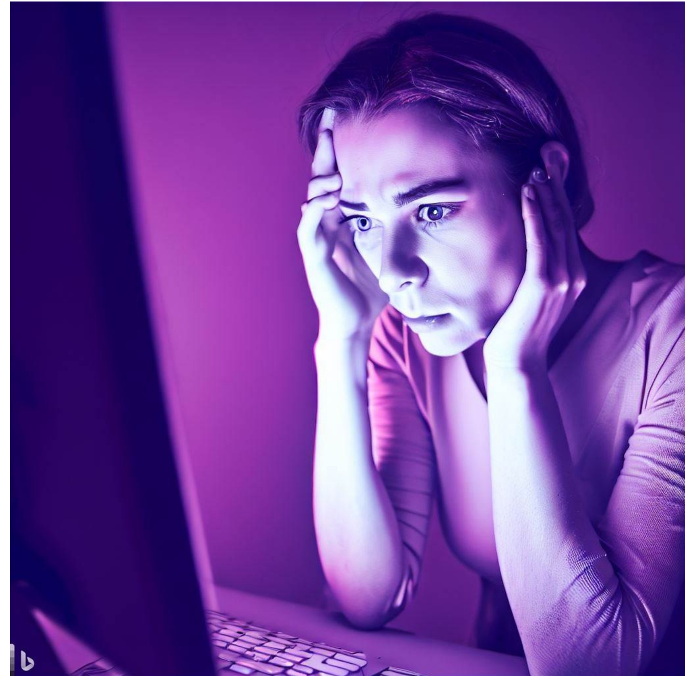


CSE 331

Mutable ADTs

Kevin Zatloukal

Last Time

- Started working with mutable heap state (Level 3)

- Saw how aliasing makes correctness non-local
  - destroys modularity
  - errors in one place show up much later, far away in code
  - correctness is too hard

- Best option is to disallow aliasing of ADT state
  - copy in, copy out to avoid “rep exposure”
  - you have the only reference to your mutable state
  - still difficult

Last Time

- Specifying & reasoning about mutable ADTs
  - mutator methods instead of producers
  - spec says how obj is changed (in `@effects`)

- Can still have errors in one method show up in another
  - but all within *one file*

- Methods coordinate via the RI
  - data written in one method can be read in another
  - methods must all work together

Recall: List ADT with a Fast `getLast`

```
class FastListImpl implements FastList {

  // RI: this.last = last(this.list)
  // AF: obj = this.list
  readonly last: number | undefined;
  readonly list: List<number>;

  constructor(list: List<number>) {
    this.list = list;
    this.last = last(this.list);
  }

  getLast = (): number => { return this.last; };
```

- Calculates `this.last` in the constructor
  - constructor is $O(n)$ but `getLast` is $O(1)$

Alternative Concrete Rep for `FastList`

```
class FastListImpl implements FastList {

  constructor(list: List<number>) {
    this.list = list;
    this.last = undefined;
  }

  getLast = (): number => {
    if (this.last === undefined)
      this.last = last(this.list);
    return this.last;
  };
```

- **Calculates** `this.last` **in** `getLast`
  - **constructor is** $O(1)$
  - **second and later calls to** `getLast` **are** $O(1)$

Alternative Concrete Rep for `FastList`

```
class FastListImpl implements FastList {

  // RI: this.last = last(this.list)
  // AF: obj = this.list
  last: number | undefined;
  list: List<number>;

  constructor(list: List<number>) {
    this.list = list;
    this.last = undefined;
  }
```

no longer always true that this.last = last(this.list)!

- *This* RI does not hold at the end of the constructor
  - we need a different RI
  - this one doesn't describe what we wanted

Alternative Concrete Rep for `FastList`

```
class FastListImpl implements FastList {

  // RI: if this.last != undefined,
  //       then this.last = last(this.list)
  // AF: obj = this.list

  constructor(list: List<number>) {
    this.list = list;
    this.last = undefined;
  }

  getLast = (): number => {
    if (this.last === undefined)
      this.last = last(this.list);
    return this.last;
  };
```

RI says postcondition holds in the ”else” branch

- Updated RI formalizes what is true here

Alternative Concrete Rep for `FastList`

```
class FastListImpl implements FastList {

  // RI: if this.last != undefined,
  //       then this.last = last(this.list)
  // AF: obj = this.list

  constructor(list: List<number>) {
    this.list = list;
    this.last = undefined;
  }

  getLast = (): number => {
    if (this.last === undefined)
      this.last = last(this.list);
    return this.last;
  };
```

class is still immutable!
(it has no mutators)

- **Only concrete state is changed, not abstract state**

Alternative Rep for `MutableFastList`

```
class MutableFastListImpl implements MutableFastList {
  // RI: if this.last != undefined,
  //       then this.last = last(this.list)
  // AF: obj = this.list
  last: number | undefined;
  list: List<number>;

  // @modifies obj
  // @effects obj = cons(x, obj_0)
  cons = (x: List<number>): void => {
    this.list = cons(x, list);
    this.last = undefined;
```

