



CSE 331

Abstract Data Types (ADTs)

James Wilcox and Kevin Zatloukal

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

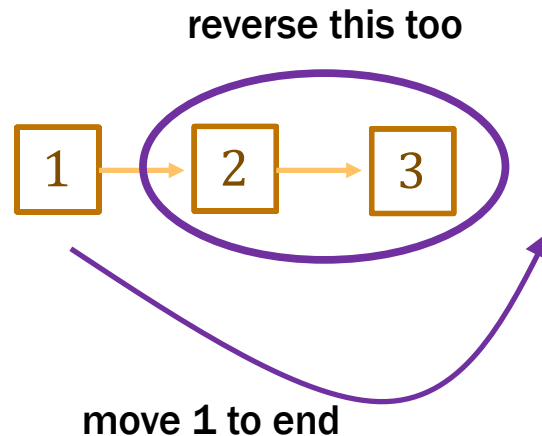
- Mathematical definition of $\text{rev}(S)$

$\text{rev}(\text{nil}) \quad := \text{nil}$

$\text{rev}(x :: L) \quad := \text{rev}(L) \# [x]$

included in the reference sheet
posted on the Topics page

- note that rev uses $\text{concat } (\#)$ as a helper function



Definition of List Reversal

- **Mathematical definition of $\text{rev}(S)$**

$$\begin{aligned}\text{rev}(\text{nil}) &:= \text{nil} \\ \text{rev}(x :: L) &:= \text{rev}(L) \# [x]\end{aligned}$$

- **note that rev uses $\text{concat } (\#)$ as a helper function**
- **ex: $\text{rev}(1 :: 2 :: 3 :: \text{nil}) = 3 :: 2 :: 1 :: \text{nil}$**

$$\begin{aligned}\text{rev}(1 :: 2 :: 3 :: \text{nil}) &= \text{rev}(2 :: 3 :: \text{nil}) \# [1] \\ &= \text{rev}(3 :: \text{nil}) \# [2] \# [1] \\ &= \text{rev}(\text{nil}) \# [3] \# [2] \# [1] \\ &= [] \# [3] \# [2] \# [1] \\ &= \dots = [3, 2, 1]\end{aligned}$$

what other tests should we do?

def of rev

def of rev

def of rev

def of rev

def of concat (many times)

Procedural Abstraction Example

- Specification of rev is imperative:

```
// @return same numbers but in reverse order, i.e.  
//   rev(nil)      := nil  
//   rev(x :: L) := rev(L) ++ [x]  
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 $\text{rev_acc}(L, \text{nil}) = \text{rev}(L)$ 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

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

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

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:**
$$\text{type FastLastList} \quad := \quad \{\text{last: } \mathbb{Z}, \text{list: List}\}$$
 - **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 List 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 use `FastList` after "**new**"
- **Standard style in Java**
 - e.g., create an `ArrayList` but store it as `List`

Example 1: Fast List

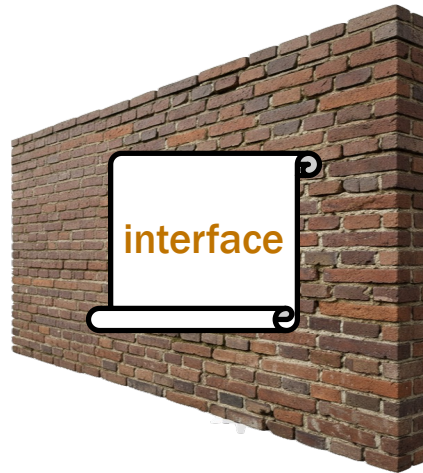
```
interface FastList {  
    int getLast();  
    List getList();  
}
```

Interfaces contain only
operations (methods)
not data structures

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



client



abstraction barrier



implementer

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();  
}
```

**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

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

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

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**: return properties of an object
 - **mutators**: change properties of the object
 - **producers**: create new objects from an existing one

Observers 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 return information about the object, but do not change it in any way
 - also called "getters"

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 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

	<u>Immutable</u>	<u>Mutable</u>
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 / Representation

actual fields of the record and the data stored in them

Last example: `int last, List list`

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)

ADT Terminology

New terminology for specifying ADTs

Concrete State / Representation

actual fields of the record and the data stored in them

Last example: `int last, List 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 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  $x :: \text{obj}$ 
   */
  FastList cons(int x);
}
```

- **Producer method:** makes a new list for you
 - “obj” above is a list, so $x :: \text{obj}$ 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 $(x, y) := \text{obj}$
 - 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 {
```

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

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

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

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

Specifying Polynomials

```
/**
 * Represents a polynomial with int coefficients. An
 * example int polynomial would be 3x + 5x^2 + 8.
 */
interface IntPoly {

    /** @return obj(x) */
observer    int eval(int x);

    /** @return obj + p */
producer   IntPoly add(IntPoly p);
```

- Simple specs *if* the reader knows polynomials already
 - not typical for classes we write to be well-known math ideas
 - how would we do this with math we covered?

