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:

- \( \text{get}(\text{DynArray}, \text{Index}): \text{Element} \)
- \( \text{set}(\text{DynArray}, \text{Index}, \text{Element}): \text{Void} \)
- \( \text{push}(\text{DynArray}, \text{Element}): \text{Void} \)
- \( \text{pop}(\text{DynArray}): \text{Element} \)
- \( \text{size}(\text{DynArray}): \text{Natural} \)

Laws:

- \( \{ a = [v_0, \ldots, v_k] \} \ \text{get}(a, i) = v_i \)
- \( \{ a = [v_0, \ldots, v_k] \} \ \text{set}(a, i, v) \{ a = [v_0, \ldots, v_{i-1}, v, v_{i+1}, \ldots, v_k] \} \)
- \( \{ a = [v_0, \ldots, v_k] \} \ \text{push}(a, v) \{ a = [v_0, \ldots, v_k, v] \} \)
- \( \{ a = [v_0, \ldots, v_k] \} \ \text{pop}(a) = v_k \{ a = [v_0, \ldots, v_{k-1}] \} \)
- \( \{ a = [v_0, \ldots, v_k] \} \ \text{size}(a) = k + 1 \)
A naïve representation (1/2)

# A DynArrayOf<X> is dyn-array(VectorOf<X>)
def struct 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)
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 $O(1)$
- *push/pop* are $O(n)!$
Naïve representation complexities

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

How long does it take to build an $n$–element array by *pushes*?
Naïve representation complexities

- `get/set/size` are $O(1)$
- `push/pop` are $O(n)$!

How long does it take to build an $n$–element array by `pushes`?

$$\sum_{i=1}^{n} O(i) = O(n^2)$$
A better idea: leave extra space in the array

data
size

5

0 1 2 3 4 5 6 7
3 8 2 9 5
Implementation (1/4)

# A DynArray0f<X> is dyn_array(Vector0f<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)
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)

```python
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')
```

def da_pop!(a):
    a.size = a.size - 1
    let result = a.data[a.size]
    a.data[a.size] = False
    result
def da_push!(a, v):
    da_ensure_capacity!(a, a.size + 1)
    a.data[a.size] = v
    a.size = a.size + 1
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

- \( \text{get/set/size are } \mathcal{O}(1) \)
- \( \text{pop is } \mathcal{O}(1) \)
- \( \text{push is } \mathcal{O}(n) \) 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 *pushes*?
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 pushes?

$$\sum_{i=0}^{n} \mathcal{O}(i) = \mathcal{O}(n^2)$$
The peculiar thing about *push*

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

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 \( i \)th 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 $i$th insertion:

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

\[
\begin{array}{c|ccccccccccc}
  i & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
  s_i & 1 & 2 & 4 & 4 & 8 & 8 & 8 & 8 & 16 & 16 \\
  c_i & 1 & 2 & 3 & 1 & 5 & 1 & 1 & 1 & 9 & 1 \\
\end{array}
\]
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,

\[
\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} (1 + d_i) \\
= n + \sum_{i=1}^{n} d_i \\
= n + \sum_{i=0}^{\log_2 n} 2^i \\
= n + (n + \frac{n}{2} + \frac{n}{4} + \cdots ) \\
\leq 3n
\]
Example: banker’s queue (FIFO)
Banker’s queue implementation (1/2)

# A BankersQueueOf<X> is bq(Stack0f<X>, Stack0f<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)

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)
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:

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

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:

\[ \Phi(q) = \text{stack\_size}(q\text{.back}) \]

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

<table>
<thead>
<tr>
<th>operation</th>
<th>single operation</th>
<th>amortized</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>dequeue</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Next time: hashing