CSE 331

Mutation and Data Abstraction

James Wilcox

with thanks to Kevin Zatloukal for many slides

331 So Far…

- Saw how to implement ADTs without mutation
- Introducing more mutation going forward
  - core idea is that mutation makes things harder
- Introduced **local variable** mutation last time
  - causes **some** difficulty for *implementers*

    need to reason line-by-line for any variable that is mutated
  - causes **no** difficulty for *clients*

    they literally cannot tell the difference

When we mutate objects and arrays...

- Objects and arrays are "heap" data
  - can still be in use after the call returns
- Mutation of heap data is different
  - clients can see that mutation occurred!
- So, we must also update specifications
  - need to explain any possible mutation that may happen

    by default, nothing is being mutated
  - higher likelihood of potential bugs

    miscommunication between programmers is a common cause
  - these will be harder to debug

Plan for today

Learn how to specify heap mutation for clients

1. Mutation in simple functions (revisit Topic 1)
2. Mutation in ADTs (revisit Topic 3)

Mutation of Arguments

Recall: Writing Method Specifications in Java

- Every input falls in one of three cases:
  1. input is disallowed
  2. input is allowed and will return something
  3. input is allowed and will throw something

- Item 1 is the precondition
  - explained in `@param` and `@requires`

- Items 2-3 are the postcondition
  - explained in `@return` and `@throws`

Writing Method Specifications in Java

- Every input falls in one of three cases:
  1. input is disallowed
  2. input is allowed and will return something
  3. input is allowed and will throw something

- The postcondition can also include mutation
  - client will see that something argument was changed
  - explained in `@modifies` and `@effects`

Describing Mutation in Specifications

- List anything that may change in `@modifies`
  - anything not listed is assumed **not modified**
  - no `@modifies` means nothing is mutated

- Results of the mutation listed in `@effects`
  - promises about the state when the call returns
  - no `@effects` means any change is possible

```
// @modifies A
// @effects all entries of A set to zero
void clear(int[] A)
```

Example 1

```
/**
 * Changes the first instance of v in A to w
 * @param A The list to look in. Must be non-null
 * @param v The value to look for
 * @param w The value to replace the first v with
 * @modifies A
 * @effects changes A[i] to w, where i is the
 *      smallest index with A[i] = v, and leaves
 *      A[j] unchanged for all j != i
 * @throws NotFound if no such index i exists
 */
void changeFirst(List<Integer> A, int v, int w)
```

Recall: Example 2

```
/**
 * Returns the concatenation of two lists.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null
 * @return A ++ B
 */
List<Integer> concat(
    List<Integer> A, List<Integer> B)
```

How would we change this to mutate instead?

Example 2

```
/**
 * Returns the concatenation of two lists.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null
 * @modifies A
 * @effects A = A_0 ++ B
 */
void concat(List<Integer> A, List<Integer> B)
```

We are now using Floyd logic in the spec!

What about a version that modifies B instead?

Is there any scenario where *both* arguments are modified?

Example 3

```
/**
 * Returns the number of common elements in both
 * A and B. Sorts A and B in the process.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null
 *
 *
 *
 *
 */
int commonElems(List<Integer> A, List<Integer> B)
```

How should we specify this?

Example 3

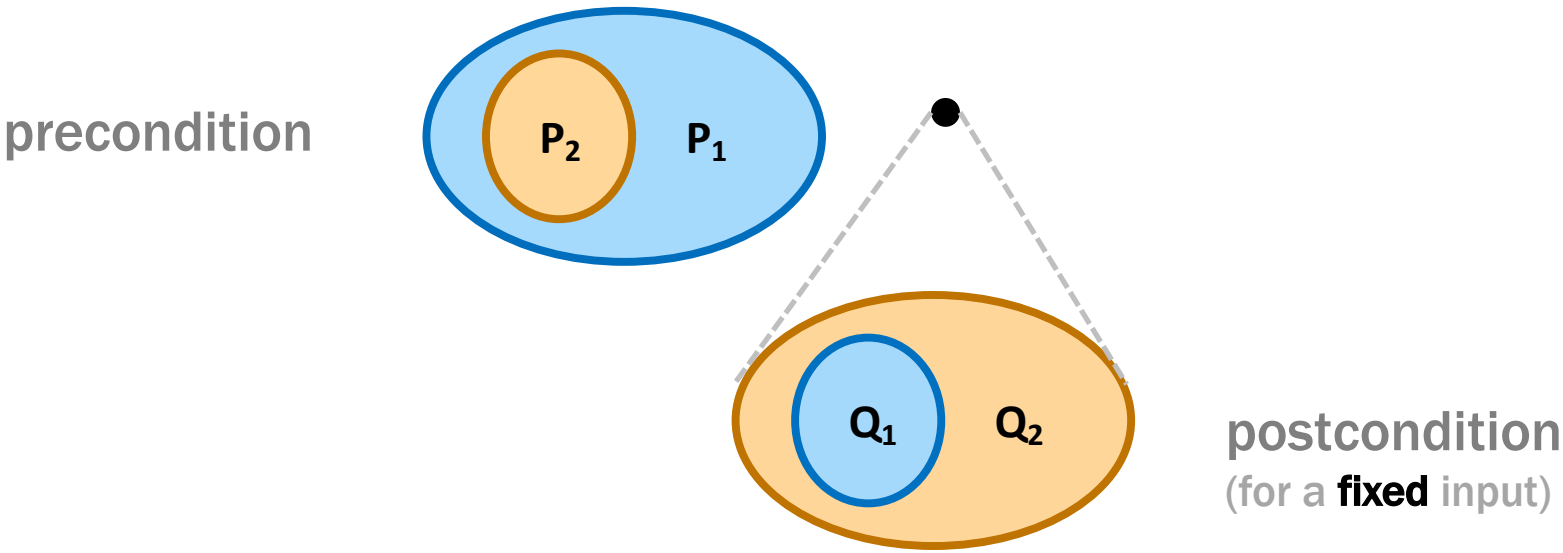
```
/**
 * Returns the number of common elements in both
 * A and B. Sorts A and B in the process.
 * @param A The first list. Must be non-null
 * @param B The second list. Must be non-null
 * @modifies A, B
 * @effects A is sorted and B is sorted
 * @returns the number of indexes i such that
 *       A[i] also appears in B somewhere
 */
int commonElems(List<Integer> A, List<Integer> B)
```

Recall: Comparing Specifications

- Specification S_1 is stronger than S_2...
  - whenever is S_1 satisfied, S_2 is also satisfied
  - i.e., satisfying S_1 implies satisfying S_2
- Changing from S_2 to S_1 (strengthening)...
  - cannot break any clients!
  - client works with any implementation satisfying S_2 and that includes anything satisfying S_1
- But what does this mean...
  - in terms of precondition and postcondition

Recall: Comparing Specifications

- Specification S_1 is stronger than S_2 if it has…
  - a weaker precondition and the same postcondition
  - a stronger postcondition and the same precondition
  - (or both)



Comparing Specifications With Mutation

- Specification S_1 is stronger than S_2 if it has…

- A stronger postcondition:
  - adds more to `@returns`
  - adds more to `@effects`
  - removes from `@modifies`

    promise is **not** to modify anything not listed

- A weaker precondition:
  - no change here

Example 4

```
int commonElems(List<Integer> A, List<Integer> B)

// Specification S1
// @modifies A, B
// @effects A is sorted and B is sorted
// @returns the number of indexes i such that
//      A[i] also appears in B somewhere

// Specification S2
// @modifies A, B
// @effects
// @returns the number of indexes i such that
//      A[i] also appears in B somewhere
```

How does S_1 relate to S_2?

Example 5

```
int commonElems(List<Integer> A, List<Integer> B)

// Specification S3
// @modifies A, B
// @effects A is sorted
// @returns the number of indexes i such that
//      A[i] also appears in B somewhere

// Specification S4
// @modifies A
// @effects A is sorted
// @returns the number of indexes i such that
//      A[i] also appears in B somewhere
```

How does S_3 relate to S_4?

Example 5

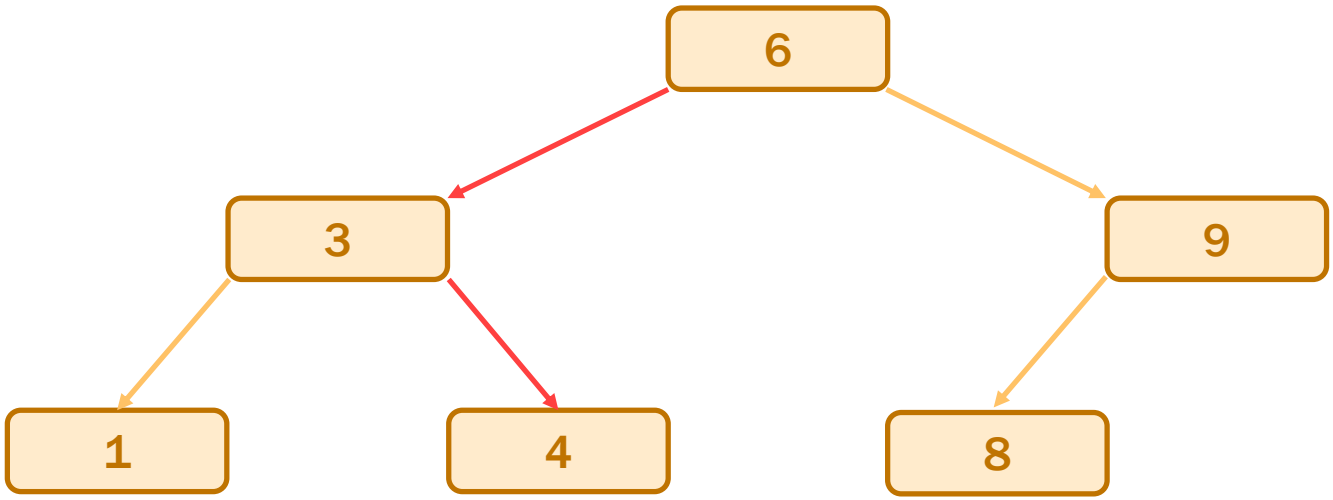
```
int commonElems(List<Integer> A, List<Integer> B)

// Specification S1
// @modifies A, B
// @effects A is sorted and B is sorted
// @returns the number of indexes i such that
//      A[i] also appears in B somewhere

// Specification S4
// @modifies A
// @effects A is sorted
// @returns the number of indexes i such that
//      A[i] also appears in B somewhere
```

How does S_1 relate to S_4?