Specifying Polynomials

```
/**
 * Represents a polynomial with int coefficients.
 * This is a list of pairs (c, n), where each "c" is
 * called a coefficient and "n" an exponent.
 *
 * A polynomial can be thought of as a function whose
 * value at a given x is calculated as follows:
 *
 *     value(nil, x)           := 0
 *     value((c, n) :: L, x) := c * x^n + value(L, x)
 *
 * For example, (3, 1) :: (5, 2) :: (8, 0) :: nil has
 * the value 3x + 5x^2 + 8 at a given x.
 */
interface IntPoly {
```

Specifying Polynomials

```
/**
 * Represents a polynomial with int coefficients.
 * This is a list of pairs (c, n), where each "c" is
 * called a coefficient and "n" an exponent.
 *
 * A polynomial can be thought of as a function whose
 * value at a given x is calculated as follows:
 *
 *     value(nil, x)           := 0
 *     value((c, n) :: L, x) := c * x^n + value(L, x)
 */
interface IntPoly {

    /** @return value(obj, x) */
    int eval(int x);
```


Specifying Polynomials

```
/**
 * Represents a polynomial with int coefficients.
 * This is a list of pairs (c, n), ...
 */
interface IntPoly {

    /** @return ?? */
    IntPoly add(IntPoly p);
}
```

- How do we specify `add`?

Specifying Polynomials

```
/**
 * Represents a polynomial with int coefficients.
 * This is a list of pairs (c, n), ...
 */
interface IntPoly {

    /** @return obj ++ p */
    IntPoly add(IntPoly p);
}
```

- Does this match the other definition ($\text{obj} + p$)?
 - would need to show that:

$$\text{value}(\text{obj} \# p, x) = \text{value}(\text{obj}, x) + \text{value}(p, x)$$

Specifying Polynomials

```
/**
 * Represents a polynomial with int coefficients.
 * This is a list of pairs (c, n), ...
 */
interface IntPoly {

    // Returns the coefficient with exponent "m"
    // @return ??
    int coeff(int m);
}
```

- How do we specify `coeff`?
 - what if there are multiple pairs with the same exponent?
 - what if there is no pair with that exponent?

Specifying Polynomials

```
// Returns the coefficient with exponent "m"
// @return coeff(obj, m), where
//   coeff(nil, m)           := 0
//   coeff((c, n) :: L, m) := c + coeff(L, m) if m = n
//   coeff((c, n) :: L, m) := coeff(L, m)    if m /= n
int coeff(int n);
```

Example: $3x + 5x^2 + 8$

```
coeff((3, 1) :: (5, 2) :: (8, 0) :: nil, 2)
= coeff((5, 2) :: (8, 0) :: nil, 2)
= 5 + coeff((8, 0) :: nil, 2)
= 5 + coeff(nil, 2)
= 5
```

```
def of coeff (since  $1 \neq 2$ )
def of coeff (since  $2 \neq 2$ )
def of coeff (since  $0 \neq 2$ )
def of coeff
```

Specifying Polynomials

```
/**
 * Represents a polynomial with int coefficients.
 * This is a list of pairs (c, n), ...
 */
interface IntPoly {

    // Returns the coefficient with exponent "m"
    // @return coeff(obj, m), where
    //   coeff(nil, m)           := 0
    //   coeff((c, n) :: L, m) := c + coeff(L, m) if m = n
    //   coeff((c, n) :: L, m) := coeff(L, m)    if m /= n
    int coeff(int n);
}
```

- Must add together all the coefficients with that exponent
- More than one option for missing exponents

Specifying Polynomials

```
/**
 * Represents a polynomial with int coefficients.
 * This is a list of pairs (c, n), ...
 */
interface IntPoly {

    // Returns the coefficient with exponent "m"
    // @return coeff(obj, m), where
    //   coeff(nil, m)           := 0
    //   coeff((c, n) :: L, m) := c + coeff(L, m) if m = n
    //   coeff((c, n) :: L, m) := coeff(L, m)    if m /= n
    int coeff(int n);
}
```

- What if we required all exponents to be distinct?
 - could stop at the first pair with that exponent...