{{ this.list = cons(x, this.list_0) and **this.last = undefined** }}

{{ **Post**: obj = this.list and
if this.last ≠ undefined, this.last = last(this.list) }}

```
};
```

Rep Invariant holds (second part is vacuously true)

Morals of the Story for Level 3

- More mutation gave us better efficiency
  - immutable version could be just as fast (only level 1)

- More mutation means more complex reasoning
  - more facts to keep track of
  - more ways to make mistakes

- RI is critical to make methods work together
  - methods that read get what they need from those that write
  - formalizes the data structure idea

    like how loop invariants formalize the algorithm idea

    different RIs translate to different code

Recall: Mutable Queue ADT

- Mutable versions has mutators instead of producers

```
// Mutable array that only supports adding to the front
// and removing from the end.
interface MutableNumberQueue {

  // @returns obj
  elements(): number[];

  // @modifies obj
  // @effects obj = [x] ++ obj_0
  enqueue(x: number): void;

  // @requires len(obj) > 0
  // @modifies obj
  // @effects obj_0 = obj ++ [x]
  // @returns x
  dequeue(): number;
}
```

observer

mutator

mutator

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  readonly front: number[];
  readonly back: number[];

  // @modifies obj
  // @effects obj = [x] ++ obj_0
  enqueue = (x: number): void => {
    this.front.push(x);
  };
```

- **We are supposed to add to the front of the queue**
  - **why are we adding to the *back* of `front`?**

    because the abstract state is the *reverse* of `front`

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  readonly front: number[];
  readonly back: number[];

  // @modifies obj
  // @effects obj = [x] ++ obj_0
  enqueue = (x: number): void => {
    this.front.push(x);
    {{ this.front = this.front_0 ++ [x] }}
    {{ Post: obj = [x] ++ obj_0 }}
  };
```

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  readonly front: number[];
  readonly back: number[];

  // @modifies obj
  // @effects obj = [x] ++ obj_0
  enqueue = (x: number): void => {
    this.front.push(x);
    {{ this.front = this.front_0 ++ [x] }}
    {{ Post: obj = [x] ++ obj_0 }}
  };
```

obj	$= \text{rev}(\text{this.front}) + \text{this.back}$	**by AF**
	$= \text{rev}(\text{this.front}_0 + [x]) + \text{this.back}$	**since** this.front = ...
	$= [x] + \text{rev}(\text{this.front}_0) + \text{this.back}$	**def of** rev
	$= [x] + \text{obj}_0$	**by AF**

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back
  readonly front: number[];
  readonly back: number[];

  // @requires len(obj) > 0
  // @modifies obj
  // @effects obj_0 = obj ++ [x]
  // @returns x
  dequeue = (): number => {
    let x: number = this.back.pop();
    return x;
  };
```

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back

  // @requires len(obj) > 0
  // @modifies obj
  // @effects obj_0 = obj ++ [x]
  // @returns x
  dequeue = (): number => {
    let x: number = this.back.pop();
    {{ this.back₀ = this.back ++ [x] }}
    {{ Post: obj₀ = obj ++ [x] }}
    return x;
  };
```

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back

  // @requires len(obj) > 0
  // @modifies obj
  // @effects obj_0 = obj ++ [x]
  // @returns x
  dequeue = (): number => {
    let x: number = this.back.pop();
    {{ this.back_0 = this.back ++ [x] }}
    {{ Post: obj_0 = obj ++ [x] }}
    return x;
  };
```

We skipped something...

Need to check `pop`'s precondition

Do we know $\text{this.back.length} > 0$?

No, we do not... this is a bug!

$\text{obj}_0 = \text{rev(this.front)} \mathbin{+\!\!+} \text{this.back}_0$	by AF
$= \text{rev(this.front)} \mathbin{+\!\!+} \text{this.back} \mathbin{+\!\!+} [x]$	since $\text{this.back}_0 = \ldots$
$= \text{obj} \mathbin{+\!\!+} [x]$	by AF

Implementing Mutable Queue with Two Arrays

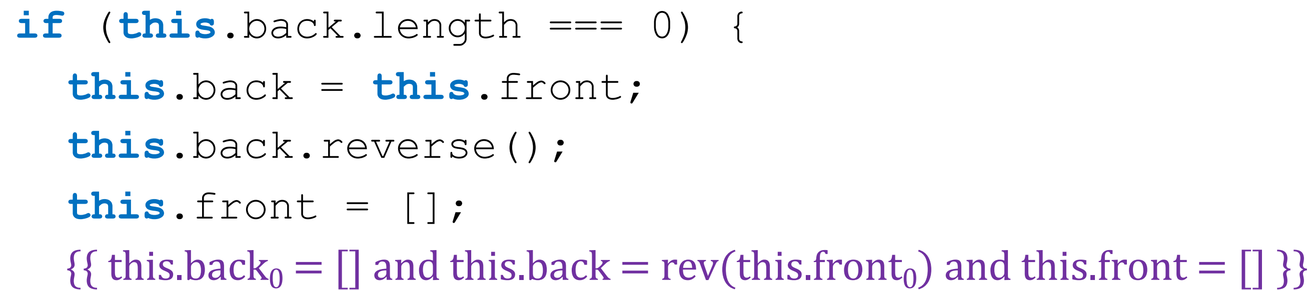
```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back

  // @requires len(obj) > 0
  // @modifies obj
  // @effects obj_0 = obj ++ [x]
  // @returns x
  dequeue = (): number => {
    if (this.back.length === 0) {
      this.back = this.front;
      this.back.reverse();
      this.front = [];
    }
    let x: number = this.back.pop();
    return x;
  };
```