Aliasing

Recall: Binary Search Trees

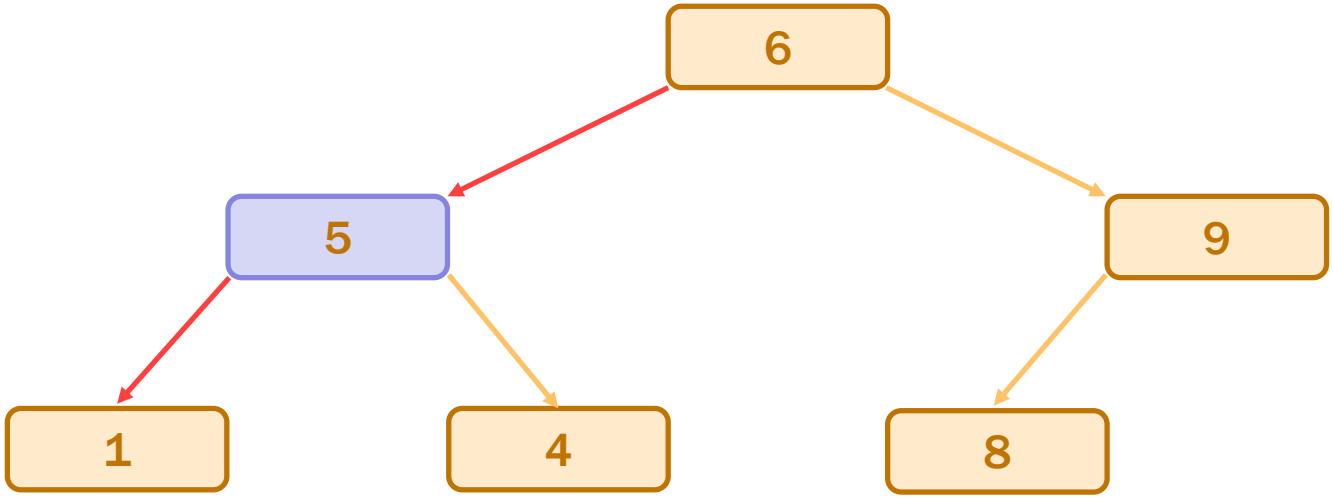
- Consider the following tree
  - searching for "4" proceeds as follows:



- Suppose someone changed "3" into "5"...

Recall: Binary Search Trees

- Suppose someone changed "3" into "5"...
  - now this happens when we search for "4":



  - It can no longer be found!

    Doesn't crash. It's just not found.
  - Problem doesn't occur on the line with the change

Scary Bugs

- Do not fear crashes
  - often no debugging at all

    get a stack trace that tells you exactly where it went wrong

- Do fear unexpected mutation
  - failure will give you no clue what went wrong

    will take a long time to realize the BST invariant was violated by mutation
  - bug could be almost anywhere in the code

    anyone who mutates a `TreeNode` could have caused it
  - could take *weeks* to track it down

Another Example

```
class Name {
  private String first;
  private String last;

  public String toString() {
    return first + " " + last;
   }

  public void capitalize() {
    this.first = first.substring(0, 1).toUpperCase()
        + first.substring(1);
    this.last = last.substring(0, 1).toUpp
        + last.substring(1);
 }
}
```

Somewhere else...

```
Map<Name, Integer> M;
```

Even Worse in C/C++

- C/C++ strings are mutable
  - commonly used as map keys
  - this sort of bug is still very common

- Java strings are immutable
  - was hugely controversial at the time

    in retrospect, it was clearly a good idea
  - other mutable types can still be used as keys

Aliases

- Extra references to an object are called "aliases"
  - possible for any reference type
- Aliases are fine when objects are *immutable*
  - we don't care if someone else reads the data
  - we only care if someone mutates it
- Aliases are scary when objects are **mutable**...
  - creates the potential for failures far from bugs
  - that means painful debugging

Mutable Heap State

- "With great power, comes great responsibility"
  - Uncle Ben

- With aliases to mutable heap state:
  - gain efficiency in some cases
  - must keep track of every alias that could mutate that state

    any alias, anywhere in the *entire* program could cause a bug

- EJ 17: minimize mutability in classes

Easy Ways to Stay Safe

1. Do not mutate heap state
   - don't need to think about aliasing at all
   - any number of aliases is fine

2. Do not allow aliases…
   - create the state in your constructor and **don't share** it

```
class MyClass {
  // RI: vals is sorted
  private String[] vals;

  public MyClass() {
    this.vals = new String[10];  // only reference
    …
  }
```

Easy Ways to Stay Safe

1. Do not use mutable state
   - don't need to think about aliasing at all
   - any number of aliases is fine

2. Do not allow aliases to *mutable* state
   a) do not hand out aliases yourself
   b) make a copy of anything you want to keep

ensures only one reference to the object (no aliases)

- For 331, mutable aliasing across files is a bug!
  - gives other parts the ability to break your code
  - we will stick to these simple strategies for avoiding it

An Advanced (Two-Stage) Approach

- Mutable object has only one reference (owner)
  - one reference that is allowed to use & mutate it
- Object is eventually "frozen", making it immutable
  - no longer necessary to track ownership
- Example: Java's `StringBuilder` vs `String`
  - `StringBuilder` is mutable (be careful!)
  - `StringBuilder.toString` returns the value as a `String`
  - `String` is immutable

Rules of Thumb

Client Side

1. Data is small
   - anything on screen is O(1)
2. Aliasing is common
   - UI design forces modules
   - data is widely shared

Rule: avoid mutation

- create new values instead
- performance will be fine

Server Side

1. Data is large
   - efficiency maters
2. Aliasing is avoidable
   - you decide on modules
   - data is not widely shared

Rule: avoid aliases

- do not allow aliases to your data
- hand out copies not aliases
- (good enough for us in 331)

Another Alternative

- With `List`, a third option is sometimes used:

```
public List<String> values() {
  return Collections.unmodifiableList(this.vals);
};
```

  - throws an exception when mutators are called
  - runs in $O(1)$ time instead of $O(n)$ to copy

Can this change break the client?

Another Alternative

- This can break clients
  - this works with a copy

```
MyClass m = …;
List<String> list = m.values()
list.add("another");
```

  - but not with `UnmodifiableList`

- Specification must make clear the behavior
  - how do the two options relate?

Another Alternative

- These two are incomparable
  - they have differing behavior
  - client can work with one but not the other and v.v.
- How is this possible when both return `List`?
  - the unmodifiable list does not implement `List`!

    the spec doesn't let you throw on any call to `add`
  - this is a terrible idea

    but occasionally necessary in extreme circumstances
- Really these are different return types
  - would be better to make then different interfaces

Unmodifiable View

- Unmodifiable list is a "view" of the underlying list
- It changes whenever the underlying list changes
  - updates to that list show up in the view immediately
  - it is not a copy of the data at that point
- This can lead to difficult bugs
  - do not use such a view as a key in a map
  - any alias to it can mutate it at any point

Unmodifiable View

- Why would someone do this?
- Like most CS bugs, it is for performance
  - we all know that $O(1)$ is better than $O(n)$
- But most client uses are $O(n)$ anyway!
  - client probably wants to loop through the list
  - in that case, there is no $O(..)$ gain to
- We will stick to immutable or copying (no aliases)

Calling Methods in Floyd Logic

Reasoning about Function Calls

- Not too difficult if the function is…
  1. defined for all inputs
  2. defined imperatively
  3. no arguments are mutated

Reasoning about Function Calls

- For the simplest case only…
- Forward reasoning rule is

$\{\{ P \}\}$

```
x = Math.sin(a);
```

$\{\{ P[x \mapsto x_0] \text{ and } x = \sin(a) \}\}$

```
// @param x
// @return sin(x)
double sin(double x)
```

- Backward reasoning rule is

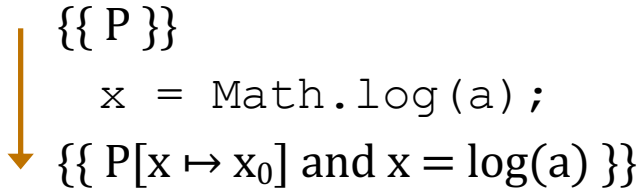
$\{\{ Q[x \mapsto \sin(a)] \}\}$

```
x = Math.sin(a);
```