Specifying Polynomials

```
/**
 * Represents a polynomial with int coefficients.
 * This is a list of pairs (c, n), ...
 * All pairs are required to have distinct exponents.
 */
interface IntPoly {

    // Returns the coefficient with exponent "m"
    // @return coeff(obj, m), where
    //   coeff(nil, m)           := 0
    //   coeff((c, n) :: L, m) := c           if m = n
    //   coeff((c, n) :: L, m) := coeff(L, m) if m /= n
    int coeff(int n);
}
```

- Can stop at the first matching exponent now

Specifying Polynomials

```
/**  
 * Represents a polynomial with int coefficients.  
 * This is a list of pairs (c, n), ...  
 * All pairs are required to have distinct exponents.  
 */  
interface IntPoly {
```

- Requiring distinct exponents is an "invariant"
 - extra assertion about a state that must always hold
 - we will see several types during this course
- An invariant of the abstract state is an "abstract invariant"

Recall: Data Structures

- ADTs hide the details of data structures
 - 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

List Indexing

$\text{at} : (\text{List}, \mathbb{N}) \rightarrow \mathbb{Z}$

$\text{at}(x :: L, 0) \quad := \quad x$

$\text{at}(x :: L, n+1) \quad := \quad \text{at}(L, n)$

- **Example:**

$\text{at}(1 :: 2 :: 3 :: \text{nil}, 2)$

$= \text{at}(2 :: 3 :: \text{nil}, 1)$

$= \text{at}(3 :: \text{nil}, 0)$

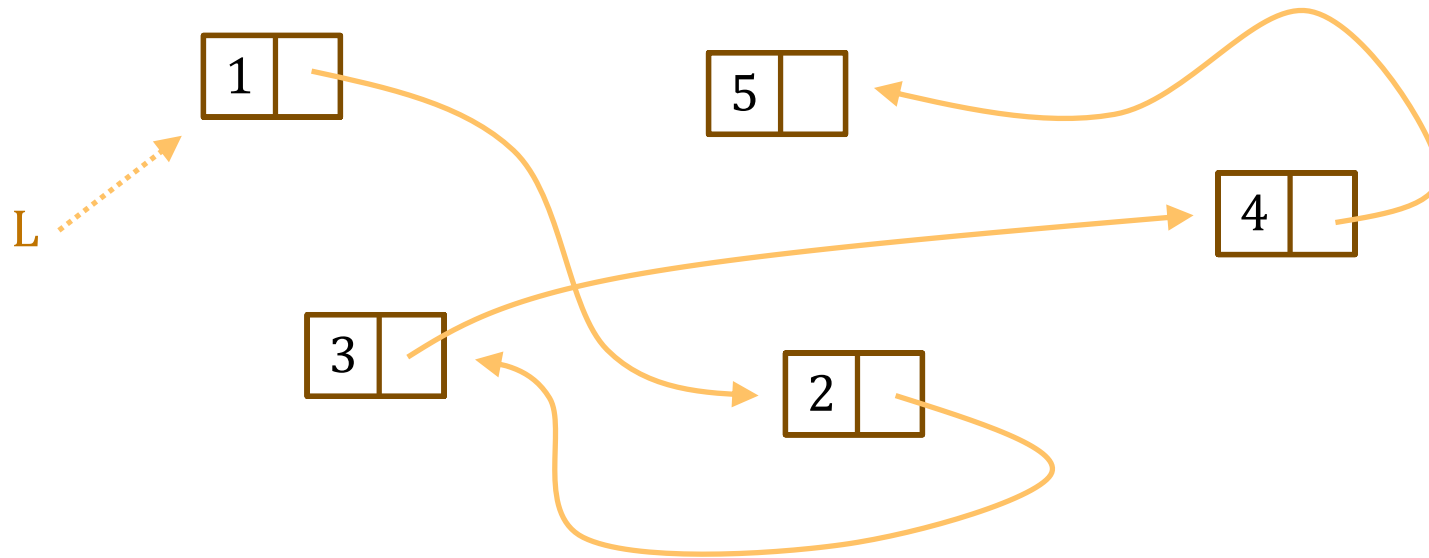
$= 3$

def of at

def of at

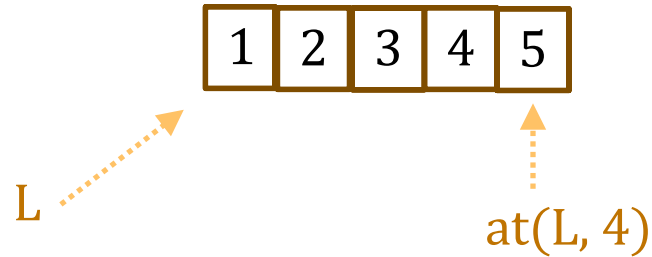
- **Not an especially efficient operation on linked lists...**

Linked Lists in Memory



- **Must follow the "next" pointers to find elements**
 - $\text{at}(L, n)$ is an $O(n)$ operation
 - no faster way to do this on a linked list...
 - how can we do it faster?

