CSE 331
Spring 2025

Abstraction

LATEST: 10.17
UPDATE
CHANGES IN VERSION 10.17:
THE CPU NO LONGER OVERHEATS WHEN YOU HOLD DOWN SPACEBAR.
COMMENTS:
LONGTIME USER4 WRITES:
THIS UPDATE BROKE MY WORKFLOW! MY CONTROL KEY IS HARD TO REACH, SO I HOLD SPACEBAR INSTEAD, AND I CONFIGURED EMACS TO INTERPRET A RAPID TEMPERATURE RISE AS "CONTROL".
ADMIN WRITES:
THAT'S HORRIFYING.
LONGTIME USER4 WRITES:
LOOK, MY SETUP WORKS FOR ME. JUST ADD AN OPTION TO REENABLE SPACEBAR HEATING.
EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

xkcd #1172

Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

Administrivia (05/16)

- HW7 is out!

The Third Leg of the Class

- HW1–3: write more realistic applications
  - saw how debugging gets harder

- HW4–6: write code correctly the first time
  - checked correctness without a computer

- HW7–9: write more complex applications
  - most applications have a core, tricky part
  - use the correctness toolkit to get that right
  - can work faster where debugging is easier

    only way to really know the UI is right is to try it

Procedural Abstraction

- Hide the details of the function from the caller
  - caller only needs to read the **specification**
  - (“procedure” means function)

- Caller promises to pass valid inputs
  - no promises on invalid inputs

- Implementer then promises to return correct outputs
  - does not matter how

Procedural Abstraction Example

- Specification of rev is imperative:

```
// @returns same numbers but in reverse order, i.e.
//    rev(nil) := nil
//    rev(cons(x, L)) := rev(L) ++ [x]
const rev = (L: List): List => {
  return rev_acc(L, nil);  // faster way
};
```

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

    we proved that rev_acc(L, nil) = rev(L) for all L by structural induction

Other Properties of High-Quality Code

- Professionals are expected to write high-quality code

- Correctness is the most important part of quality
  - users **hate** products that do not work properly

- Also includes the following
  - easy to change
  - easy to understand
  - modular

abstraction provides all three properties

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

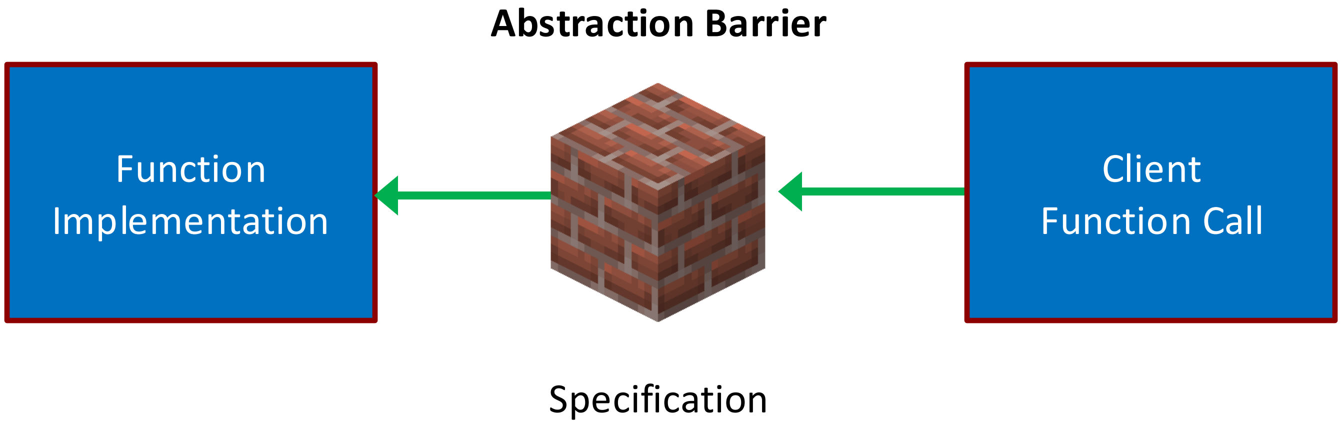
Benefits of Specifications

Clear specifications help with understandability and

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

Abstraction Barrier

- Specification is an...



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

Performance Improvements

- Before, we saw `rev-acc`, which is faster than `rev`
  - faster *algorithm* 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…

Recall: Last Element of a List

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

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

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

analogous idea:
store references to both
"front" and "back" nodes

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

empty list has undefined last

Defining Fast-Last List

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

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

Implementing Fast-Last List Helpers

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

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

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

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

Another Fast List

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

```
type FastBackList = List<bigint>;

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

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

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

Another Fast List Gone Wrong

```
type FastBackList = List<bigint>;

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

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

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

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

Yet Another Fast List?

```
type FastBackList =
    {list: List<bigint>, origList: List<bigint>};

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

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

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

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

Another Fast List — Take Three

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

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

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

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

Fast List as an Interface

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

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

- You’ve seen this in Java

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

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

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

Data Abstraction

Data Abstraction & ADTs

- Give clients only operations, not data
  - operations are “public”, data is “private”

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

    built into Java, JavaScript, etc.
  - data abstraction via procedural abstraction

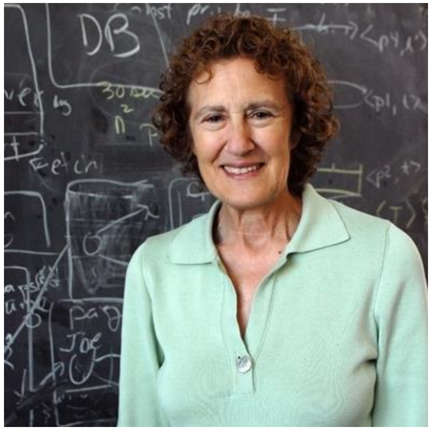


photo courtesy MIT

- Critical for the properties we want
  - easier to change data structure
  - easier to understand (hides details)
  - more modular

How to Make a `FastList` — Attempt One

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

- Values in `getLast` and `toList` fields are functions

- Note: getLast is *not* linear-time, but the constructor is!

- There is a cleaner way to do this
  - will also look more familiar

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