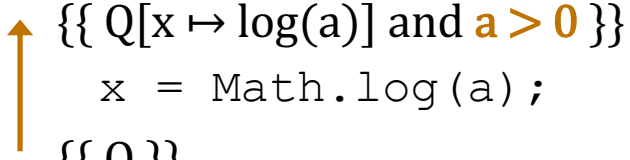
$\{\{ Q \}\}$

Reasoning about Function Calls

- Preconditions must be checked
  - not valid to call the function on disallowed inputs
- Forward reasoning rule is



- Backward reasoning rule is



Must also check $a > 0$

```
// @param x with x > 0
// @return log(x)
double log(double x)
```

Reasoning about Function Calls

- Applies to functions we define with imperative specs

```
// @param n a non-negative integer
// @returns square(n), where
//        square(0) := 0
//      square(n+1) := square(n) + 2n + 1
public int square(int n) {..}
```

- Reasoning is the same. E.g., forward rule is

{{ P }}

```
x = square(n);
```

{{ P[x ↦ x_0] and x = square(n) }}

Must also check that `n` is non-negative

Example 1: Forward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    double r = x + 2;
    {{ _____ }}
    r = Math.sqrt(r);
    {{ _____ }}
    r = r + 1;
    {{ _____ }}
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

Example 1: Forward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    double r = x + 2;
    {{ x ≥ 0 and r = x + 2 }}
    r = Math.sqrt(r);
    {{ _____ }}
    r = r + 1;
    {{ _____ }}
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

Example 1: Forward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    double r = x + 2;
    {{ x ≥ 0 and r = x + 2 }}
    r = Math.sqrt(r);
    {{ x ≥ 0 and r = √(x + 2) }}
    r = r + 1;
    {{ _____ }}
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

inverting operation gives $r^2 = x + 2$

c.f. to when $\mathrm{r} = r_o + 1$ and $\mathrm{P}(r_o)$

here we have $\mathrm{r} = \sqrt{r_o}$ and $r_o = x + 2$

second fact is *already* solved for r_o
so we can substitute right into left
instead of left into right

Example 1: Forward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    double r = x + 2;
    {{ x ≥ 0 and r = x + 2 }}
    r = Math.sqrt(r);
    {{ x ≥ 0 and r = √(x + 2) }}
    r = r + 1;
    {{ x ≥ 0 and r = √(x + 2) + 1 }}
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

$$\begin{aligned} r &= x + 2 \\ &\geq 0 + 2 \qquad \text{since } x \geq 0 \\ &\geq 0 \end{aligned}$$

this looks good

what did we forget?

Example 2: Backward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    {{ ____________ }}
    double r = x + 2;
    {{ ____________ }}
    r = Math.sqrt(r);
    {{ ____________ }}
    r = r + 1;
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

Example 2: Backward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    {{ ____________ }}
    double r = x + 2;
    {{ ____________ }}
    r = Math.sqrt(r);
    {{ r + 1 = √(x + 2) + 1 }}
    r = r + 1;
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

Example 2: Backward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
    {{ x ≥ 0 }}
    {{ ____________________ }}
    double r = x + 2;
    {{ √r + 1 = √(x + 2) + 1 and r ≥ 0 }}
    r = Math.sqrt(r);
    {{ r + 1 = √(x + 2) + 1 }}
    r = r + 1;
    {{ r = √(x + 2) + 1 }}
    return r;
}
```

Example 2: Backward

```
// @param x a positive number
// @return sqrt(x + 2) + 1
public double f(double x) {
```

$\{\{ x \geq 0 \}\}$

$\{\{ \sqrt{x+2} + 1 = \sqrt{x+2} + 1 \text{ and } x + 2 \geq 0 \}\}$

check this implication

```
    double r = x + 2;
```

$\{\{ \sqrt{r} + 1 = \sqrt{x+2} + 1 \text{ and } r \geq 0 \}\}$

```
    r = Math.sqrt(r);
```

$\{\{ r + 1 = \sqrt{x+2} + 1 \}\}$

```
    r = r + 1;
```

$\{\{ r = \sqrt{x+2} + 1 \}\}$

```
    return r;
}
```

$x \geq 0$ implies $x + 2 \geq 0$

Reasoning about ADT Calls

- ADT methods are calls involving abstract states
- Forward reasoning rule is

$\{\{ P \}\}$

```
L = L.cons(x);
```

$\{\{ P[L \mapsto L_0] \text{ and } L = x :: L_0 \}\}$

```
// @return x :: obj
FastList cons(int x)
```

L is a mathematical list

- Backward reasoning rule is

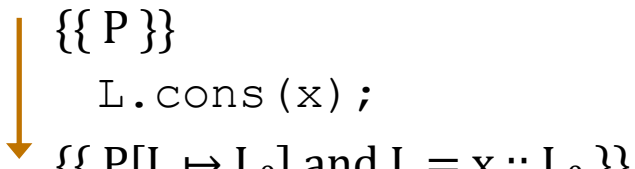
$\{\{ Q[L \mapsto x :: L] \}\}$

```
L = L.cons(x);
```

$\{\{ Q \}\}$

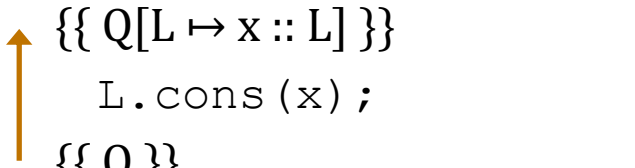
Reasoning about ADT Calls

- Very little changes with mutators…
- Forward reasoning rule is




```
// @modifies obj
// @effects obj = x :: obj0
void cons(int x)
```

- Backward reasoning rule is



`@effects` says it is an assignment
so we get an identical result

Example Calls with Declarative Specs

```
// @returns x such that x >= a and x >= b
public int max(int a, int b) {..}
```

- Forward reasoning rule is

{{ P }}
  x = **max**(a, b);
{{ P[x ↦ x_0] and x ≥ a and x ≥ b }}

- Backward reasoning rule is

{{ a > 0 }}
  x = **max**(a, b);
{{ a > 0 and 2x ≥ a + b }}

Must check that x ≥ a and x ≥ b **implies** 2x ≥ a + b

Function Calls with Declarative Specs

```
// @requires P2                  -- preconditions a, b
// @returns x such that R -- conditions on a, b, x
public int f(int a, int b) {..}
```

- **Forward reasoning rule is**

{{ P }}
  x = f(a, b);
{{ P[x ↦ x_0] and R }}

Must also check that P implies P_2

- **Backward reasoning rule is**

{{ Q_1 and P_2 }}
  x = f(a, b);
{{ Q_1 and Q_2 }}

Must also check that R implies Q_2

Q_2 is the part of postcondition using "x"

Recall: 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

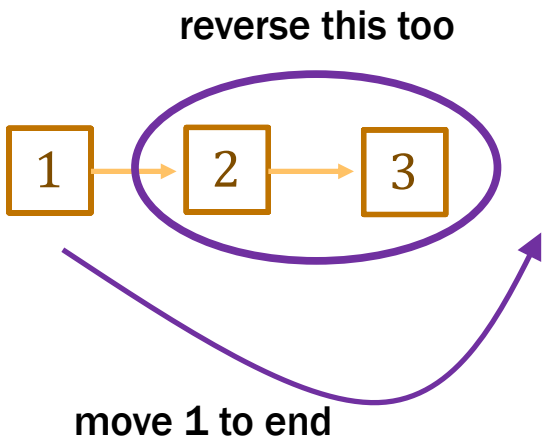
Recall: Procedural Abstraction

- 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

Definition of List Reversal

- **Recursive definition of** $\mathrm{rev}(S)$

$$\begin{aligned} \mathrm{rev}(\mathrm{nil}) &:= \mathrm{nil} \\ \mathrm{rev}(x :: L) &:= \mathrm{rev}(L) +\!\!+ [x] \end{aligned}$$



Procedural Abstraction Example

- **Specification of rev is imperative:**

```
// @return same numbers but in reverse order
public static List rev(List L) {
  return rev_acc(L, nil);  // faster way
}
```

  - code implements a different function
  - second version is $O(n)$ instead of $O(n^2)$
  - need to use reasoning to check that these two match

    can prove that rev_acc(L, nil) = rev(L) for all L by structural induction

Performance Improvements

- Faster algorithm, rev-acc, for reversing a list
  - rare to see this
- Most perf improvements change ***data structures***
  - different kind of abstraction barrier for data
- Let's see an example...

Last Element of a List

$$\begin{aligned} \text{last}(x :: \text{nil}) &:= x \\ \text{last}(x :: y :: L) &:= \text{last}(y :: L) \end{aligned}$$

last is undefined on nil

- Runs in $\Theta(n)$ time
  - walks down to the end of the list
  - no faster way to do this on a list
- How could we change data to make this faster?
  - we could cache the last element!

    analogous idea: store references to front and **back** nodes
  - can declare this in math as:

    type FastLastList := {last: $\mathbb{Z}$, list: List}
  - can do this in Java as well…

Fast-Last List

```
class FastLastList {
  private final int last;
  private final List list;

  FastLastList(List list) {
    this.list = list;
    this.last = last(list);
  }

  public int getLast() {
    return this.last;
  }

  public List getList() {
    return this.list;
  }
}
```

lots of real-world performance improvements look like this

in what way is this worse than just using `List`?

— less memory efficient

Fast-Last List

