Amortized Time

EECS 214, Fall 2018

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:

```
interface DYN_ARRAY[T]:
    def len(self) -> nat?
    def get(self, index: nat?) -> T
    def set(self, index: nat?, element: T) -> VoidC
    def push(self, element: T) -> VoidC
    def pop(self) -> T
```

Laws:

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

A naïve representation (1/2)

```
class DynArray[T] (DYN_ARRAY):
    let data: VecC[T]

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

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

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

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

    …
```

A naïve representation (2/2)

```
class DynArray[T] (DYN_ARRAY):
    …

    def push(self, element):
        def each(i):
            if i < self.len():
                self.data[i]
            else:
                element
        self.data = [ each(i) for i in self.len() + 1 ]

    def pop(self):
        let new_len = self.len() - 1
        let result = self.data[new_len]
        self.data = [ self.data[i] for i in new_len ]
        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?

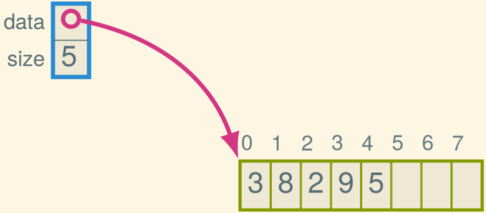
Naïve representation complexities

- *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



This is a *dynamic array*

It’s called:

- `std::vector` in C++
- `ArrayList` in Java
- `list` in Python

Implementation (1/4)

```
class DynArray[T] (DYN_ARRAY):
    let data: VecC[OrC(T, False)]
    let size: nat?

    def __init__(self, initial_capacity: nat?):
        self.data = [False; initial_capacity]
        self.size = 0

    def len(self):
        self.size

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

    …
```

Implementation (2/4)

```
class DynArray[T] (DYN_ARRAY):
    …

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

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

Implementation (2/4)

```
class DynArray[T] (DYN_ARRAY):
    …

    def get(self, index):
        self._bounds_check(index)
        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')

    …
```

Implementation (3/4)

```
class DynArray[T] (DYN_ARRAY):
    …

    def pop(self):
        self.size = self.size - 1
        let result = self.data[self.size]
        self.data[self.size] = False
        result

    …
```

Implementation (4/4)

```
class DynArray[T] (DYN_ARRAY):
    …

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

Implementation (4/4)

```
class DynArray[T] (DYN_ARRAY):
    …

    def push(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 = [ False; cap ]
            for i, v in self.data:
                new_data[i] = v
            self.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

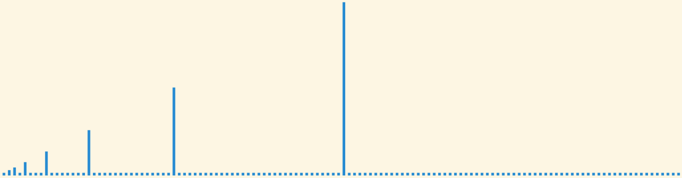
- *get*/*set*/*size* are $\mathcal{O}(1)$
- *pop* is $\mathcal{O}(1)$
- *push* is $\mathcal{O}(n)$ still

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

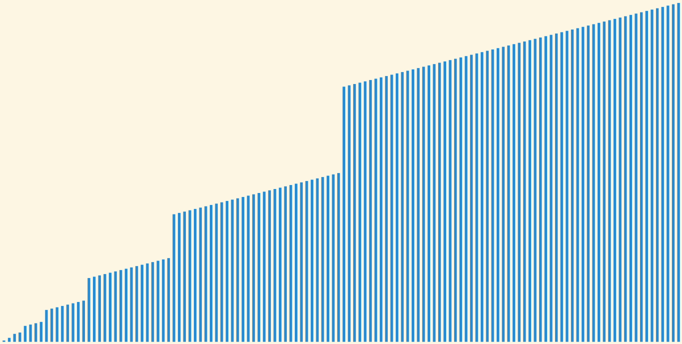
$$\sum_{i=0}^{n} \mathcal{O}(i) = \mathcal{O}(n^2)?$$

The peculiar thing about *push*

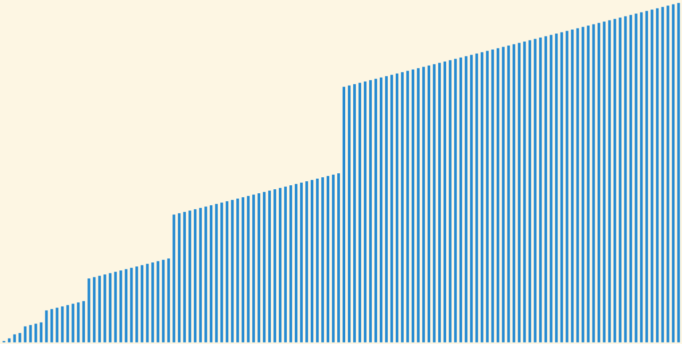
- Most of the time it's cheap
- Only occasionally do we need to grow (which is expensive):



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)

```
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, Stack: FunC[STACK!]):
        self.front = Stack()
        self.back = Stack()

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

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

    …
```

Banker’s queue implementation (2/2)

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

    def enqueue(self, element):
        self.back.push(element)
```

Banker's queue implementation (2/2)

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

    def enqueue(self, element):
        self.back.push(element)

    def dequeue(self):
        if self.front.empty?():
            if self.back.empty?():
                error('BankersQueue.dequeue: empty')
            while not self.back.empty?():
                self.front.push(self.back.pop())
        self.front.pop()
```

Banker's queue analysis (physicist style)

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

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

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:

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

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 `q.back.len()`, 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: random binary search trees