

CSE 331

Spring 2025

Abstraction

Jaela Field



xkcd #1172, ty matt

Administrivia

- **HW6 is out!**
 - **5 written, 2 coding questions**
Written problems build in difficulty.
 - **Start early!**
- **Katherine's Mon OH now Hybrid**

Calendar Updates/Reminders

- **Topic 7 & 8 switched**
- **Last HW (HW8)**
 - Due Wednesday (8/20), no usual 48-hour extension available
- **Currently 3 weeks out from final exam**
 - mostly short answer & multiple choice
 - will release a practice exam & last section is review

The Third Leg of the Class

- HW1–3: write more **realistic** applications
 - saw how **debugging** gets harder
- HW4–6: write code **correctly** the first time
 - checked correctness without a computer
- HW7–8: write more **complex** applications
 - most applications have a core, tricky part
 - use the **correctness toolkit** to get that right
 - can work faster where debugging is easier
 - only way to really know the UI is right is to try it

Procedural Abstraction

- **Hide the details of the function from the caller**
 - caller only needs to read the **specification**
 - (“procedure” means function)
- **Caller promises to pass valid inputs**
 - no promises on invalid inputs
- **Implementer then promises to return correct outputs**
 - does not matter how

Procedural Abstraction Example

- Specification of peachCipher is imperative:

```
// @returns keep(L) ++ skip(L)
export const peachCipher = (L: List): List => {
  // helper, calculates keep & skip in same pass
  return peachEncode(L, nil, nil);
};
```

- code implements a different function
- need to use reasoning to check that these two match

we proved, for all L by structural induction, that:

$$\text{concat}(\text{concat}(k, \text{keep}(L)), \text{concat}(s, \text{skip}(L))) = \text{peach-cipher}(L, k, s)$$

Other Properties of High-Quality Code

- Professionals are expected to write **high-quality** code
- Correctness is the most important part of quality
 - users **hate** products that do not work properly
- Also includes the following

- easy to change
- easy to understand
- modular

abstraction provides
all three properties

start with rev straight from the spec
later change it to a faster version

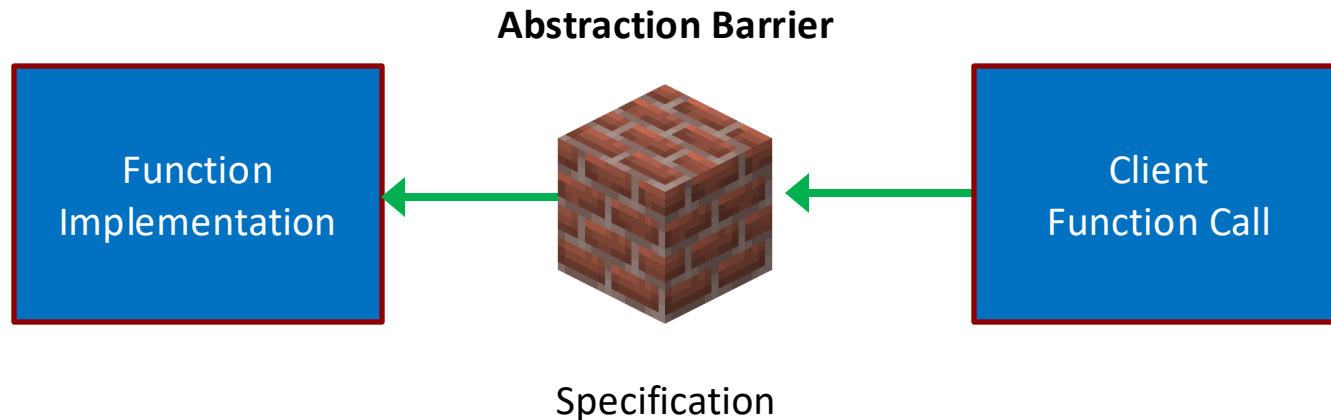
Benefits of Specifications

Clear specifications help with **understandability** and

- **Correctness**
 - reasoning requires clear definition of what the function does
- **Changeability**
 - implementer is free to write any code that meets spec
 - client can pass any inputs that satisfy requirements
- **Modularity**
 - people can work on different parts once specs are agreed

Abstraction Barrier

- Specification is an...