Faster Implementation of at

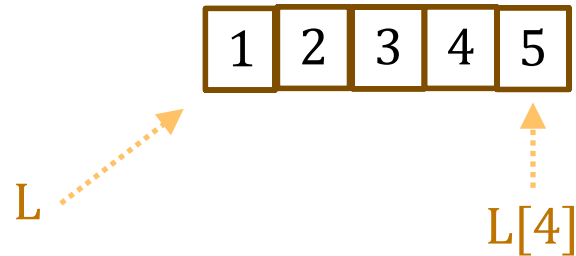


- **Alternative: store the elements next to each other**
 - can find the n-th entry by arithmetic:

$$\text{location of } L[4] = (\text{location of } L) + 4 * \text{sizeof}(\text{int})$$

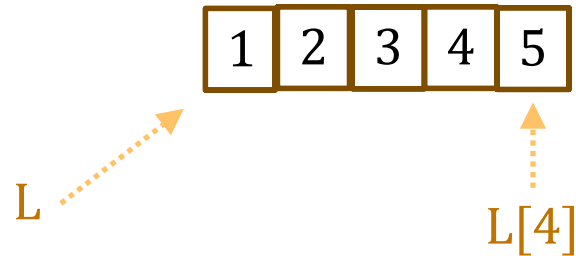
- **Resulting data structure is an array**

Faster Implementation of at



- Resulting data structure is an **array**
 - **Efficient** to read $L[i]$
 - **Inefficient** to...
 - insert elements anywhere but the end
- trees can do everything in $O(\log n)$ time

Arrays Are a Data Structure



- One concrete representation of the List ADT
 - linked lists are another
 - trees can also implement the List ADT
- Mathematically, these are all List
 - a *critical* inductive data type

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*
 - $\text{obj} = \dots$ 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 FastLastList ADT

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

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

Documenting the FastLastList 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 FastLastList 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 FastLastList 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 FastLastList ADT

```
/** Represents an (x, y) point in 2D space. */  
class PolarPoint implements FastList {  
  
    private final double r, theta;  
    ...  
}
```

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

Documenting the FastLastList ADT

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

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

Documenting the FastLastList ADT

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

```
/** Represents an (x, y) point in 2D space. */  
class PolarPoint implements FastList {  
    // RI:  $r = \sqrt{x^2 + y^2}$  and  $\theta = \text{atan2}(y, x)$   
    // 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 List Q;
    public final int x;
}
```

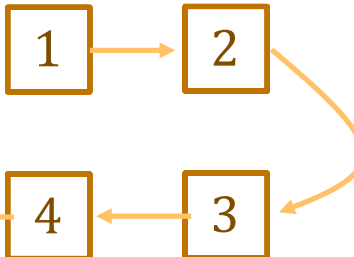
Implementing a Queue with Two Lists

```
// 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
```

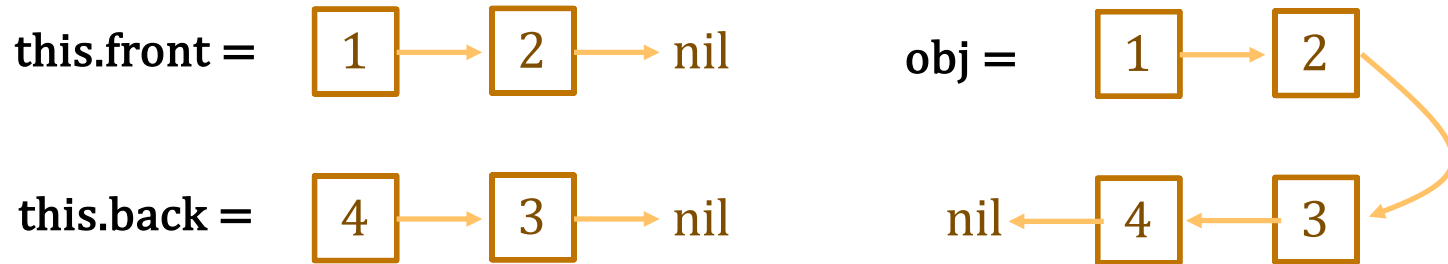
- Back part stored in reverse order
 - head of front is the first element
 - head of back is the *last* element

this.front = 

this.back = 

obj = 

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

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
(311 alert: this is the contrapositive)