```
class FastLastList {
  private final int last;
  private final List list;

  public int getLast() { … }

  public List getList() { … }
}
```

- How do we switch to this type?
  - change every `List` into `FastLastList`
  - not truly hiding data structure changes yet…

Fast-Last List

```
class FastLastList {
  private final int last;
  private final List list;

  public int getLast() { … }

  public List getList() { … }
}
```

- What if we also want the second-to-last element?
  - this is $\Theta(n)$ to retrieve it
  - could cache the second to last element also
- What if we just want the items at the end of the list?
  - store it in **reverse** order!

Reversed List

```
private List L;  // regular order
private List R;  // reversed

/** @param L a list. Must be in *reverse* order */
public static void f(List L) { … }
```

- Why is this a terrible idea?
  - the type checker will not catch mistakes

    humans make mistakes. count on it
  - will unit tests catch this?

    might only show up in integration tests
  - what will the debugging be like?

Fast-Last List

```
class FastBackList {
  private final List revList;

  public int getLast() {
    return this.revList.hd;
  }

  public List getList() {
    return rev(this.revList);
  }
}
```

How could we avoid having a linear-time `getList`?

— keep `list` and `revList`

— time / space tradeoff

- How do we switch to this type?
  - change every `FastLastList` into `FastBackList`
  - not good that every data structure change is a code change

Fast-Last List

```
class FastLastList {
  private final int last;
  private final List list;
  …
}

class FastBackList {
  private final Last revList;
  …
}
```

- How can we make data structure changes without having to change all the clients every time?
  - use a Java `interface`

Fast List

```
interface FastList {
  int getLast();
  List getList();
}

class FastLastList implements FastList {
  private final int last;
  private final List list;
  …
}

class FastBackList implements FastList {
  private final List revList;
  …
}
```

Fast List

```
interface FastList {
  int getLast();
  List getList();
}
```

Interfaces contain only operations (methods) not data structures

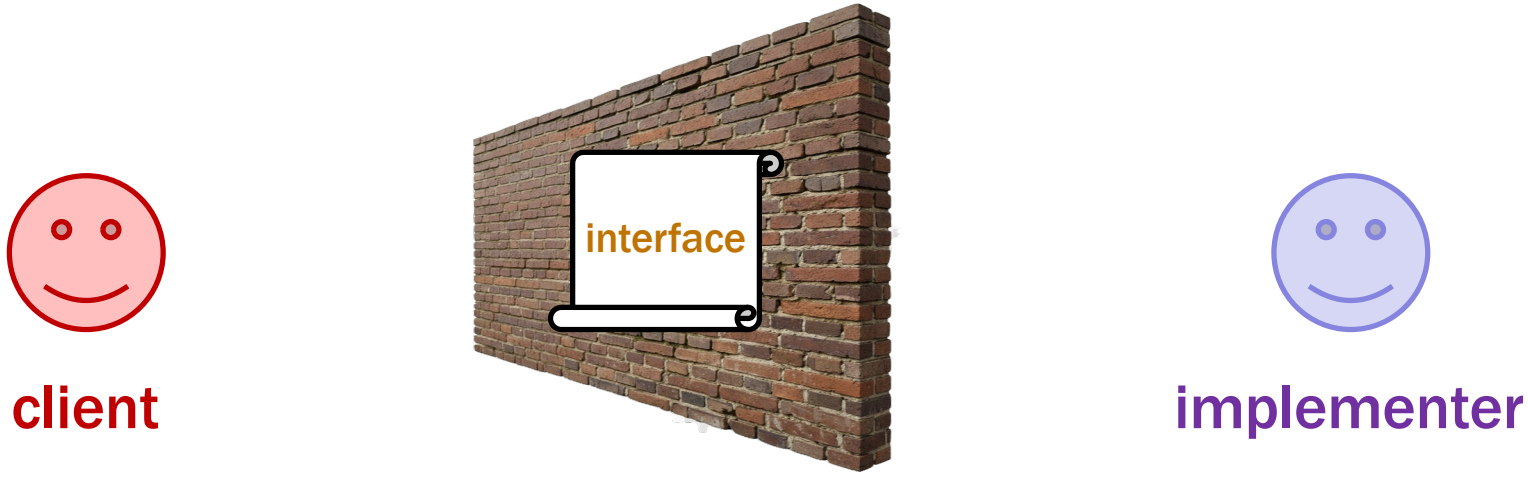
- Clients should use only `FastList` in declarations
  - only usen `FastLastList` after "`new`"
- Standard style in Java
  - e.g., create an `ArrayList` but store it as `List`

Example 1: Fast List

```
interface FastList {
  int getLast();
  List getList();
}
```

Interfaces contain only operations (methods) not data structures

- Interface provides an abstraction barrier
  - hides the details of data structures from the client



Data Abstraction

- Give clients only operations, not data
  - operations are the spec, data is hidden

- We call this an Abstract Data Type (ADT)
  - invented by Barbara Liskov in the 1970s

    won the Turing award in 2008
  - fundamental concept in computer science

    built into Java's `public` and `private`
  - data abstraction via procedural abstraction

- See recent interview with Computing History Museum

Example 2: Point in 2D Space

```
// Represents an (x, y) point in 2D space.
class Point {
  private double x;
  private double y;

  public double getX() { return this.x; }
  public double getY() { return this.y; }

  public double getR() {
    return Math.sqrt(x*x + y*y);
  }

  public double getTheta() {
    return Math.atan2(y, x);
  }
```

What if these are too slow?

— store polar coordinates

— time / space tradeoff

How do we write this as an ADT?

Example 2: Point in 2D Space

```
// Represents an (x, y) point in 2D space.
interface Point {
  double getX();
  double getY();
  double getR();
  double getTheta();
}

class SimplePoint implements Point {
  private double x, y;
}

class PolarPoint implements Point {
  private double x, y;
  private double r, theta;
}
```

What if we want to use `PolarPoint` for users with lots of memory and `SimplePoint` for users with little?

— need code to decide which to make

Example 2: Point in 2D Space

```
/** Creates a point at the given coordinates. */
public static Point makePoint(double x, double y) {
  if (Runtime.getRuntime().totalMemory() > MIN_MEM) {
    return new PolarPoint(x, y);
  } else {
    return new SimplePoint(x, y);
  }
}
```

- This is a "factory function"
  - an example of a design pattern

    more on these later…
  - Java SDK includes many, e.g., `Arrays.asList(..)`

Types of Operations

```
interface Point {
  public double getX();
  public double getY();
  public double getR();
  public double getTheta();
  …
}
```

- Operations usually fall into a few classes:
  - **observers/getters**: return properties of an object
  - **mutators**: change properties of the object
  - **producers**: create new objects from an existing one

Observers/Getters on Point

```
public double getX();
public double getY();
public double getR();
public double getTheta();

/** Return distance from this point to given one. */
public double distTo(Point p);
```

- Observers/Getters return information about the object, but do not change it in any way

Mutators on Point

```
/** Move the point by dx in x coord and dy in y */
public void shift(double dx, double dy);

/** Move by rotating about the origin by theta. */
public void rotate(double theta);
```

- Mutators change the properties of the object, but usually do not return anything important
  - also sometimes called “setters”

Producers on Point

```
/** Move the point by dx in x coord and dy in y */
public Point shift(double dx, double dy);

/** Move by rotating about the origin by theta. */
public Point rotate(double theta);
```

- Producers create new objects
  - ***new*** object is returned
  - they ***do not*** change the existing object

Mutable vs Immutable ADTs

	Immutable	Mutable
observers	✅	✅
mutators	❌	✅
producers	✅	❌ (usually not)

- Sensible to pick one or the other
  - would be dangerous to provide both

    will see why later on

- We will stick to immutable ADTs for now
  - mutation makes reasoning & debugging harder
  - **EJ 17**: minimize mutability in classes

Recall: Fast List

```
interface FastList {
  int getLast();
  List getList();
}
```

- What kind of operations are these?
  - both are observers

- What would be examples of useful producers?

Producers on FastList

```
/** Returns the list with x added at the front. */
public FastList cons(int x);
```

- What factory functions should we provide?
  - what is the minimum we could get away with?

Creator of FastLists

```
/** @return nil. */
public static FastList emptyList() {
  return new FastBackList(null);
}
```

- How could we make this more memory efficient?
  - no need to create a new object every time

Creator of FastLists

```
public static FastList EMPTY_LIST =
    new FastBackList(null);

/** @return nil. */
public static FastList emptyList() {
  return EMPTY_LIST;
}
```

- This is the "singleton" design pattern
  - **note: this is only possible since `FastList` is immutable!**

    we will see why later on…

Producers on FastList

```
/** Returns the list with x added at the front. */
public FastList cons(int x);

/**
 * Returns the list containing the elements of
 * this list followed by those of R.
 */
public FastList concat(FastList R);
```

- How do we formalize this?
  - everything above is English

    there is possibility for confusion

Specifications for ADTs

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

    we want to hide those details from clients

```
interface FastList {
   /**
    * Returns the last element of the list.
    * @return ??
    */
   int getLast();
};
```

- **Need some terminology to clear up confusion**

ADT Terminology

New terminology for specifying ADTs

Concrete State

actual fields of the record and the data stored in them

Last example: `int last, List list`

Abstract State

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)**

ADT Terminology

New terminology for specifying ADTs

Concrete State

actual fields of the record and the data stored in them

Last example: `int last, List list`

Abstract State

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 represents

Specifying FastList

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
interface FastList {
  /**
   * Returns the last element of the list (O(1) time)
   * @requires obj != nil
   * @return last(obj)
   */
  int getLast();
```