- specification is the “barrier” between the sides
- clients depend only on the spec
- implementer can write any code that satisfies the spec

Performance Improvements

- Rarely, we see faster *algorithms* for operations
 - Stay tuned for “tail recursion” topic!
- Most perf improvements change ***data structures***
 - different kind of abstraction barrier for data
- Let's see an example...

Last Element of a List

`last(nil)` `:= undefined`
`last(x :: nil)` `:= x`
`last(x :: y :: L)` `:= last(y :: L)`

- **Runs in $\Theta(n)$ time**
 - walks down to the end of the list
 - no faster way to do this on a list, *algorithmically*

- **We could cache the last element**
 - new data type just dropped:

analogous idea:
store references to both
“front” and “back” nodes

```
type FastLastList = {list: List, last: bigint | undefined}
```

empty list has undefined last

Defining Fast-Last List

```
type FastLastList = {list: List, last: bigint | undefined}
```

- **How do we switch to this type?**
 - change every `List` into `FastLastList`
- **Will still have functions that operate on List**
 - e.g., `len`, `sum`, `concat`, `rev`
- **Suppose `F` is a `FastLastList`**
 - instead of calling `rev(F)`, we have call `rev(F.list)`
 - cleaner to introduce a helper function

Implementing Fast-Last List Helpers

```
type FastLastList = {list: List, last: bigint | undefined}

const getLast = (F: FastLastList): bigint | undefined => {
  return F.last;
};

const toList = (F: FastLastList): List<bigint> => {
  return F.list;
};
```

- **How do we switch to this type?**
 - **change every** `List` **into** `FastLastList`
 - **replace** `F` **with** `toList(F)` **where a** `List` **is expected**

Another Fast List (1/2)

- Suppose we often need the 2nd to last, 3rd to last, ... (back of the list). How can we make it faster?
 - store the list in *reverse* order!

```
rev(nil)           := nil
rev(x :: L)        := rev(L) # (x :: nil)
```

```
// @returns rev(L)
const rev = (L: List< bigint>): List< bigint> => {
  if (L.kind === "nil") {
    return nil;
  } else {
    return concat(rev(L), cons(x, nil));
  }
};
```

Another Fast List (2/2)

- store the list in *reverse* order!

```
type FastBackList = List<bigint>;
```

```
const getLast = (F: FastBackList): bigint | undefined => {  
  return (F.kind === "nil") ? undefined : F.hd;  
};
```

```
const getSecondToLast = (F: FastBackList): bigint | undefined => {  
  return (F.kind === "nil") ? undefined :  
    (F.tl.kind === "nil") ? undefined : F.tl.hd;  
};
```

```
const toList = (F: FastBackList): List<bigint> => {  
  return rev(F);  
};
```

Another Fast List Gone Wrong

```
type FastBackList = List<bigint>;

const getLast = (F: FastBackList): bigint | undefined => {
  return (F.kind === "nil") ? undefined : F.hd;
};

const toList = (F: FastBackList): List<bigint> => {
  return rev(F);
};
```

- Problems with this solution...
 - no type errors if someone forgets to call `toList`!

```
const F: FastBackList = ...;
return concat(F, cons(1, nil)); // bad!
```


Yet Another Fast List?

```
type FastBackList =  
  {list: List<bigint>, origList: List<bigint>;  
  
  const getLast = (F: FastBackList): bigint | undefined => {  
    return (F.list.kind === "nil") ? undefined : F.list.hd;  
  };  
  
  const toList = (F: FastBackList): List<bigint> => {  
    return F.origList;  
  };
```

- Still some problems...
 - no type errors if someone grabs the field directly

```
const F: FastBackList = ...;  
return concat(F.list, cons(1, nil)); // bad!
```

Another Fast List — Take Three

```
const F: FastBackList = ...;  
return concat(F.list, cons(1, nil)); // bad!
```

- Only way to completely stop this is to hide `F.list`
 - do not give them the data, just the functions

```
type FastList = {  
  getLast: () => bigint|undefined,  
  toList: () => List<bigint>  
};
```

- the only way to get the list is to call `F.toList()`
- seems weird... but we can make it look familiar

Fast List as an Interface

```
interface FastList {  
  getLast(): bigint|undefined;  
  toList(): List<bigint>;  
}
```

- In TypeScript, “interface” is synonym for “record type”

- You’ve seen this in Java

Java interface is a record where
field values are functions (methods)

```
interface FastList {  
  int getLast() throws EmptyList;  
  List<Integer> toList();  
}
```