Implementing Mutable Queue with Two Arrays

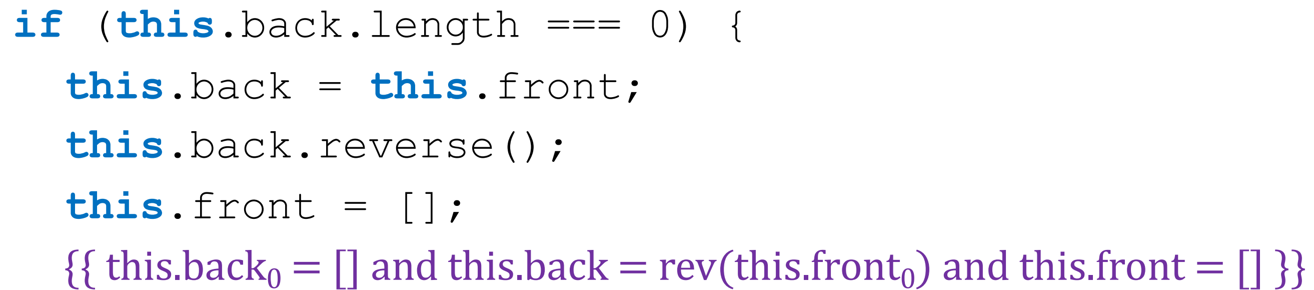
```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back

  // @requires len(obj) > 0
  // @modifies obj
  // @effects obj_0 = obj ++ [x]
  // @returns x
  dequeue = (): number => {
    if (this.back.length === 0) {
      this.back = this.front;
      this.back.reverse();
      this.front = [];
      {{ this.back_0 = [] and this.back = rev(this.front_0) and this.front = [] }}
```

- **Claim: abstract state is not changed** $(\text{obj} = \text{obj}_0)$
  - let's check this…

Implementing Mutable Queue with Two Arrays



Implementing Mutable Queue with Two Arrays



obj	$= \text{rev(this.front)} \mathbin{+\!\!+} \text{this.back}$	**by AF**
	$= \text{rev([])} \mathbin{+\!\!+} \text{this.back}$	**since** this.front = []
	$= \text{this.back}$	**def of** rev
	$= \text{rev}(\text{this.front}_0)$	**since** this.front = ...
	$= \text{rev}(\text{this.front}_0) \mathbin{+\!\!+} []$	
	$= \text{rev}(\text{this.front}_0) \mathbin{+\!\!+} \text{this.back}_0$	**since** $\text{this.back}_0 = []$
	$= \text{obj}_0$	**by AF**

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back

  // @requires len(obj) > 0
  // @modifies obj
  // @effects obj_0 = obj ++ [x]
  // @returns x
  dequeue = (): number => {
    if (this.back.length === 0) {
      this.back = this.front;
      this.back.reverse();
      this.front = [];
      {{ this.back_0 = [] and this.back = rev(this.front_0) and this.front = [] }}
```

- **Abstract state is unchanged, but is the bug gone?**
  - **do we know that** this.back.length > 0**?**

Implementing Mutable Queue with Two Arrays

```
// Implements a mutable queue using two arrays.
class ArrayPairQueue implements MutableNumberQueue {

