_i be the cost of the ith insertion:

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is a power of 2} \\ 1 & \text{otherwise} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
s_i	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

Adding it up

Let $d_i = c_i - 1$ (the doubling cost)

Adding it up

Let $d_i = c_i - 1$ (the doubling cost)

Then,

$$\begin{aligned}
\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} (1 + d_i) \\
&= n + \sum_{i=1}^{n} d_i \\
&= n + \sum_{i=0}^{\log_2 n} 2^i \\
&= n + (n + \frac{n}{2} + \frac{n}{4} + \cdots) \\
&\leq 3n
\end{aligned}$$

Example: banker’s queue (FIFO)

Banker’s queue implementation (1/2)

```
# A BankersQueueOf<X> is bq(StackOf<X>, StackOf<X>)
defstruct bq(front, back)
# Interpretation: the queue is the elements of
# `front` in pop order followed by `back` in reverse

def bq_new(cap):
    bq(stack_new(cap), stack_new(cap))

def bq_size(q):
    stack_size(q.front) + stack_size(q.back)

def bq_empty?(q):
    stack_empty?(q.front) and stack_empty?(q.back)
```

Banker's queue implementation (2/2)

```
def bq_enqueue!(q, v):
    stack_push!(q.back, v)
```

Banker's queue implementation (2/2)

```
def bq_enqueue!(q, v):
    stack_push!(q.back, v)

def bq_dequeue!(q):
    if stack_empty?(q.front):
        if stack_empty?(q.back):
            error('bq_dequeue!: empty')
        while !stack_empty?(q.back):
            stack_push!(q.front, stack_pop!(q.back))
    stack_pop!(q.front)
```

Banker's queue analysis (physicist style)

We assign a "potential" to each data structure state:

$$\Phi(q) = \texttt{stack\_size}(q\texttt{.back})$$

Note that the potential of a new queue is 0, and the potential is never negative

Banker's queue analysis (physicist style)

We assign a "potential" to each data structure state:

$$\Phi(q) = \texttt{stack\_size}(q\texttt{.back})$$

Note that the potential of a new queue is 0, and the potential is never negative

Then the amortized cost of an operation is

$$c + \Phi(q') - \Phi(q)$$

where c is the actual cost, q is the state before, and q' is the state after

Actual costs

Actual cost of enqueue operation: 1

Actual costs

Actual cost of enqueue operation: 1

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

Actual costs

Actual cost of enqueue operation: 1

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

Actual cost of expensive dequeue operation (with reversal) is the cost of the reversal (the number of elements reversed) plus the cost of a cheap dequeue: $n + 1$

Amortized cost of enqueue

- Actual cost of enqueue is 1
- Increases the length of the back by 1, hence $\Phi(q') - \Phi(q) = 1$

So amortized cost is $1 + 1 = 2$

Amortized cost of cheap dequeue

- Actual cost of cheap dequeue is 1
- No change in potential

So amortized cost is 1

Amortized cost of expensive dequeue

Let n be \texttt{stack_len(q.back)}, the length of the back stack. Then:

- Actual cost is $n + 1$
- $\Phi(q) = n$ (before reversal)
- $\Phi(q') = 0$ (after reversal)

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

Banker's queue operation worst-case time complexities

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

Next time: hashing