- in 331, our interfaces will only include functions (methods)

Data Abstraction

Data Abstraction & ADTs

- Give clients only operations, not data
 - operations are “public”, data is “private”
- We call this an Abstract Data Type (ADT)
 - invented by Barbara Liskov in the 1970s
 - fundamental concept in computer science
 - built into Java, JavaScript, etc.
 - data abstraction via procedural abstraction
- Critical for the properties we want
 - easier to change data structure
 - easier to understand (hides details)
 - more modular

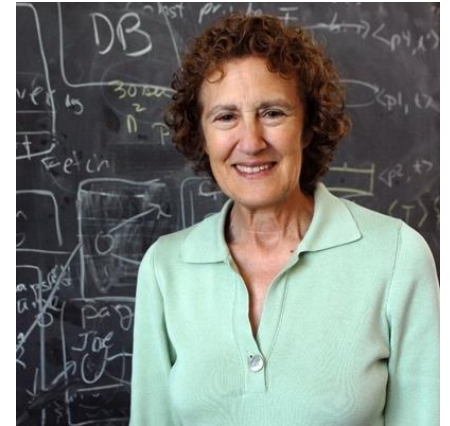


photo courtesy MIT

How to Make a FastList — Attempt One

```
const makeFastList = (list: List<bigint>): FastList => {  
  const last = last(list);  
  return {  
    getLast: () => { return last; },  
    toList: () => { return list; }  
  };  
};
```

- Values in `getLast` and `toList` fields are functions
- Note: `getLast` is *not* linear-time, but the constructor is!
- There is a cleaner way to do this
 - will also look more familiar

How to Make a FastList — As a Class (1/3)

```
class FastLastList implements FastList {  
  last: bigint | undefined; // should be "readonly"  
  list: List<bigint>;  
  
  constructor(list: List<bigint>) {  
    this.last = last(list);  
    this.list = list;  
  }  
  
  getLast = () => { return this.last; };  
  toList = () => { return this.list; };  
}
```

- Can create a new record using “**new**”
 - each record has fields `list`, `last`, `getLast`, `toList`
 - bodies of functions use “**this**” to refer to the record

How to Make a FastList — As a Class (2/3)

```
class FastLastList implements FastList {  
  last: bigint | undefined; // should be "readonly"  
  list: List<bigint>;  
  
  constructor(list: List<bigint>) {  
    this.last = last(list);  
    this.list = list;  
  }  
  
  getLast = () => { return this.last; };  
  toList = () => { return this.list; };  
}
```

- Can create a new record using “**new**”
 - all four assignments are executed on each call to “**new**”
 - `getLast` and `toList` are always the same functions

How to Make a FastList — As a Class (3/3)

```
class FastLastList implements FastList {  
  last: bigint | undefined; // should be "readonly"  
  list: List<bigint>;  
  
  constructor(list: List<bigint>) {  
    this.last = last(list);  
    this.list = list;  
  }  
  
  getLast = () => { return this.last; };  
  toList = () => { return this.list; };  
}
```

- Implements the FastList interface
 - i.e., it has the expected `getLast` and `toList` fields
 - (okay for records to have more fields than required)

Another Way to Make a FastList

```
class FastBackList implements FastList {
  original: List<bigint>;
  reversed: List<bigint>; // in reverse order

  constructor(list: List<bigint>) {
    this.original = list;
    this.reversed = rev(list);
  }

  getLast = () => {
    return (this.reversed.kind === "nil") ?
      undefined : this.reversed.hd;
  };

  toList = () => { return this.original; }
}
```

How Do Clients Get a FastList

```
const makeFastList = (list: List<bigint>): FastList => {  
  return new FastLastList(list);  
};
```

- **Export only FastList and makeFastList**
 - completely hides the data representation from clients
- **This is called a “factory function”**
 - another design pattern
 - can change implementations easily in the future
becomes FastBackList with a one-line change
- **Difficult to add to the list with this interface**
 - requires three calls: toList, cons, makeFastList

More Convenient Cons (via Interface)

```
interface FastList {  
  cons(x: bigint): FastList;  
  getLast(): bigint | undefined;  
  toList(): List<bigint>;  
};  
  
const makeFastList = (): FastList => {  
  return new FastBackList(nil);  
};
```