  // AF: obj = rev(this.front) ++ this.back

  // @requires len(obj) > 0
  // @modifies obj
  // @effects obj_0 = obj ++ [x]
  // @returns x
  dequeue = (): number => {
    if (this.back.length === 0) {
      this.back = this.front;
      this.back.reverse();
      this.front = [];
      {{ this.back_0 = [] and this.back = rev(this.front_0) and this.front = [] }}
```

- Is the bug gone? Yes!
  - $\text{len}(\text{obj}) > 0$ **means at least one of** this.front **and** this.back **is not** []
  - **we know** this.back = [], **so** this.front **cannot be** []

Using All the Machinery

- Postcondition (specification) talks about obj

$\{\{$ this.front $=$ this.front$_0$ $+\!\!\!+$ [x] $\}\}$
$\{\{$ **Post**: obj $=$ [x] $+\!\!\!+$ obj$_0$ $\}\}$

- AF translates this into claims about concrete state
  - usually, the first and last line of the calculation

obj $=$ rev(this.front) $+\!\!\!+$ this.back   **by AF**
$=$ …
$=$ [x] $+\!\!\!+$ obj$_0$ **by AF**

  - then we can apply the usual tools to prove it

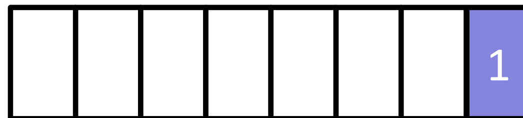
Morals of the Story for Level 3

- Paying a cost for introducing abstraction
  - AF needs to be applied in most calculations
- Paying a cost for having tricky data structures
  - carefully design RI and make sure it always holds
- Lean on the tools when it gets tricky
  - the tools can find all the bugs
  - most necessary on the trickiest problems

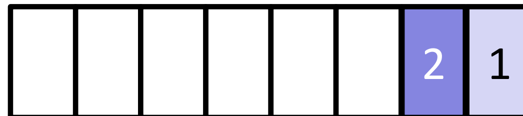
Implementing Mutable Queue with One Array

- Can implement a queue (quickly in one array)
  - add new elements at the end

enqueue 1

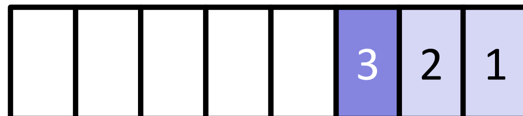


enqueue 2



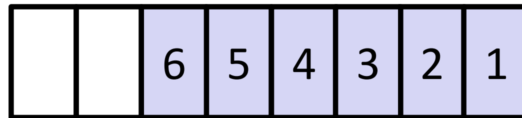
Leave for elements to be added

enqueue 3

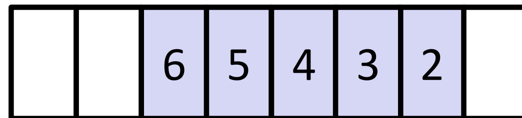


Implementing Mutable Queue with One Array

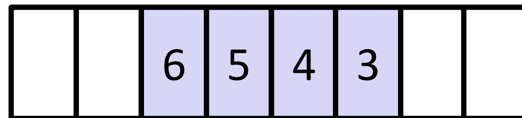
- Can implement a queue (quickly in one array)
  - remove elements from the front



dequeue 1



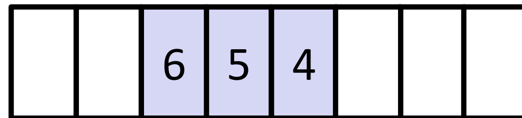
dequeue 2



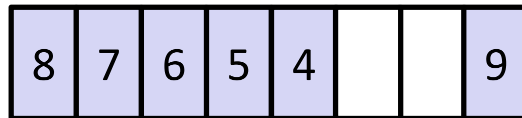
  - leaves empty space at the front and the back

Implementing Mutable Queue with One Array

- Can implement a queue (quickly in one array)



enqueue 7, 8, 9



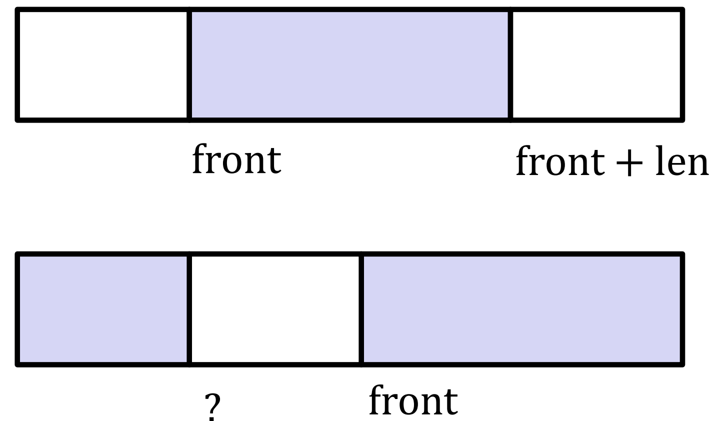
  – now there the empty space is in the middle!

- Thankfully, this is all that can happen
  – empty space is either in the middle or at the ends

Implementing Mutable Queue with One Array

- **Will record** front **index and** len **of queue**

- **In general, the array looks one of these**



- **Abstraction function needs to handle both cases**
  - **first case holds if** $\mathrm{front} + \mathrm{len} \le \mathrm{vals.length}$

Implementing Mutable Queue with One Array

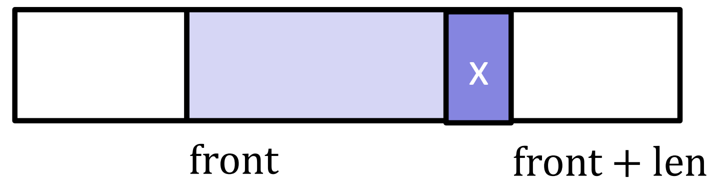
```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {