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

Specifying FastList

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

- **Producer method: makes a new list for you**
  - "obj" **above is a list, so** x :: obj **makes sense in math**

Specifying FastList

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

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

Specifying FastList

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
interface FastList {
  …
  /**
   * Returns the object as a regular list of items.
   * @return ??
   */
  List getList();
```

- How do we specify this?

Specifying FastList

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
interface FastList {
  …
  /**
   * Returns the object as a regular list of items.
   * @return obj
   */
  List getList();
```

- In math, this function does nothing ("`@return obj`")
  - two *different* concrete representations of the same idea
  - details of the representations are ***hidden*** from clients

Specifying Point

```
/** Represents an (x, y) point in 2D space. */
interface Point {

  /** @return x */
  double getX();

  /** @return y */
  double getY();
```

- Abstract state *is* a pair (x, y)
  - i.e., we have $(x, y) := obj$
  - so, we can refer to "x" and "y"

Specifying Point

```
/** Represents an (x, y) point in 2D space. */
interface Point {

  /** @return (x^2 + y^2)^(1/2) */
  double getR();

  /** @return arctan(y/x) */
  double getTheta();
```

- Imperative specifications
  - code may or may not actually do these calculations
  - `PolarPoint` just returns the value in a field

Specifying Point

```
/** Represents an (x, y) point in 2D space. */
interface Point {

  /** @return (x + dx, y + dy) */
  Point shiftBy(double dx, double dy);
```

- Describe the *abstract state* of what is returned
  - actual value returned is a `Point` of some variety

Recall: ADTs

- Abstraction over data
  - hide the details of the data representation
  - only give users a set of operations (the interface)

    data abstraction via procedural abstraction

- Interface can make clever data structures possible

- Some commonly used ADTs:
  - stack: add & remove from one end
  - queue: add to one end, remove from other
  - set: add, remove, & check if contained in list
  - map: add, remove, & get value for (key, value) pair

Immutable Queue

- 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 {

  // @return len(obj)
  int size();

  // @return x :: obj
  NumberQueue enqueue(int x);

  // @requires len(obj) > 0
  // @return (x, Q) with obj = Q ++ [x]
  DequeueParts dequeue();
}
```

observer

producer

producer

```
class DequeueParts {
  public final NumberQueue Q;
  public final int x;
}
```

Documenting an ADT Implementation

Documenting an ADT Implementation

- Last lecture, we saw how to write an ADT spec
- Key idea is the "abstract state"
  - simple definition of the object (easier to think about)
  - clients use that to reason about calls to this code
- Write specifications in terms of the abstract state
  - describe the return value 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 an ADT Implementation

- 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*

    $\text{obj} = \ldots$ **right-hand side uses the fields**

Documenting the FastList ADT

```
class FastLastList implements FastList {
  // AF: obj = this.list
  private final int last;
  private final List list;
  …
}
```

- **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 an ADT Implementation

- 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

Documenting the FastLastList ADT

```
class FastLastList implements FastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  private final int last;
  private final List list;
  …
}
```

- **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

Documenting the FastBackList ADT

```
class FastBackList implements FastList {


  private final List revList;
  …
}
```

- How can we specify this?
  - what is the AF and RI?

Documenting the FastBackList ADT

```
class FastBackList implements FastList {

  // AF: obj = rev(this.revList)
  private final List revList;
  …
}
```

- No need for an RI
  - any value for `revList` is fine

Documenting the FastBackList ADT

```
class FastBackList implements FastList {


  private final List list;
  private final List revList;
  …
}
```

- How can we specify this version?
  - what is the AF and RI?

Documenting the FastBackList ADT

```
class FastBackList implements FastList {
  // RI: this.revList = rev(this.list)
  // AF: obj = this.list
  private final List list;
  private final List revList;
  …
}
```

- Complexity moves from the AF to the RI

Documenting the PolarPoint ADT

```
/** Represents an (x, y) point in 2D space. */
class PolarPoint implements Point {


  private final double r, theta;
  …
}
```

- How can we specify this version of `Point`?
  - what is the AF and RI?

Documenting the PolarPoint ADT

```
/** Represents an (x, y) point in 2D space. */
class PolarPoint implements Point {
  // RI: r >= 0
  // AF: (r cos(theta), r sin(theta))
  private final double r, theta;
  …
}
```

- No constraints on theta
  - could restrict it by $-\pi < \theta \leq \pi$ (for example)

Documenting the PolarPoint ADT

```
/** Represents an (x, y) point in 2D space. */
class PolarPoint implements Point {


  private final double x, y;
  private final double r, theta;
  …
}
```

- **How can we specify this version of `Point`?**
  - **what is the AF and RI?**

Documenting the PolarPoint ADT

```
/** Represents an (x, y) point in 2D space. */
class PolarPoint implements Point {
  // RI: r = sqrt(x^2 + y^2) and theta = atan2(y, x)
  // AF: obj = (x, y)
  private final double x, y;
  private final double r, theta;
  …
}
```

- Complexity moves from AF to RI

Recall: Immutable Queue

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

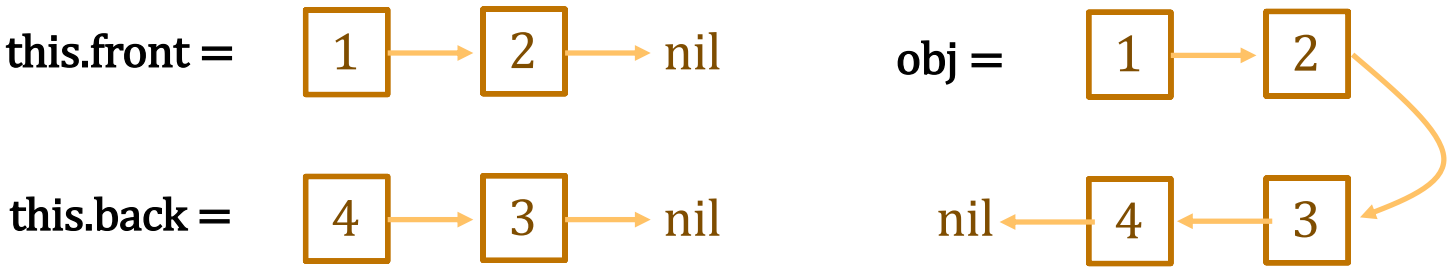
```
// List that only supports adding to the front and
// removing from the end
interface NumberQueue {

  // @return len(obj)
  int size();

  // @return [x] :: obj
  NumberQueue enqueue(int x);

  // @requires len(obj) > 0
  // @return (x, Q) with obj = Q ++ [x]
  DequeueParts dequeue();
}
```

```
class DequeueParts {
  public final NumberQueue Q;
  public final int x;
}
```

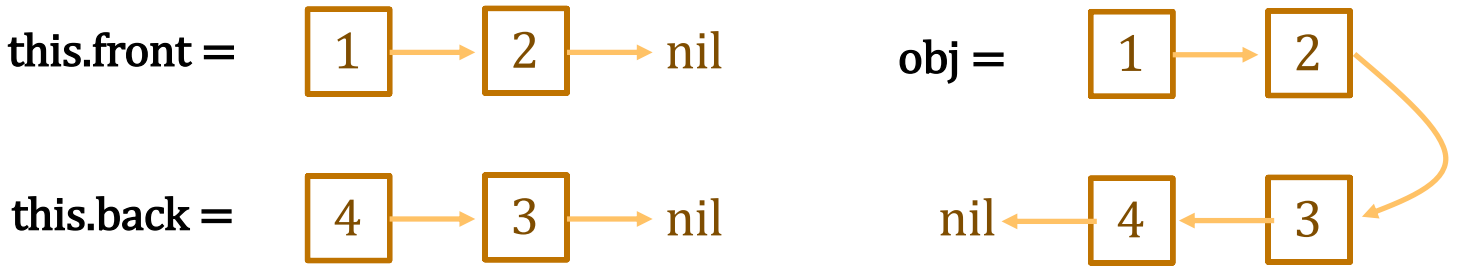
Implementing a Queue with Two Lists

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

  // AF: obj = this.front ++ rev(this.back)
  private final List front;
  private final List back;   // in reverse order
```

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



Implementing a Queue with Two Lists



- How do we enqueue (add at the front)?
  - remember that this is a producer
- How do we dequeue (remove from end)?
  - when is this not easy?
  - can we make this problem go away?