- New method `cons` returns list with `x` in front
 - example of a “producer” method (others are “observers”)
produces a new list for you
 - now, we only need to make an empty `FastList`
anything else can be built via `cons`

Re-using the Empty List (as a “Singleton”)

```
interface FastList {  
  cons(x: bigint): FastList;  
  getLast(): bigint | undefined;  
  toList(): List<bigint>;  
};  
  
const nilList: FastList = new FastBackList(nil);  
  
const makeFastList = (): FastList => {  
  return nilList;  
};
```

- No need to create a new object using “**new**” *every time*
 - can reuse the same instance
 - only possible since these are immutable!
 - example of the “singleton” **design pattern**

The 331 ADT Design Pattern

We will use the following **design pattern** for ADTs:

- “**interface**” used for defining ADTs
 - declares the methods available
- “**class**” used for implementing ADTs
 - defines the fields and methods
 - implements the ADT interface above
 - *not* exported! (~ private)
- Factory function used to create instances

Stick to regular functions for rest of the code!

Specifications for ADTs

How to Specifications for ADTs?

- Run into problems when we try to write specs
 - for example, what goes after `@return`?
 - don't want to say returns the `.list` field (or reverse of that)
 - we want to hide those details from clients

```
interface FastList {  
  /**  
    * Returns the last element of the list.  
    * @returns ??  
    */  
  getLast: () => bigint | undefined;  
};
```

- Need some terminology to clear up confusion

New ADT Terminology: States

New terminology for specifying ADTs

Concrete State / Representation

actual fields of the record and the data stored in them

Last example: `{list: List, last: bigint | undefined}`

Abstract State / Representation

how clients should *think* about the object

Last example: `List` (i.e., `nil` or `cons`)

- We've had different abstract and concrete types all along!
 - in our math, `List` is an inductive type (abstract)
 - in our code, `List` is a record (concrete)

List State: Concrete vs Abstract

Inductive types also differ in abstract / concrete states:

Concrete State / Representation

actual fields of the record and the data stored in them

Last example: `{kind:"nil"} | {kind:"cons", hd: bigint, tl: List}`

Abstract State / Representation

how clients should *think* about the object

Last example: `List` (i.e., `nil` or `cons`)

- Inductive types also use a **design pattern** to work in TypeScript
 - details are different than ADTs (e.g., no interfaces)

New ADT Terminology: “object” (or “obj”)

New terminology for specifying ADTs

Concrete State / Representation

actual fields of the record and the data stored in them

Last example: `{kind:"nil"} | {kind:"cons", hd: bigint, tl: List}`

Abstract State / Representation

how clients should *think* about the object

Last example: List (i.e., nil or cons)

- Term “**object**” (or “**obj**”) will refer to abstract state
 - “object” means mathematical object
 - “obj” is the mathematical value that the record represents³⁵

Specifying FastList & getLast with “obj”

```
/**
 * A list of integers that can retrieve the last
 * element in  $O(1)$  time.
 */
export interface FastList {
  /**
   * Returns the last element of the list ( $O(1)$  time).
   * @returns last(obj)
   */
  getLast(): bigint | undefined;
}
```

- “obj” refers to the abstract state (the list, in this case)
 - actual state will be a record with fields `last` and `list`

Specifying FastList & cons with “obj” (1/2)

```
/**
 * A list of integers that can retrieve the last
 * element in  $O(1)$  time.
 */
export interface FastList {
  ...
  /**
   * Returns a new list with  $x$  in front of this list.
   * @returns cons( $x$ , obj)
   */
  cons(x: bigint): FastList;
```

- **Producer method:** makes a new list for you
 - “obj” above is a list, so `cons(x, obj)` makes sense in math

Specifying FastList & cons with “obj” (2/2)

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
export interface FastList {
  ...
  /**
   * Returns a new list with x in front of this list.
   * @returns cons(x, obj)
   */
  cons(x: bigint): FastList;
```

- Specification does not talk about fields, just “obj”
 - fields are *hidden* from clients

Specifying FastList & toList with “obj” (1/2)

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
export interface FastList {
  ...
  /**
   * ??
   * @returns ??
   */
  toList(): List<bigint>;
}
```

- How do we specify this?