  // RI: 0 <= front < vals.length and
  //     0 <= len <= vals.length
  readonly vals: number[];
  readonly front: number;
  readonly len: number;
```

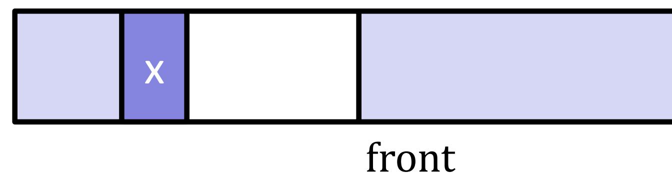
Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {

  // RI: 0 <= front < vals.length and
  //     0 <= len <= vals.length
  // AF: obj = vals[front .. front+len-1]
  //           if front + len <= this.vals.length
  //     obj = vals[front .. vals.length - 1]
  //             ++ vals[0 .. front + len - vals.length - 1]
  //           otherwise
  readonly vals: number[];
  readonly front: number;
  readonly len: number;
```

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {

  // @requires len(obj) > 0
  // @modifies obj
  // @effects obj_0 = obj ++ [x]
  // @returns x
  dequeue = (): number => { … };
```





Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {

  // @requires len(obj) > 0
  // @modifies obj
  // @effects obj_0 = obj ++ [x]
  // @returns x
  dequeue = (): number => {
    let x;
    if (this.front + this.len <= this.vals.length) {
      x = this.vals[this.front + this.len - 1];
    } else {
      const k = this.front + this.len - this.vals.length - 1;
      x = this.vals[k];
    }
    this.len = this.len - 1;
    return x;
  };
```

Need to check RI and postcondition

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {

  // @requires len(obj) > 0
  // @modifies obj
  // @effects obj_0 = obj ++ [x]
  // @returns x
  dequeue = (): number => {
    let x;
    if (this.front + this.len <= this.vals.length) {
      x = this.vals[this.front + this.len - 1];
    } else {
      const k = this.front + this.len - this.vals.length - 1;
      x = this.vals[k];
    }
    this.len = this.len - 1;
    return x;
  };
```

Check that the RI holds:

$0 \leq \text{front} < \text{vals.length}$ **(still) holds**

Why does $0 \leq \text{len} \leq \text{vals.length}$ **hold?**

Precondition ensures $0 \leq \text{len} - 1$

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {

  // @effects obj_0 = obj ++ [x]
  dequeue = (): number => {
    let x = undefined;
    if (this.front + this.len <= this.vals.length) {
      x = this.vals[this.front + this.len - 1];
      {{ front + len ≤ vals.length and x = vals[front + len – 1] }}
    } else {
      const k = this.front + this.len - this.vals.length - 1;
      x = this.vals[k];
      {{ front + len > vals.length and x = vals[front + len - vals.length – 1] }}
    }
    this.len = this.len - 1;
    return x;
  };
```

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {

  // @effects obj_0 = obj ++ [x]
  dequeue = (): number => {
    let x = undefined;
    if (this.front + this.len <= this.vals.length) {
      x = this.vals[this.front + this.len - 1];
      {{ front + len ≤ vals.length and x = vals[front + len0 – 1] }}
    } else {
      const k = this.front + this.len - this.vals.length - 1;
      x = this.vals[k];
      {{ front + len > vals.length and x = vals[front + len - vals.length – 1] }}
    }
    this.len = this.len - 1;
    {{ (front + len_0 ≤ vals.length and x = vals[front + len_0 – 1] or
       (front + len_0 > vals.length and x = vals[front + len_0 – vals.length – 1]) and
       len = len_0 – 1 }}
```

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {

  // @effects obj_0 = obj ++ [x]
  dequeue = (): number => {
    ...
    {{ (front + len_0 ≤ vals.length and x = vals[front + len_0 – 1] or
       (front + len_0 > vals.length and x = vals[front + len_0 – vals.length – 1]) and
       len = len_0 – 1 }}
    {{ Post: obj_0 = obj ++ [x] }}
    return x;
  };
```

- How do we prove this? (Notice the "or")
  - proof by cases

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {

  // @effects obj_0 = obj ++ [x]
  dequeue = (): number => {
    …
    {{ (front + len_0 ≤ vals.length and x = vals[front + len_0 – 1] or
        (front + len_0 > vals.length and x = vals[front + len_0 – vals.length – 1]) and
        len = len_0 – 1 }}
    {{ Post: obj_0 = obj ++ [x] }}
    return x;
  };
```

- To prove "$obj_0 = obj \mathbin{+\!\!+} [x]$", we need to apply the AF
  - but the AF itself has cases…
  - need to know which case applies in order ot use the AF

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {
```

{{ (front + $len_0 \le$ vals.length and x = vals[front + $len_0 - 1$] or
(front + $len_0 >$ vals.length and x = vals[front + len_0 – vals.length – 1]) and
len = $len_0 - 1$ }}