Implementing a Queue with Two Lists

```
// 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
  private final List front;
  private final List back;   // in reverse order
```

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

    obj =

Implementing a Queue with Two Lists

```
// 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
  private final List front;
  private final List back;   // in reverse order
```

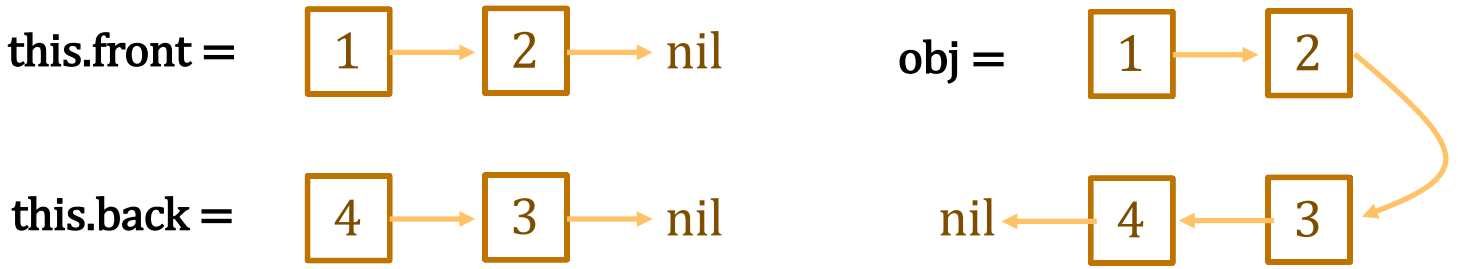
- **If back is nil, then the queue is *empty***
  - **if** back = nil, **then** front = nil **(by RI) and thus**

$$\begin{aligned} \text{obj} &= \text{nil} +\!\!+ \text{rev}(\text{nil}) && \textbf{by AF} \\ &= \text{rev}(\text{nil}) && \textbf{def of } \text{concat} \\ &= \text{nil} && \textbf{def of } \text{rev} \end{aligned}$$

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

    (**311 alert**: this is the contrapositive)

Implementing a Queue with Two Lists



- How do we dequeue (remove from end)?
  - What's different with our new RI?
- Easier to observe (find the last element), but harder to produce (the new queue)
  - i.e., easier to read but harder to write

CSE 331

Mutation of Heap State

James Wilcox and Kevin Zatloukal

Mutation in ADTs

Recall: Mutable vs Immutable ADTs

	Immutable	Mutable
observers	✅	✅
mutators	❌	✅
producers	✅	❌ (usually not)

- **Sensible to pick one or the other**
  - **would be dangerous to provide both**

    will see why later on

Recall: Specifying FastList

```
/**
 * A list of integers that can retrieve the last
 * element in O(1) time.
 */
interface FastList {

  // Returns the last element of the list (O(1) time)
  // @requires obj != nil
  // @return last(obj)
  int getLast();

  // Returns the object as a regular list of items.
  // @return obj
  List getList();
```

observer

observer

Recall: Specifying FastList

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

producer

- How do we make this a mutator?

Specifying a Mutable FastList

```
/**
 * A mutable list of integers that can retrieve the
 * last element in O(1) time.
 */
interface MutableFastList {
  …
  /**
   * Adds x to the front of this list.
   * @modifies obj
   * @effects obj = x :: obj_0
   */
  void cons(int x);
```

- Changes obj to have x at the beginning

Recall: Specifying Point

```
/** Represents an (x, y) point in 2D space. */
interface Point {

  /** @return x */
  double getX();

  /** @return y */
  double getY();
```

- Abstract state *is* a pair (x, y)
  - i.e., we have $(x, y) := obj$
  - so, we can refer to "x" and "y"

Recall: Specifying Point

```
/** Represents an (x, y) point in 2D space. */
interface Point {

  /** @return (x^2 + y^2)^(1/2) */
  double getR();

  /** @return arctan(y/x) */
  double getTheta();
```

- Imperative specifications
  - code may or may not actually do these calculations
  - `PolarPoint` just returns the value in a field

Recall: Specifying Point

```
/** Represents an (x, y) point in 2D space. */
interface Point {

  /** @return (x + dx, y + dy) */
  Point shiftBy(double dx, double dy);
```

- How do we make this a mutator?

Specifying a Mutable Point

```
/** Represents a mutable (x, y) point in 2D space. */
interface MutablePoint {

  /**
   * Moves the point right by dx and up by dy
   * @modifies obj
   * @effects obj = (x_0 + dx, y_0 + dy)
   */
  void shiftBy(double dx, double dy);
```

Recall: Immutable Queue

- 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 {

  // @return len(obj)
  int size();

  // @return [x] ++ obj
  NumberQueue enqueue(int x);

  // @requires len(obj) > 0
  // @return (x, Q) with obj = Q ++ [x]
  DequeueParts dequeue();
}
```

Which method(s) change in a mutable version?

```
class DequeueParts {
  public final NumberQueue Q;
  public final int x;
}
```

Mutable Queue

```
// @return [x] ++ obj
NumberQueue enqueue(int x);
```

- How do we make this mutable?

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

Mutable Queue

```
// @requires len(obj) > 0
// @return (x, Q) with obj = Q ++ [x]
DequeueParts dequeue();
```

- How do we make this mutable?

```
// @modifies obj
// @effects obj_0 = obj ++ [x]
// @return x
int dequeue();
```

Mutable Queue

- Note the symmetry between these operations:

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

// @modifies obj
// @effects obj_0 = obj ++ [x]
// @return x
int dequeue();
```

Which one of these is declarative?

Converting Between Mutators and Producers

- We can transform between these in general
  - assume that "`T`" is our interface

```
// @return f(obj, x)
T produce(int x);
```

1. change return type
2. change `@return` expression into `@effects obj = expression`

```
// @modifies obj
// @effects obj = f(obj_0, x)
void mutate(int x);
```

CSE 331

Reasoning about ADTs

James Wilcox and Kevin Zatloukal

Clients of ADTs

Correct ADT Implementation

Recall: Documenting the FastList ADT

```
class FastLastList implements FastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  private int last;
  private List list;
  …
}
```

- 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

```
class FastLastList implements FastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  private int last;
  private List list;

  FastLastList(List L) {
    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

```
class FastLastList implements FastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  private int last;
  private List list;

  // @effects obj = L
  FastLastList(List L) {
    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 $\text{obj} = \text{this.list} = \text{L}$

Correctness of getLast

```
class FastLastList implements FastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  …
  // @return last(obj)
  public int getLast() {
    return this.last;
  };
}
```

- **Use both RI and AF to check correctness**

last(obj) =

Correctness of getLast

```
class FastLastList implements FastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  …
  // @return last(obj)
  public int getLast() {
    return this.last;
  };
}
```

- **Use both RI and AF to check correctness**

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

Correctness of Immutable ADTs

- 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
  - will need to use RI in most methods
  - will need to use AF in every method

ADTs: the Good and the Bad

- Provides data abstraction
  - can change data structures without breaking clients
- Comes at a cost
  - more work both to specify and to 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

Immutable Queues

Recall: Immutable Queue

- 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 {

  // @return len(obj)
  int size();

  // @return [x] ++ obj
  NumberQueue enqueue(int x);

  // @requires len(obj) > 0
  // @return (x, Q) with obj = Q ++ [x]
  DequeueParts dequeue();
}
```

Implementing a Queue with a List

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

  // AF: obj = this.items
  private List items;
```

- 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

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

  // AF: obj = this.items
  private List items;

  // @returns len(obj)
  public int size() {
    return len(this.items);
  };
```

- **Correctness of `size`:**

  len(this.items) =

Implementing a Queue with a List

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

  // AF: obj = this.items
  private List items;

  // @returns len(obj)
  public int size() {
    return len(this.items);
  };
```

- **Correctness of `size`:**

$$\text{len}(\text{this.items}) = \text{len}(\text{obj})$$ **by AF**

Implementing a Queue with a List

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

  // AF: obj = this.items
  private List items;

  // @effects obj = items
  ListQueue(List items) {
    this.items = items;
  }
```

- **Correctness of `constructor`:**

items =

Implementing a Queue with a List

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

  // AF: obj = this.items
  private List items;

  // @effects obj = items
  ListQueue(List items) {
    this.items = items;
  }
```

- **Correctness of `constructor`:**

items = this.items (from code)
= obj **AF**

Implementing a Queue with a List

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

  // AF: obj = this.items
  private List items;

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

- **Correctness of `enqueue`:**

return value =

Implementing a Queue with a List

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

  // AF: obj = this.items
  private List items;

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

- **Correctness of `enqueue`:**

return value	= x :: this.items	**spec of constructor**
	= x :: obj	**AF**
	= [] ⧺ (x :: obj)	**def of** concat
	= [x] ⧺ obj	**def of** concat

Summary of `ListQueue`

- Simplest possible implementation of ADT
  - abstract state = concrete state of one field

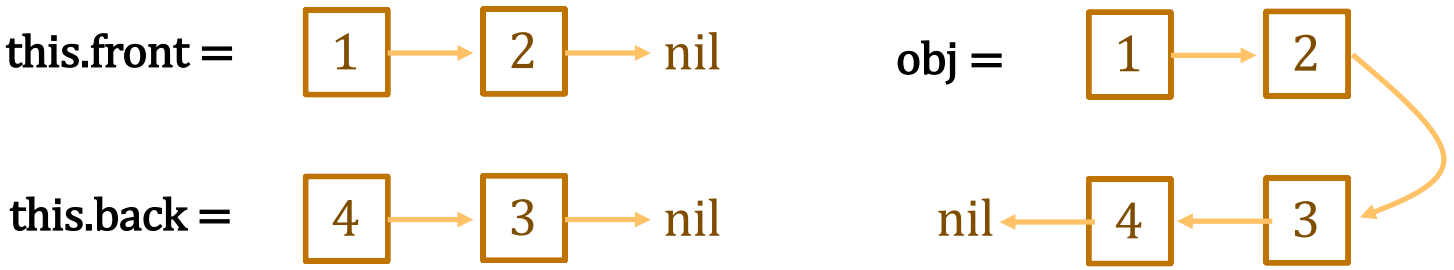
- Reasoning about every method is more complex
  - must apply AF to relate return value to spec's postcondition

    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)
  private List front;
  private List back;   // in reverse order
```

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



Implementing a Queue with Two Lists

```
// 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
  private List front;
  private List back;
```

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

    obj =

Implementing a Queue with Two Lists