Specifying FastList & toList with “obj” (2/2)

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
export interface FastList {
  ...
  /**
   * Returns the object as a regular list of items.
   * @returns obj
   */
  toList(): List<bigint>;
}
```

- In math, this function does nothing (“@returns obj”) – two *different* concrete representations of the same idea – details of the representations are *hidden* from clients

(Internally)
Documenting an
ADT Implementation

Recall: Abstract State

- Key idea of a public ADT spec is the “**abstract state**”
 - simple definition of the object (easier to think about)
 - clients use that to **reason** about calls to this code
 - descriptions in terms of “**obj**”
- We also need to reason about ADT implementation
 - for this, we do want to talk about fields
 - fields are hidden from clients, but visible to implementers

Documenting ADT Impls: Abstraction Function

- We also need to document the ADT implementation
 - for this, we need two new tools

Abstraction Function

defines what abstract state the field values currently represent

- Maps the field values to the object they represent
 - object is math, so this is a *mathematical* function
 - there is no such function in the code — just a tool for reasoning
 - will usually write this as an *equation*
 - $\text{obj} = \dots$ right-hand side uses the fields

Example Abstraction Function: FastLastList

```
class FastLastList implements FastList {  
  // AF: obj = this.list  
  last: bigint | undefined;  
  list: List<bigint>;  
  ...  
}
```

- **Abstraction Function (AF) gives the abstract state**
 - obj = abstract state
 - this = concrete state (record with fields .last and .list)
 - AF relates abstract state to the current concrete state
 - okay that “last” is not involved here
 - specifications only talk about “obj”, not “this”
 - “this” will appear in our reasoning

Documenting ADT Impls: Representation Invariant

- We also need to document the ADT implementation
 - for this, we need two new tools

Abstraction Function

defines what abstract state the field values currently represent
only needs to be defined when RI is true

Representation Invariants (RI)

facts about the field values that should always be true
defines what field values are allowed
AF only needs to apply when RI is true

Example Representation Invariant: FastLastList

```
class FastLastList implements FastList {  
  // RI: this.last = last(this.list)  
  // AF: obj = this.list  
  last: bigint | undefined;  
  list: List<bigint>;  
  ...  
}
```

- **Representation Invariant (RI)** holds info about `this.last`
 - fields cannot have *just any* number and list of numbers
 - they must fit together by satisfying RI
 - last must be the last number in the list stored

Correctness of FastList Constructor: RI

```
class FastLastList implements FastList {  
  // RI: this.last = last(this.list)  
  // AF: obj = this.list  
  last: bigint | undefined;  
  list: List<bigint>;  
  
  constructor(L: List<bigint>) {  
    this.list = L;  
    this.last = last(this.list);  
  }  
  ...  
}
```

- Constructor must ensure that RI holds at end
 - we can see that it does in this case
 - since we **don't mutate**, they will *always* be true

Correctness of FastList Constructor: AF

```
class FastLastList implements FastList {  
  // RI: this.last = last(this.list)  
  // AF: obj = this.list  
  last: bigint | undefined;  
  list: List<bigint>;  
  
  // makes obj = L  
  constructor(L: List<bigint>) {  
    this.list = L;  
    this.last = last(this.list);  
  }  
}
```

- **Constructor must create the requested abstract state**
 - client wants obj to be the passed in list
 - we can see that $\text{obj} = \text{this.list} = L$

Correctness of getLast (1/2)

```
class FastLastList implements FastList {  
  // RI: this.last = last(this.list)  
  // AF: obj = this.list  
  
  ...  
  // @returns last(obj)  
  getLast = (): bigint | undefined => {  
    return this.last;  
  };  
}
```

- Use both RI and AF to check correctness

last(obj) =

Correctness of getLast (2/2)

```
class FastLastList implements FastList {  
  // RI: this.last = last(this.list)  
  // AF: obj = this.list  
  
  ...  
  // @returns last(obj)  
  getLast = (): bigint | undefined => {  
    return this.last;  
  };  
}
```

- Use both RI and AF to check correctness

last(obj)	= last(this.list)	by AF
	= this.last	by RI

Correctness of ADT implementation

- **Check that the constructor...**
 - creates a concrete state satisfying RI
 - creates the abstract state required by the spec
- **Check the correctness of each method...**
 - check value returned is the one stated by the spec
 - may need to use both RI and AF

Think, Pair, Share

```
class FastBackList implements FastList {  
    // RI: _____  
    // AF: _____  
    original: List<bigint>;  
    reversed: List<bigint>; // in reverse order  
    ...  
}
```

sli.do #cse331



- What should the RI and AF be?
 - RI describes facts that need to be true about fields
 - AF maps abstract (obj) to concrete state (fields)

Think, Pair, Share

