Amortized Time

CS 214, Fall 2019
We never said how much a single union or find operation costs

Instead, we said that \( m \) operations on \( n \) objects is

\( \mathcal{O}((m + n) \log^* n) \)
Remember Union-Find?

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

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

This is because some long-running operations do maintenance that make other operations faster.
Example: dynamic array
Dynamic Array ADT (1/2)

Looks like: \([3, 8, 2, 90, 5]\)

Signature:

```python
interface DYN_ARRAY[T]:
    def len(self) -> nat?
    def get(self, index: nat?) -> T
    def set(self, index: nat?, element: T) -> NoneC
    def push(self, element: T) -> NoneC
    def pop(self) -> T
```
Dynamic Array ADT (2/2)

Laws:

\[
\begin{align*}
&\{ a = [v_1, \ldots, v_k] \} \quad a.\text{len}() \Rightarrow k \quad \{ a = a_0 \} \\
&\{ a = [v_1, \ldots, v_k] \} \quad a.\text{push}(w) \Rightarrow \text{None} \quad \{ a = [v_1, \ldots, v_k, w] \} \\
&\{ a = [v_0, \ldots, v_{k-1}, v_k] \} \quad a.\text{pop}() \Rightarrow v_k \quad \{ a = [v_0, \ldots, v_{k-1}] \} \\
&\{ a = [\ldots, v_i, \ldots] \} \quad a.\text{get}(i) \Rightarrow v_i \quad \{ a = a_0 \} \\
&\{ a = [\ldots, v_{i-1}, v_i, v_{i+1}, \ldots] \} \quad a.\text{set}(i, w) \Rightarrow \text{None} \\
&\{ a = [\ldots, v_{i-1}, w, v_{i+1}, \ldots] \}
\end{align*}
\]
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, val):
        let n = self.len()
        self._data = [ self._data[i] if i < n else val
                        for i in range(n + 1) ]

    def pop(self):
        let n = self.len()
        let val = self._data[n - 1]
        self._data = [ self._data[i]
                        for i in range(n - 1) ]
        return val
Naïve representation complexities

- \( \text{get/set/len} \) are \( \mathcal{O}(1) \)
- \( \text{push/pop} \) are \( \mathcal{O}(n)! \)
Naïve representation complexities

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

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

- \textit{get/set/len} are $O(1)$
- \textit{push/pop} are $O(n)!$

How long does it take to build an $n$–element array by \textit{pushes}?

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

```
struct dyn_int_array {
    int* data_;  // number of elements
    size_t len_;  // max `len_` without realloc
    size_t cap_;  // number of elements
};
```
A better idea: leave extra space in the array

Note: DSSL2 vectors know their sizes and can tell you, but C pointers don’t know how many objects they point to, so in C you need to store the capacity yourself:

```c
struct dyn_int_array
{
    int* data_;  // store pointers
    size_t len_;  // number of elements
    size_t cap_;  // max `len_` without realloc
};
```
This is a dynamic array

It’s called:

- `std::vector` in C++
- `ArrayList` in Java
- `list` in Python
class DynArray[T] (DYN_ARRAY):
    let _data: VecC[OrC(T, NoneC)]
    let _len:  nat?

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

def len(self):
    return self._len

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

...
class DynArray[T] (DYN_ARRAY):
    ...

    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
class DynArray[T] (DYN_ARRAY):
    ...

    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.len():
            error('DynArray: out of bounds')

    ...

...
Implementation (3/4)

```python
class DynArray[T] (DYN_ARRAY):
    ...

    def pop(self):
        self._len = self._len - 1
        return self._data[self._len]
```

class DynArray[T] (DYN_ARRAY):
    ...

    def pop(self):
        self._len = self._len - 1
        return self._data[self._len]

    # Avoids memory leaks:
    def pop(self):
        self._len = self._len - 1
        let result = self._data[self._len]
        self._data[self._len] = None
        return result

    ...

13
class DynArray[T] (DYN_ARRAY):
    ...

def push(self, element):
    self._ensure_capacity(self._len + 1)
    self._data[self._len] = element
    self._len = self._len + 1
class DynArray[T] (DYN_ARRAY):
    ...

    def push(self, element):
        self._ensure_capacity(self._len + 1)
        self._data[self._len] = element
        self._len = self._len + 1

    def _ensure_capacity(self, cap):
        if cap <= self.capacity(): return
        cap = max(cap, 2 * self.capacity())
        self._data = vec_copy_resize(cap, self._data)

    ...

...
Time complexities

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

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

How long does it take to build an $n$–element array by \textit{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!
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?

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{align*}
  i & \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \\
  s_i & \quad 1 \quad 2 \quad 4 \quad 4 \quad 8 \quad 8 \quad 8 \quad 8 \quad 16 \quad 16 \\
  c_i & \quad 1 \quad 2 \quad 3 \quad 1 \quad 5 \quad 1 \quad 1 \quad 1 \quad 9 \quad 1
\end{align*}
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:

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

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>$c_i$</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
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)

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)
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:

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

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:

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

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

<table>
<thead>
<tr>
<th>operation</th>
<th>single operation</th>
<th>amortized</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
<tr>
<td>dequeue</td>
<td>$\mathcal{O}(n)$</td>
<td>$\mathcal{O}(1)$</td>
</tr>
</tbody>
</table>
Next time: random binary search trees