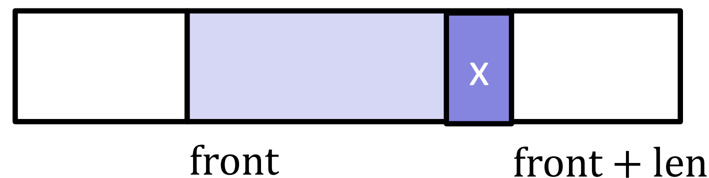
{{ **Post**: obj_0 = obj ++ [x] }}

```
    return x;
```

Case front + $len_0 \le$ vals.length and x = vals[front + $len_0 - 1$]:

Since front + $len_0 \le$ vals.length, **we have** obj_0 = vals[front .. front + len – 1]

Since len $\le len_0$, **we also have** front + len $\le$ vals.length, **so** **we know that** obj = vals [front .. front + len – 1] **as well**



Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {
```

{{ (front + $len_0 \le$ vals.length and x = vals[front + len_0 – 1] or
(front + $len_0 >$ vals.length and x = vals[front + len_0 – vals.length – 1]) and
len = len_0 – 1 }}

{{ **Post**: obj_0 = obj ⧺ [x] }}

```
    return x;
```

Case front + $len_0 \le$ vals.length and x = vals[front + len_0 – 1]:

obj_0 = vals[front .. front + len_0 – 1]	**by AF**
= vals[front .. front + len]	**since** len = len_0 – 1
= vals[front .. front + len – 1] ⧺ [vals[front + len]]	
= obj ⧺ [vals[front + len]]	**by AF**
= obj ⧺ [vals[front + len_0 – 1]]	**since** len = len_0 – 1
= obj ⧺ [x]	**since** x = vals[…]

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {
```

{{ (front + len_0 ≤ vals.length and x = vals[front + len_0 – 1] or
(front + len_0 > vals.length and x = vals[front + len_0 – vals.length – 1]) and
len = len_0 – 1 }}

{{ **Post**: obj_0 = obj ⧺ [x] }}

```
return x;
```

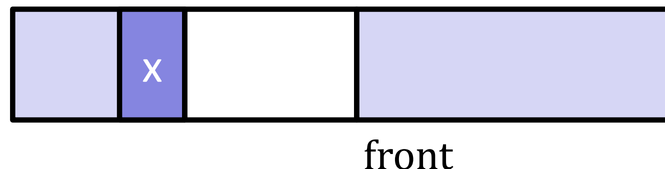
Case front + len_0 > vals.length and x = vals[front + len_0 – vals.length – 1]:

Since front + len_0 > vals.length, **we have**

obj_0 = vals[front .. vals.length - 1] ⧺ vals[0 .. front + len_0 – vals.length – 1]

Assume the same holds for obj **(other case later), so**

obj = vals[front .. vals.length - 1] ⧺ vals[0 .. front + len – vals.length – 1]



Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {
```

{{ (front + $len_0 \le$ vals.length and x = vals[front + len_0 – 1] or
(front + $len_0 >$ vals.length and x = vals[front + len_0 – vals.length – 1]) and
len = len_0 – 1 }}

{{ **Post**: obj_0 = obj ⧺ [x] }}

```
return x;
```

Case front + $len_0 >$ vals.length and x = vals[front + len_0 – vals.length – 1]:

Define k := front – vals.length

obj_0 = vals[front .. vals.length - 1] ⧺ vals[0 .. k + len_0 – 1]
= vals[front .. vals.length - 1] ⧺ vals[0 .. k + len] **since** len = ...
= vals[front .. vals.length - 1] ⧺ vals[0 .. k + len – 1] ⧺ [vals[k+len]]
= obj ⧺ [vals[k + len]] **by AF**
= obj ⧺ [vals[k + len_0 – 1]] **since** len = ...
= obj ⧺ [x] **since** x = ...

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {
```

{{ (front + $len_0 \le$ vals.length and x = vals[front + len_0 – 1] or
(front + $len_0 >$ vals.length and x = vals[front + len_0 – vals.length – 1]) and
len = len_0 – 1 }}
{{ **Post**: obj_0 = obj ⧺ [x] }}

```
return x;
```

Case front + $len_0 >$ vals.length and x = vals[front + len_0 – vals.length – 1]:

Now, assume we have front + len $\le$ vals.length (**last case**)

Since len = len_0 – 1, **this is only possible if** front + len_0 = vals.length + 1.