```
class FastBackList implements FastList {  
    // RI: this.original = rev(this.reversed)  
    // AF: obj = this.original  
    original: List<bigint>;  
    reversed: List<bigint>; // in reverse order  
    ...  
}
```

Think, Pair, Share

```
class FastBackList implements FastList {  
    // RI: this.original = rev(this.reversed)  
    // AF: obj = this.original  
    original: List<bigint>;  
    reversed: List<bigint>; // in reverse order  
    constructor(L: List<bigint>) {  
        this.original = L;  
        this.reversed = rev(L);  
    }  
}
```

...

sli.do #cse331



- Prove our constructor correct
 - User wants L to be the list represented by this `FastBackList`
 - $L = \text{rev}(\text{rev}(L))$ (By induction, I promise!)

Think, Pair, Share

```
class FastBackList implements FastList {  
    // RI: this.original = rev(this.reversed)  
    // AF: obj = this.original  
    original: List<bigint>;  
    reversed: List<bigint>; // in reverse order  
    constructor(L: List<bigint>) {  
        this.original = L;  
        this.reversed = rev(L);  
    }  
}
```

...

rev(this.reversed) = rev(rev(L))	constructor
= L	because $L = \text{rev}(\text{rev}(L))^*$
= this.original	constructor

obj = this.original	AF
= L	constructor

ADTs: the Good and the Bad

- Provides data abstraction
 - can change data structures without breaking clients
- Comes at a cost
 - more work to specify and check correctness
- Not everything needs to be an ADT
 - don't be like Java and make everything a class
- Prefer concrete types for most things
 - concrete types are easier to think about
 - introduce ADTs when the first *change* occurs

Worked Example: Immutable Queues

Immutable Queue Interface

- A queue is a list that can *only* be changed two ways:
 - add elements to the front
 - remove elements from the back

```
// List that only supports adding to the front and  
// removing from the end
```

```
interface NumberQueue {
```

observer

```
    // @returns len(obj)  
    size(): bigint;
```

producer

```
    // @returns [x] ++ obj  
    enqueue(x: bigint): NumberQueue;
```

producer

```
    // @requires len(obj) > 0  
    // @returns (x, Q) with obj = Q ++ [x]  
    dequeue(): [bigint, NumberQueue];
```

```
}
```

Implementing a Queue with a List (“Easiest”)

```
// Implements a queue with a list.  
class ListQueue implements NumberQueue {  
  
    // AF: obj = this.items  
    items: List<bigint>;  
}
```

- Easiest implementation is concrete = abstract state
 - just store the abstract state in a field
- Still requires extra work to check correctness...
 - abstraction barrier comes with a cost

Implementing a Queue with a List: Size

```
// Implements a queue with a list.
class ListQueue implements NumberQueue {

    // AF: obj = this.items
    items: List<bigint>;

    // @returns len(obj)
    size = (): bigint => {
        return len(this.items);
    };
}
```

- **Correctness of** `size`:

`len(this.items) = len(obj)` **by AF**

nothing is "straight from the spec" anymore

Implementing a Queue with a List: Constructor

```
// Implements a queue with a list.
class ListQueue implements NumberQueue {

    // AF: obj = this.items
    items: List<bigint>;

    // makes obj = items
    constructor(items: List<bigint>) {
        this.items = items;
    }
}
```

- **Correctness of** `constructor`:

items = this.items
 = obj

(from code)

AF

Implementing a Queue with a List: Enqueue

```
// Implements a queue with a list.
class ListQueue implements NumberQueue {

  // AF: obj = this.items
  items: List<bigint>;

  // @returns [x] ++ obj
  enqueue = (x: bigint): NumberQueue => {
    return new ListQueue(cons(x, this.items));
  };
}
```

- **Correctness of enqueue:**

return value = $x :: \text{this.items}$
 = $x :: \text{obj}$
 = $[] \# (x :: \text{obj})$
 = $[x] \# \text{obj}$

spec of constructor
AF
def of concat
def of concat

Implementing a Queue with a List: Dequeue

```
// Implements a queue with a list.
class ListQueue implements NumberQueue {

    // AF: obj = this.items
    items: List<bigint>;