```
// 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
  private List front;
  private List back;
```

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

$$\begin{aligned} \text{obj} &= \text{nil} +\!\!+ \text{rev}(\text{nil}) && \textbf{by AF} \\ &= \text{rev}(\text{nil}) && \textbf{def of } \text{concat} \\ &= \text{nil} && \textbf{def of } \text{rev} \end{aligned}$$

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

    (**311 alert**: this is the contrapositive)

Implementing a Queue with Two Lists

```
// 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
  private List front;
  private List back;

  // makes obj = front ++ rev(back)
  ListPairQueue(List front, List back) {
    …
  }
```

- Will implement this later...

Implementing a Queue with Two Lists

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

// @returns len(obj)
public int size() {
  return len(this.front) + len(this.back);
};
```

- **Correctness of `size`:**

$$\text{len(obj)} =$$

Implementing a Queue with Two Lists

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

// @returns len(obj)
public int size() {
  return len(this.front) + len(this.back);
};
```

- **Correctness of `size`:**

$$\begin{aligned} \text{len(obj)} &= \text{len(this.front} + \text{rev(this.back))} && \text{by AF} \\ &= \text{len(this.front)} + \text{len(rev(this.back))} && \text{by Example 3} \\ &= \text{len(this.front)} + \text{len(this.back)} && \text{by another induction} \end{aligned}$$

Implementing a Queue with Two Lists

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

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

- **Correctness of `enqueue`:**

ret value =

Implementing a Queue with Two Lists

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

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

- **Correctness of** `enqueue`**:**

ret value = (x :: this.front) ⧺ rev(this.back)	**spec of constructor**
= x :: (this.front ⧺ rev(this.back))	**def of** concat
= x :: obj	**AF**
= [] ⧺ (x :: obj)	**def of** concat
= [x] ⧺ obj	**def of** concat

Implementing a Queue with Two Lists

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

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

- as noted previously, precondition means this.back ≠ nil
- as we know, this means this.back = x :: L
  where x = this.back.hd and some L = this.back.tl

Implementing a Queue with Two Lists

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

- this.back = x :: L, where x = this.back.hd and L = this.back.tl

obj =

...

...

= (this.front ++ rev(this.back.tl)) ++ [this.back.hd]

Implementing a Queue with Two Lists

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

– this.back = x :: L, where x = this.back.hd and L = this.back.tl

obj	= this.front # rev(this.back)	**by AF**
	= this.front # rev(this.back.hd :: this.back.tl)	**since** back = x :: L
	= this.front # (rev(this.back.tl) # [this.back.hd])	**def of** rev
	= (this.front # rev(this.back.tl)) # [this.back.hd]	

Well-Known Facts About List Concatenation

- The List notes mention these facts

Identity	$L \mathbin{+\!\!+} nil = L$	($nil \mathbin{+\!\!+} L = L$ by definition)
Associativity	$(L \mathbin{+\!\!+} R) \mathbin{+\!\!+} S = L \mathbin{+\!\!+} (R \mathbin{+\!\!+} S)$	

- We will use those going forward

Implementing a Queue with Two Lists

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

// makes obj = front ++ rev(back)
ListPairQueue(List front, List back) {
  if (back == null) {
    this.front = null;
    this.back = rev(front);
  } else {
    this.front = front;
    this.back = back;
  }
}
```

holds since this.front = nil

holds since this.back ≠ nil

- Need to check that RI holds at end of constructor

Implementing a Queue with Two Lists

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

// makes obj = front ++ rev(back)
ListPairQueue(List front, List back) {
  if (back == null) {
    this.front = null;
    this.back = rev(front);
  } else {
    this.front = front;
    this.back = back;
  }
}
```

$obj = nil +\!\!+ rev(rev(front))$??

$obj = front +\!\!+ rev(back)$

- Need to check this creates correct abstract state

Implementing a Queue with Two Lists

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

ListPairQueue(List front, List back) {
  if (back == null) {
    this.front = null;
    this.back = rev(front);
  } else {
    …
  }
}
```

obj = nil ⧺ rev(rev(front))	
= nil ⧺ front	**AF**
= front	**because I said so**
= front ⧺ nil	**def of** concat
= front ⧺ rev(nil)	**def of** rev
= front ⧺ rev(back)	**since** back = nil

Correct ADT Implementation (Mutable Case)

Mutable ADTs

- **Previously:**
  - **state was immutable**
  - **set in the constructor and then never changed**

    only need to confirm RI holds at the end of the constructor

    if RI holds there, then it holds forever

- **Now:**
  - **allow state to be changed by methods**

Correctness of Mutable ADTs

- Check that the constructor…
  - creates a concrete state satisfying RI
  - creates the abstract state required by the spec
- Check the correctness of each observer method…
  - check value returned is the one stated by the spec
- Check the correctness of each mutator method…
  - check abstract state produced is one stated by the spec
  - check that the RI still holds
  - make sure there are no aliases

Recall: Mutable Version of Fast List ADT

```
// Represents a mutable list of numbers.
interface MutableFastList {

  // @return last(obj)
  int getLast();

  // @return obj
  List getList();

  // @modifies obj
  // @effects obj = x :: obj_0
  void cons(int x);
}
```

mutator method

Recall: One Concrete Rep for `FastList`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  private int last;
  private List list;

  // makes obj = list
  MutableFastListImpl(List list) {
    this.list = list;
    this.last = last(this.list);
  }
```

- We can use the same rep for a mutable version

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  private int last;
  private List list;

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    this.list = cons(x, this.list);
  };
```

- Let's check correctness...

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  private int last;
  private List list;

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list_0 }}
    {{ Post: obj = x :: obj_0 }}
};
```

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  private int last;
  private List list;

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list_0 }}
    {{ Post: obj = x :: obj_0 }}
};
```

What is missing?

Also, need the RI to hold!

obj	= this.list	by AF
	= x :: this.list$_0$	since this.list = x :: this.list$_0$
	= x :: obj$_0$	by AF

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  private int last;
  private List list;

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list_0 }}
    {{ Post: obj = x :: obj_0 and
             this.last = last(this.list) }}
  };
```

Also, need the RI to hold!

Does it? No!

- Postcondition is **`@returns`**, **`@effects`**, and **RI**

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  private int last;
  private List list;

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    this.list = cons(x, this.list);
    this.last = last(this.list);
    {{ this.list = x :: this.list_0 and this.last = last(this.list) }}
    {{ Post: obj = x :: obj_0 and this.last = last(this.list) }}
};
```

Rep Invariant now holds

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  private int last;
  private List list;

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    this.last = last(this.list);
    {{ this.last = last(this.list) }}
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list_0 and this.last = last(this.list_0) }}
    {{ Post: obj = x :: obj_0 and this.last = last(this.list) }}
};
```

Rep Invariant would not hold if we switched the order

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  private int last;
  private List list;

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    this.list = cons(x, this.list);
    this.last = last(this.list);
    {{ this.list = x :: this.list_0 and this.last = last(this.list) }}
    {{ Post: obj = x :: obj_0 and this.last = last(this.list) }}
};
```

This version is obviously correct, but O(n).

Can we do it faster?

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {
  // RI: this.last = last(this.list)
  // AF: obj = this.list
  private int last;
  private List list;

  // @modifies obj
  // @effects obj = x :: obj_0
  public void cons(int x) {
    if (this.list == null)
      this.last = x;
    this.list = cons(x, this.list);
    {{ ____________________________________________ }}
    {{ Post: obj = x :: obj_0 and this.last = last(this.list) }}
  };
```

O(1) version, but more complex reasoning (two branches)

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {

  public void cons(int x) {
    if (this.list == null)
      this.last = x;
    this.list = cons(x, this.list);
    {{ this.list = x :: this.list0 and this.list0 = nil and this.last = x }}
    {{ Post: obj = x :: obj0 and this.last = last(this.list) }}
  };
```

Case "then":

last(this.list) = last(x :: this.list$_0$) **since** this.list = x :: this.list$_0$

= last(x :: nil) **since** this.list$_0$ = nil

= x **def of** last

= this.last **since** x = this.last

last(x :: nil) := x

last(x :: y :: L) := last(y :: L)

Mutable List ADT with a Fast `getLast`

```
class MutableFastListImpl implements MutableFastList {

  public void cons(int x) {
    if (this.list == null)
      this.last = x;
    this.list = cons(x, this.list);
```

{{ $\text{this.list} = x :: \text{this.list}_0$ and $\text{this.list}_0 \neq \text{nil}$ and
$\text{this.last} = \text{this.last}_0$ and **$\text{this.last}_0 = \text{last}(\text{this.list}_0)$** }}

{{ **Post**: $\text{obj} = x :: \text{obj}_0$ and $\text{this.last} = \text{last}(\text{this.list})$ }}

from the RI
(will need this)

Case "else":

$\text{last}(\text{this.list}) = \text{last}(x :: \text{this.list}_0)$
$= \text{last}(\text{this.list}_0)$
$= \text{this.last}_0$
$= \text{this.last}$

since $\text{this.list} = x :: \text{this.list}_0$
def of last (**since** $\text{this.list}_0 \neq \text{nil}$)
since $\text{this.last}_0 = \text{last}(\text{this.list}_0)$
since $\text{this.last} = \text{this.last}_0$

$\text{last}(x :: \text{nil}) := x$
$\text{last}(x :: y :: L) := \text{last}(y :: L)$