Thus, we have

obj_0 = vals[front .. vals.length - 1] ⧺ vals[0 .. front + len_0 – vals.length – 1]
obj = vals[front .. front + len_0 - 1]

(front + len_0 – vals.length – 1) = 0

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {
```

{{ ($front + len_0 \le vals.length$ and $x = vals[front + len_0 - 1]$ or
($front + len_0 > vals.length$ and $x = vals[front + len_0 - vals.length - 1]$) and
$len = len_0 - 1$ }}

{{ **Post**: $obj_0 = obj \mathbin{+\!\!+} [x]$ }}

```
    return x;
```

Case $front + len_0 > vals.length$ and $x = vals[front + len_0 - vals.length - 1]$:

Now, assume we have $front + len \le vals.length$ **(last case)**

Since $len = len_0 - 1$, **this is only possible if** $front + len_0 = vals.length + 1$.

Thus, we have

$$obj_0 = vals[front .. vals.length - 1] \mathbin{+\!\!+} [vals[0]]$$
$$obj \;\; = vals[front .. front + len - 1]$$

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {
```

{{ (front + $len_0 \le$ vals.length and x = vals[front + len_0 – 1] or
(front + $len_0 >$ vals.length and x = vals[front + len_0 – vals.length – 1]) and
len = len_0 – 1 }}

{{ **Post**: obj_0 = obj ⧺ [x] }}

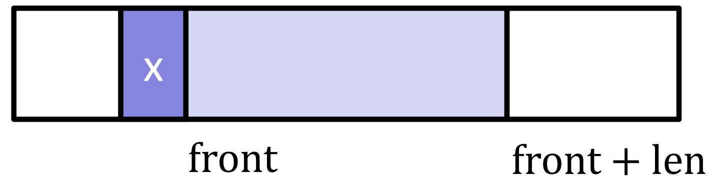
```
return x;
```

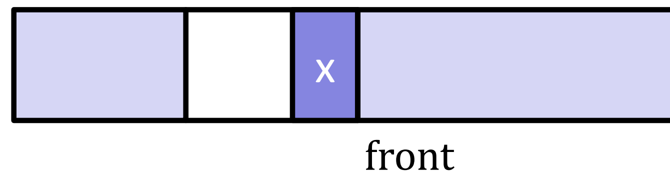
Case front + len_0 = vals.length + 1 and x = vals[front + len_0 – vals.length – 1]:

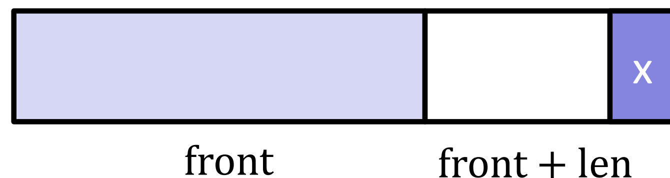
obj_0 = vals[front .. vals.length – 1] ⧺ [vals[0]]
= vals[front .. vals.length – 1] ⧺ [x] — **since** x = ...
= vals[front ... front + len_0 – 1 – 1] ⧺ [x] — **since** ...
= vals[front ... front + len – 1] ⧺ [x] — **since** len = ...
= obj ⧺ [x] — **since** obj = ...

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {

  // @modifies obj
  // @effects obj = [x] ++ obj_0
  enqueue = (x: number): void => { … };
```







Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {

  // @modifies obj
  // @effects obj = [x] ++ obj_0
  enqueue = (x: number): void => {
    this.ensureEnoughSpace();  // make sure there's room

    if (this.front === 0) {
      this.front = this.vals.length - 1;
    } else {
      this.front = this.front - 1;
    }
    this.vals[this.front] = x;
    this.len = this.len + 1;
  };
```

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {

  // @effects obj = [x] ++ obj_0
  enqueue = (x: number): void => {
    if (this.front === 0) {
      this.front = this.vals.length - 1;
    } else {
      this.front = this.front - 1;
    }
    this.vals[this.front] = x;
    this.len = this.len + 1;
    {{ (front_0 = 0 and front = vals.length – 1 or front_0 > 0 and front = front_0 – 1) or
       vals[front] = x and len = len_0 + 1 }}
    {{ Post: obj = [x] ++ obj_0 }}
  };
```

Implementing Mutable Queue with One Array

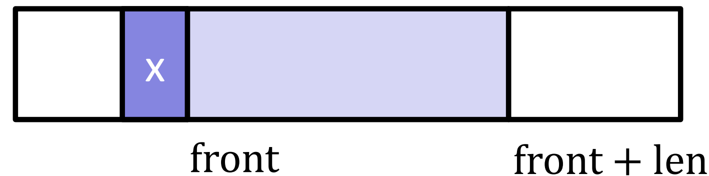
```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {
```

{{ ($front_0 = 0$ and front = vals.length – 1 or $front_0 > 0$ and front = $front_0 - 1$) or
vals[front] = x and len = $len_0 + 1$ }}
{{ **Post**: obj = [x] ⧺ obj_0 }}

Case $front_0 > 0$ and front + len ≤ vals.length:

AF says obj = vals[front .. front + len – 1]

Since $front_0 + len_0$ = front + 1 + len – 1 = front + len ≤ vals.length,
we have obj_0 = vals[front .. front + len_0 – 1] **as well**



Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {
```

{{ ($front_0 = 0$ and front = vals.length – 1 or $front_0 > 0$ and $front = front_0 - 1$) or
vals[front] = x and $len = len_0 + 1$ }}