    // @requires len(obj) > 0
    // @returns (x, Q) with obj = Q ++ [x]
    dequeue = (): [bigint, NumberQueue] => {
        return [last(this.items),
                prefix(len(this.items) - 1n, this.items)];
    };
};
```

- Handwave: `prefix(n, L)` gives first `n` items of `L`
- Declarative spec, so more reasoning is required!
 - also, slower than necessary ($\Theta(n)$ dequeue)
 - we'll skip correctness here and do something faster in a moment...

Summary of `ListQueue`

- **Simplest possible implementation of ADT**
 - abstract state = concrete state of one field
- **Reasoning about every method is more complex**
 - must apply AF to relate return value to spec's postcondition
code uses fields, but postcondition uses "obj"
 - this is the cost of the abstraction barrier

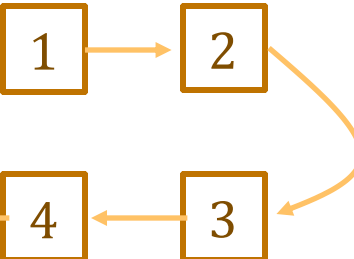
Implementing a Queue with Two Lists

```
// Implements a queue using two lists.  
class ListPairQueue implements NumberQueue {  
    // AF: obj = this.front ++ rev(this.back)  
    front: List<bigint>;  
    back: List<bigint>;    // in reverse order
```

- Back part stored in reverse order
 - head of front is the first element
 - head of back is the *last* element

this.front = 

this.back = 

obj = 

Two-Queue List: Representation Invariant (1/2)

```
// Implements a queue using two lists.  
class ListPairQueue implements NumberQueue {  
  
    // AF: obj = this.front ++ rev(this.back)  
    // RI: if this.back = nil, then this.front = nil  
    front: List<bigint>;  
    back: List<bigint>;  
}
```

- **Self-imposed RI:** If back is nil, then the queue is *empty*
 - if back = nil, then front = nil (by RI) and thus

obj =

Two-Queue List: Representation Invariant (2/2)

```
// Implements a queue using two lists.  
class ListPairQueue implements NumberQueue {  
  
    // AF: obj = this.front ++ rev(this.back)  
    // RI: if this.back = nil, then this.front = nil  
    front: List<bigint>;  
    back: List<bigint>;  
}
```

- **Self-imposed RI: If back is nil, then the queue is *empty***
 - if back = nil, then front = nil (by RI) and thus

obj = nil # rev(nil)	by AF
= rev(nil)	def of concat
= nil	def of rev

- if the queue is not empty, then back is not nil

Two-Queue List: Constructor (for now)

```
// Implements a queue using two lists.
class ListPairQueue implements NumberQueue {

    // AF: obj = this.front ++ rev(this.back)
    // RI: if this.back = nil, then this.front = nil
    front: List<bigint>;
    back: List<bigint>;

    // makes obj = front ++ rev(back)
    constructor(front: List<bigint>, back: List<bigint>) {
        ...
    }
}
```

- Will implement this later...

Two-Queue List: Size (1/2)

```
// AF: obj = this.front ++ rev(this.back)
front: List<bigint>;
back: List<bigint>;

// @returns len(obj)
size = (): bigint => {
    return len(this.front) + len(this.back);
};
```

- **Correctness of** `size`:

`len(obj) =`

Two-Queue List: Size (2/2)

```
// AF: obj = this.front ++ rev(this.back)
front: List<bigint>;
back: List<bigint>;

// @returns len(obj)
size = (): bigint => {
  return len(this.front) + len(this.back);
};
```

- **Correctness of** `size`:

$$\begin{aligned}\text{len}(\text{obj}) &= \text{len}(\text{this.front} \# \text{rev}(\text{this.back})) \\ &= \text{len}(\text{this.front}) + \text{len}(\text{rev}(\text{this.back})) \\ &= \text{len}(\text{this.front}) + \text{len}(\text{this.back})\end{aligned}$$

by AF
by earlier Ex.
by another
induction

Two-Queue List: Enqueue (1/2)

```
// AF: obj = this.front ++ rev(this.back)
front: List<bigint>;
back: List<bigint>;

// @returns [x] ++ obj
enqueue = (x: bigint): NumberQueue => {
  return new ListPairQueue(cons(x, this.front), this.back)
}
```

- **Correctness of** enqueue:

ret value =

Two-Queue List: Enqueue (2/2)