Defensive Programming

Defensive Programming

- We try to catch all bugs via
  1. Type checking
  2. Reasoning
  3. Testing

- But some will get through…

- Add extra checks to catch them
  - reduces the work of debugging
  - we call this "defensive programming"

Defensive Programming

1. Add checks for invalid inputs
   - clients should not pass them

     they violate the **precondition**
   - given enough clients, some will pass invalid inputs

- EJ 49: check parameters for validity

- Check these if's not too expensive
  - would only skip if it would cause asymptotic slowdown
  - e.g., binary search cannot check the array is sorted

Defensive Programming

2. Check that the RI holds at the end of mutators
   - mutation could produce an invalid state
   - would cause painful debugging

     code doesn't crash then

     failure doesn't occur until the **next** method call (or later)
   - we call this method "`checkRep`" in 331

- Worth checking even if it is expensive
  - add a flag to enable them when testing

Defensive Programming

3. Check that the RI holds at the start of mutators
   - wait, why?
   - that's not even possible... is it?

- Can happen with rep exposure
  - mutation through an alias that breaks the RI
  - could be worse

    they could mutate it in a way that doesn't break the RI

    it's likely still a bug because the abstract state was wrongly changed

Using Mutable ADTs

Recall: Mutable Version of Fast List ADT

```
// Represents a mutable list of numbers.
interface MutableFastList {

  // @return last(obj)
  int getLast();

  // @returns first(obj), where
  //     first(nil)    := 0
  //     first(x :: L) := x
  int getFirst();

  // @return obj
  List getList();

  // @modifies obj
  // @effects obj = x :: obj_0
  void cons(int x);
}
```

Using the Mutable List ADT

```
// @requires L != nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//    where m = first(L)
void g(MutableFastList L, MutableFastList R, int k) {
  int i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    int m = L.getFirst();
    R.cons(m + i);
    i++;
  }
};
```

Using the Mutable List ADT

```
// @requires L != nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//    where m = first(L)
void g(MutableFastList L, MutableFastList R, int k) {
  int i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
    int m = L.getFirst();
    R.cons(m + i);
    i++;
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
  }
```

Using the Mutable List ADT

```
// @requires L != nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
void g(MutableFastList L, MutableFastList R, int k) {
  int i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
    int m = L.getFirst();
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 and m = first(L) }}
    R.cons(m + i);
    i++;
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
  }
```

Using the Mutable List ADT

```
// @requires L != nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//    where m = first(L)
void g(MutableFastList L, MutableFastList R, int k) {
  int i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    int m = L.getFirst();
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 and m = first(L) }}
    R.cons(m + i);
    {{ R = (m+i) :: R1 and R1 = (m+i-1) :: ... :: (m+1) :: R0 and m = first(L) }}
    i++;
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
  }
```

Using the Mutable List ADT

```
// @requires L != nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//    where m = first(L)
void g(MutableFastList L, MutableFastList R, int k) {
  int i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    int m = L.getFirst();
    R.cons(m + i);
    {{ R = (m+i) :: R1 and R1 = (m+i-1) :: ... :: (m+1) :: R0 and m = first(L) }}
    {{ R = (m+i) :: ... :: (m+1) :: R0 }}
    i++;
    {{ R = (m+i-1) :: ... :: (m+1) :: R0 }}
  }
```

Using the Mutable List ADT

```
// @requires L != nil
// @modifies R
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//    where m = first(L)
void g(MutableFastList L, MutableFastList R, int k) {
  int i = 1;
  // Inv: R = (m+i-1) :: … :: (m+1) :: R_0
  while (i <= k) {
    int m = L.getFirst();
    R.cons(m + i);
    {{ R = (m+i) :: R1 and R1 = (m+i-1) :: ... :: (m+1) :: R0 and m = first(L) }}
    {{ R = (m+i) :: ... :: (m+1) :: R0 }}
    i++;
  }
};
```

$R = (m+i) :: R_1$

$= (m+i) :: (m+i-1) :: \ldots :: (m+1) :: R_0$ since $R_1 = \ldots$

Using the Mutable List ADT

```
void g(MutableFastList L, MutableFastList R, int k)
```

- We have proven this code correct, but…

"Beware of bugs in the above code;
I have only proved it correct, not tried it."

Donald Knuth, 1977

- We should also try it…

Using the Mutable List ADT

```
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
void g(MutableFastList L, MutableFastList R, int k)
```

- **Try out the code:**

```
… // L = 2 :: 1
… // R = 2 :: 1
g(L, R, 3)
System.out.println(R);
```

- **What list should this print?**

5 :: 4 :: 3 :: 2 :: 1 :: nil

Using the Mutable List ADT

```
// @effects R = (m+k) :: … :: (m+1) :: R_0,
//     where m = first(L)
void g(MutableFastList L, MutableFastList R, int k)
```

- **Try out the code:**

```
… // L = 2 :: 1
… // R = 2 :: 1
g(L, R, 3)
System.out.println(R);
```

- **Instead, it prints** $8 :: 5 :: 3 :: 2 :: 1 :: nil$ **! How?!?**

L and R are aliases to the same MutableFastList

Reasoning with Aliases

- **Aliasing breaks reasoning!**
  - **there was nothing wrong with our math**
  - **our math did not correctly describe the program**

    modeling programs with aliasing is basically impossible

Another Mutable Queue ADT

- Another mutable version with different methods

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

  // @returns obj
  List<Integer> elements();

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

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

Implementing Mutable Queue with Two Arrays

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

  // AF: obj = rev(this.front) ++ this.back
  private ArrayList<Integer> front;
  private ArrayList<Integer> back;

  // @effects obj = vals
  ArrayPairQueue(ArrayList<Integer> vals) {
    this.front = new ArrayList<>();
    this.back = vals;
  }
```

We should check this...

Implementing Mutable Queue with Two Arrays

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

  // AF: obj = rev(this.front) ++ this.back
  private ArrayList<Integer> front;
  private ArrayList<Integer> back;

  // @effects obj = vals
  ArrayPairQueue(ArrayList<Integer> vals) {
    this.front = new ArrayList<>();
    this.back = vals;
    {{ this.front = [] and this.back = vals }}
    {{ Post: obj = vals }}
  }
```

Implementing Mutable Queue with Two Arrays

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

  // AF: obj = rev(this.front) ++ this.back
  private ArrayList<Integer> front;
  private ArrayList<Integer> back;

  // @effects obj = vals
  ArrayPairQueue(ArrayList<Integer> vals) {
    this.front = new ArrayList<>();
    this.back = vals;
    {{ this.front = [] and this.back = vals }}
    {{ Post: obj = vals }}
  }
```

Is this really correct?

No way to know for sure at the next method call!

obj = rev(this.front) # this.back — by AF

= rev([]) # this.back — since this.front = []

= [] # this.back — def of rev

= this.back = vals — since this.back = vals

Implementing Mutable Queue with Two Arrays

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

  // AF: obj = rev(this.front) ++ this.back
  private ArrayList<Integer> front;
  private ArrayList<Integer> back;

  // @effects obj = vals
  ArrayPairQueue(ArrayList<Integer> vals) {
    this.front = new ArrayList<>();
    this.back = new ArrayList<>(vals);
  }
```

- Must make a copy of the array!
  - then, we have the only reference to it (no aliases)

Implementing Mutable Queue with Two Arrays

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

  // AF: obj = rev(this.front) ++ this.back
  private ArrayList<Integer> front;
  private ArrayList<Integer> back;

  // @returns obj
  public List<Integer> elements() {
    ArrayList<Integer> result = new ArrayList<>();
    result.addAll(this.front);
    Collections.reverse(result);
    result.addAll(this.back);
    return result;
  };
```

This is O(n)...

We can optimize it if front = [].

$$\text{rev}([]) \# \text{this.back} = [] \# \text{this.back} = \text{this.back}$$

Implementing Mutable Queue with Two Arrays

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

  // AF: obj = rev(this.front) ++ this.back
  private ArrayList<Integer> front;
  private ArrayList<Integer> back;

  // @returns obj
  public List<Integer> elements() {
    if (this.front.size() == 0) {
      return this.back;   // O(1) when this.front = []
    } else {
      ArrayList<Integer> result = new ArrayList<>();
      result.addAll(this.front);
      Collections.reverse(result);
      result.addAll(this.back);
      return result;
    }
  };
```

Is this correct?

No way to say!

Implementing Mutable Queue with Two Arrays

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

  // AF: obj = rev(this.front) ++ this.back
  private ArrayList<Integer> front;
  private ArrayList<Integer> back;

  // @returns obj
  public List<Integer> elements() {
    ArrayList<Integer> result = new ArrayList<>();
    result.addAll(this.front);
    Collections.reverse(result);
    result.addAll(this.back);
    return result;
  };
```

- Cannot return an alias to `this.back`
  - must make a copy in all cases

Moral of the Story for Mutable Heap State

- More mutation gave us better efficiency
  - saved memory
  - immutable version could be just as fast
- More mutation means more complex reasoning
  - more facts to keep track of
  - more ways to make mistakes
  - more work to make sure we did it right
- New possibilities for exciting bugs!
  - must avoid aliasing of anything mutable

    this is "**representation exposure**"

Need for Mutable Heap State

- Saw that aliased mutable heap state is complex
  - avoid mixing aliasing and mutation
- Use coding conventions depending on context
  1. server-side data storage – mutation without aliasing
  2. client-side UI – aliasing without mutation
- In other cases, may need other conventions
  - two phase builder pattern