{{ **Post**: $obj = [x] \mathbin{+\!\!\!+} obj_0$ }}

Case $front_0 > 0$ and $front + len \le vals.length$:

obj	= vals[front .. front + len – 1]	**by AF**
	$= vals[front_0 - 1 .. front_0 - 1 + len - 1]$	**since** front = ...
	$= vals[front_0 - 1 .. front_0 + len_0 - 1]$	**since** len = ...
	$= [vals[front_0 - 1]] \mathbin{+\!\!\!+} vals[front_0 .. front_0 + len_0 - 1]$	
	$= [vals[front]] \mathbin{+\!\!\!+} vals[front_0 .. front_0 + len_0 - 1]$	**since** front = ...
	$= [x] \mathbin{+\!\!\!+} vals[front_0 .. front_0 + len_0 - 1]$	**since** x = ...
	$= [x] \mathbin{+\!\!\!+} obj_0$	**by AF**

Implementing Mutable Queue with One Array

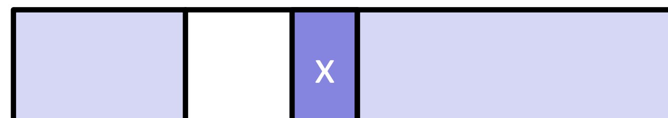
```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {
```

{{ ($front_0 = 0$ and front = vals.length – 1 or $front_0 > 0$ and front = $front_0 - 1$) or
vals[front] = x and len = $len_0 + 1$ }}
{{ **Post**: obj = [x] ⧺ obj_0 }}

Case $front_0 > 0$ and front + len > vals.length:

AF says obj = vals[front .. vals.length – 1] ⧺ vals[0 .. front + len – vals.length – 1]

Since $front_0 + len_0$ = front + 1 + len – 1 = front + len, **we also have** obj_0 = vals[$front_0$ .. vals.length – 1] ⧺ vals[0 .. $front_0 + len_0$ – vals.length – 1]



Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {
```

{{ ($front_0$ = 0 and front = vals.length – 1 or $front_0$ > 0 and front = $front_0$ – 1) or
vals[front] = x and len = len_0 + 1 }}

{{ **Post**: obj = [x] ⧺ obj_0 }}

Case $front_0 > 0$ and front + len > vals.length:

obj = vals[front .. vals.length – 1] ⧺ vals[0 .. front + len – vals.length – 1]
= vals[$front_0$ – 1 .. vals.length – 1] ⧺ vals[0 .. $front_0$ + len – vals.length – 2]
= vals[$front_0$ – 1 .. vals.length – 1] ⧺ vals[0 .. $front_0$ + len_0 – vals.length – 1]
= [vals[$front_0$ – 1]] ⧺ vals[$front_0$ .. vals.length – 1] ⧺ vals[0 .. $front_0$ + len_0 – vals.length – 1]
= [vals[$front_0$ – 1]] ⧺ obj_0
= [vals[front]] ⧺ obj_0
= [x] ⧺ obj_0

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {
```

{{ ($front_0 = 0$ and $front = vals.length - 1$ or $front_0 > 0$ and $front = front_0 - 1$) or
$vals[front] = x$ and $len = len_0 + 1$ }}
{{ **Post**: $obj = [x] +\!\!+ obj_0$ }}

Case $front_0 = 0$ and $front + len \leq vals.length$:

AF says $obj_0 = vals[front_0 .. front_0 + len_0 - 1] = vals[0 .. len_0 - 1]$

Since $front = vals.length - 1$, **the AF says we have** $obj = [vals[vals.length - 1]] +\!\!+ vals[0 .. len - 2]$

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {
```

{{ ($front_0 = 0$ and $front = vals.length - 1$ or $front_0 > 0$ and $front = front_0 - 1$) or
$vals[front] = x$ and $len = len_0 + 1$ }}
{{ **Post**: $obj = [x] \mathbin{+\!\!+} obj_0$ }}

Case $front_0 = 0$ and $front + len \le vals.length$:

obj	$= [vals[vals.length - 1]] \mathbin{+\!\!+} vals[0 .. len - 2]$	**by AF**
	$= [vals[vals.length - 1]] \mathbin{+\!\!+} vals[0 .. len_0 - 1]$	**since** $len = len_0 + 1$
	$= [vals[front]] \mathbin{+\!\!+} vals[0 .. len_0 - 1]$	**since** $front = \ldots$
	$= [x] \mathbin{+\!\!+} vals[0 .. len_0 - 1]$	**since** $x = vals[front]$
	$= [x] \mathbin{+\!\!+} obj_0$	**by AF**

Implementing Mutable Queue with One Array

```
// Implements a mutable queue using one array.
class ArrayQueue implements MutableNumberQueue {

  // Make sure there is room to add an element
  // @effects this.len < this.vals.length
  ensureEnoughSpace = (): void => { … };
```

- Function does not change the abstract state!
  - no `@modifies`
- This would be a private method
  - okay to mention “this” here