```
// AF: obj = this.front ++ rev(this.back)
front: List<bigint>;
back: List<bigint>;

// @returns [x] ++ obj
enqueue = (x: bigint): NumberQueue => {
  return new ListPairQueue(cons(x, this.front), this.back)
}
```

- **Correctness of enqueue:**

$$\begin{aligned}\text{ret value} &= (x :: \text{this.front}) \# \text{rev}(\text{this.back}) \\ &= x :: (\text{this.front} \# \text{rev}(\text{this.back})) \\ &= x :: \text{obj} \\ &= [] \# (x :: \text{obj}) \\ &= [x] \# \text{obj}\end{aligned}$$

spec of constructor
def of concat
AF
def of concat
def of concat

Two-Queue List: Dequeue (1/2)

```
// AF: obj = this.front ++ rev(this.back)
front: List<bigint>;
back: List<bigint>;

// @requires len(obj) > 0
// @returns (x, Q) with obj = Q ++ [x]
dequeue = (): [bigint, NumberQueue] => {
    return [this.back.hd,
            new ListPairQueue(this.front, this.back.tl)];
};
```

- as noted previously, precondition means $\text{this.back} \neq \text{nil}$
- as we know, this means $\text{this.back} = x :: L$
where $x = \text{this.back.hd}$ and some $L = \text{this.back.tl}$
- note that TypeScript would *not* allow this! why?
 - TypeScript can't read our preconditions :(

Two-Queue List: Dequeue (2/2)

```
// @requires len(obj) > 0
// @returns (x, Q) with obj = Q ++ [x]
dequeue = (): [bigint, NumberQueue] => {
  return [this.back.hd,
    new ListPairQueue(this.front, this.back.tl)];
};
```

– $\text{this.back} = x :: L$ **where** $x = \text{this.back.hd}$ **and some** $L = \text{this.back.tl}$

$\text{obj} = \text{this.front} \# \text{rev}(\text{this.back})$	by AF
$= \text{this.front} \# \text{rev}(x :: L)$	since $\text{back} = x :: L$
$= \text{this.front} \# (\text{rev}(L) \# [x])$	def of rev
$= (\text{this.front} \# \text{rev}(L)) \# [x]$	(list assoc.)
$= (\text{this.front} \# \text{rev}(L)) \# [\text{this.back.hd}]$	since $x = \text{this.back.hd}$
$= (\text{this.front} \# \text{rev}(\text{this.back.tl})) \# [\text{this.back.hd}]$	since $L = \text{this.back.tl}$

Two-Queue List: Constructor (1/3)

```
// AF: obj = this.front ++ rev(this.back)
// RI: if this.back = nil, then this.front = nil
front: List<bigint>;
back: List<bigint>;

// makes obj = front ++ rev(back)
constructor(front: List<bigint>, back: List<bigint>) {
  if (back.kind === "nil") {
    this.front = nil;
    this.back = rev(front);
  } else {
    this.front = front;
    this.back = back;
  }
}
```

**RI: this.front = nil
or this.back \neq nil**

holds since this.front = nil

holds since this.back \neq nil

- Need to check that RI holds at end of constructor

Two-Queue List: Constructor (2/3)

```
// AF: obj = this.front ++ rev(this.back)
// RI: if this.back = nil, then this.front = nil
front: List<bigint>;
back: List<bigint>;

// makes obj = front ++ rev(back)
constructor(front: List<bigint>, back: List<bigint>) {
  if (back.kind === "nil") {
    this.front = nil;
    this.back = rev(front);           obj = nil # rev(rev(front)) ??
  } else {
    this.front = front;               obj = front # rev(back)
    this.back = back;
  }
}
```

- Need to check this creates correct abstract state

Two-Queue List: Constructor (3/3)

```
// AF: obj = this.front ++ rev(this.back)
// RI: if this.back = nil, then this.front = nil
front: List<bigint>;
back: List<bigint>;

constructor(front: List<bigint>, back: List<bigint>) {
  if (back.kind == "nil") {
    this.front = nil;
    this.back = rev(front);
  } else {
    ...
  }
}
```

```
obj = nil # rev(rev(front))
    = nil # front
    = front
    = front # nil
    = front # rev(nil)
    = front # rev(back)
```

AF
because $L = \text{rev}(\text{rev}(L))^*$
def of concat

def of rev
since back = nil