Recall: Integer Polynomials

```
/**
 * Represents a polynomial with int coefficients.
 * This is a list of pairs (c, n), ...
 */
interface IntPoly {

    // @return obj ++ p
    IntPoly add(IntPoly p);

    // Returns the coefficient with exponent "m"
    // @return coeff(obj, m), where
    //   coeff(nil, m)           := 0
    //   coeff((c, n) :: L, m) := c + coeff(L, m) if m = n
    //   coeff((c, n) :: L, m) := coeff(L, m)    if m /= n
    int coeff(int n);
}
```

Documenting the IntPoly ADT

```
// Represents a list of pairs of the form (c, n),  
// where c is coefficient and n is exponent  
private static class MonoList {  
    public final int coeff;  
    public final int exponent;  
    public final MonoList tl;  
}
```

- **Each pair stored in an instance of** `MonoList`
 - empty list is represented by `null`

Documenting the IntPoly ADT

```
// Implements an IntPoly by storing the list.
class SimpleIntPoly implements IntPoly {

    // AF: obj = this.monos
    private final MonoList monos;

    private static class MonoList {
        public final int coeff;
        public final int exponent;
        public final MonoList tl;
    }
}
```

- Simple approach: concrete state = abstract state

Documenting the IntPoly ADT

```
// Implements an IntPoly by storing the list.
class SimpleIntPoly implements IntPoly {

    // RI: this.monos has no duplicate exponents
    // AF: obj = this.monos
    private final MonoList monos;

    // @return obj ++ p
    public IntPoly add(IntPoly p) { ... }

    // Returns the coefficient with exponent "m"
    // @return coeff(obj, m), where ...
    public int coeff(int n) { ... }
```

- What changes if we disallow duplicates?

Documenting the IntPoly ADT

```
// Implements an IntPoly by storing the list.
class SimpleIntPoly implements IntPoly {

    // RI: this.monos has no duplicate exponents
    // AF: obj = this.monos
    private final MonoList monos;

    // @return obj ++ p
write public IntPoly add(IntPoly p) { ... }

    // Returns the coefficient with exponent "m"
    // @return coeff(obj, m), where ...
read public int coeff(int n) { ... }
```

- Easier to observe, but harder to produce
 - i.e., easier to **read** but harder to **write**

Documenting the IntPoly ADT

```
// Implements an IntPoly by storing the list.
class SimpleIntPoly implements IntPoly {

    // RI: this.monos has no duplicate exponents
    // AF: obj = this.monos
    private final MonoList monos;

    // @return obj ++ p
    public IntPoly add(IntPoly p) { ... }

    // Returns the coefficient with exponent "m"
    // @return coeff(obj, m), where ...
    public int coeff(int n) { ... }
```

- More restrictions (RI) **usually** makes **reading** easier but **writing** harder, but not always...

Documenting the IntPoly ADT

```
// Implements an IntPoly by storing the list.
class SimpleIntPoly implements IntPoly {

    // RI: this.monos has no duplicate exponents,
    //      this.monos is sorted by exponent
    // AF: obj = this.monos
    private final MonoList monos;

    // @return obj ++ p
    public IntPoly add(IntPoly p) { ... }

    // Returns the coefficient with exponent "m"
    // @return coeff(obj, m), where ...
    public int coeff(int n) { ... }
```

- Sorting the monomials makes add **easier!**

Definition of List Reversal

- Can add two sorted lists of monomials in $O(n)$:

$$\begin{aligned}\text{poly-add}(\text{nil}, R) &:= R \\ \text{poly-add}(L, \text{nil}) &:= L \\ \text{poly-add}((c, m) :: L, (d, n) :: R) &:= (c, m) :: \text{poly-add}(L, (d, n) :: R) && \text{if } m < n \\ \text{poly-add}((c, m) :: L, (d, n) :: R) &:= (d, n) :: \text{poly-add}((c, m) :: L, :: R) && \text{if } m > n \\ \text{poly-add}((c, m) :: L, (d, n) :: R) &:= (c+d, m) :: \text{poly-add}(L, :: R) && \text{if } m = n\end{aligned}$$

- if $m < n$, then R has no matching exponent
- if $n < m$, then L has no matching exponent
- this is essentially merge-sort