```
class FastLastList implements FastList {
  last: bigint | undefined;  // should be "readonly"
  list: List<bigint>;

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

  getLast = () => { return this.last; };
  toList = () => { return this.list; };
}
```

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

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

```
class FastLastList implements FastList {
  last: bigint | undefined;  // should be "readonly"
  list: List<bigint>;

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

  getLast = () => { return this.last; };
  toList = () => { return this.list; };
}
```

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

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

```
class FastLastList implements FastList {
  last: bigint | undefined;  // should be "readonly"
  list: List<bigint>;

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

  getLast = () => { return this.last; };
  toList = () => { return this.list; };
}
```

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

Another Way to Make a `FastList`

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

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

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

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

How Do Clients Get a `FastList`

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

- Export only `FastList` and `makeFastList`
  - completely hides the data representation from clients

- This is called a "factory function"
  - another design pattern
  - can change implementations easily in the future

    becomes `FastBackList` with a one-line change

- Difficult to add to the list with this interface
  - requires three calls: `toList`, `cons`, `makeFastList`

More Convenient Cons (via Interface)

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

const makeFastList = (): FastList => {
  return new FastBackList(nil);
};
```

- New method `cons` returns list with `x` in front
  - example of a “producer” method (others are “observers”)

    produces a new list for you
  - now, we only need to make an empty `FastList`

    anything else can be built via `cons`

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

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

const nilList: FastList = new FastBackList(nil);

const makeFastList = (): FastList => {
  return nilList;
};
```

- No need to create a new object using “`new`” *every time*
  - can reuse the same instance

    only possible since these are immutable!
  - example of the “singleton” **design pattern**

The 331 ADT Design Pattern

We will use the following design pattern for ADTs:

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

Stick to regular functions for rest of the code!

Specifications for ADTs

How to Specifications for ADTs?

- Run into problems when we try to write specs
  - for example, what goes after `@return`?

    don’t want to say returns the `.list` field (or reverse of that)

    we want to hide those details from clients

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

- Need some terminology to clear up confusion

New ADT Terminology: States

New terminology for specifying ADTs

Concrete State / Representation

actual fields of the record and the data stored in them

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

Abstract State / Representation

how clients should *think* about the object

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

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

List State: Concrete vs Abstract

Inductive types also differ in abstract / concrete states:

Concrete State / Representation

actual fields of the record and the data stored in them

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

Abstract State / Representation

how clients should *think* about the object

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

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

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

New terminology for specifying ADTs

Concrete State / Representation

actual fields of the record and the data stored in them

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

Abstract State / Representation

how clients should *think* about the object

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

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

Specifying FastList & getLast with “obj”

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

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

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

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

- Producer method: makes a new list for you
  - “obj” above is a list, so $\text{cons}(x, \text{obj})$ makes sense in math

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

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

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

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

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

- How do we specify this?

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

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

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

CSE 331
Spring 2025

Abstraction Functions & Representation Invariants

AN x64 PROCESSOR IS SCREAMING ALONG AT BILLIONS OF CYCLES PER SECOND TO RUN THE XNU KERNEL, WHICH IS FRANTICALLY WORKING THROUGH ALL THE POSIX-SPECIFIED ABSTRACTION TO CREATE THE DARWIN SYSTEM UNDERLYING OS X, WHICH IN TURN IS STRAINING ITSELF TO RUN FIREFOX AND ITS GECKO RENDERER, WHICH CREATES A FLASH OBJECT WHICH RENDERS DOZENS OF VIDEO FRAMES EVERY SECOND
BECAUSE I WANTED TO SEE A CAT JUMP INTO A BOX AND FALL OVER.
I AM A GOD.

xkcd #676

Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

Administrivia (05/19)

- HW7 LaTeX template is out!
  - also s/o to anonymous student’s Floyd Logic formatting template (~ macro)

Recall: ADTs & Data Abstraction

- 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

(Internally) Documenting an ADT Implementation

Recall: Abstract State

- 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 ADT Impls: Abstraction Function

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

Abstraction Function

defines what abstract state the field values currently represent

- Maps the field values to the object they represent
  - object is math, so this is a *mathematical* function

    there is no such function in the code — just a tool for reasoning
  - will usually write this as an *equation*

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

Example Abstraction Function: FastLastList

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

- Abstraction Function (AF) gives the abstract state
  - obj = abstract state
  - this = concrete state (record with fields .last and .list)
  - AF relates abstract state to the current concrete state

    okay that "last" is not involved here
  - specifications only talk about "obj", not "this"

    "this" will appear in our reasoning

Documenting ADT Impls: Representation Invariant

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

Abstraction Function

defines what abstract state the field values currently represent

only needs to be defined when RI is true

Representation Invariants (RI)

facts about the field values that should always be true

defines what field values are allowed

AF only needs to apply when RI is true

Example Representation Invariant: FastLastList

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

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

    last must be the last number in the list stored

Correctness of FastList Constructor: RI

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

  constructor(L: List<bigint>) {
    this.list = L;
    this.last = last(this.list);
  }
  …
```

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

Correctness of FastList Constructor: AF

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

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

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

Correctness of getLast (1/2)

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

- Use both RI and AF to check correctness

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

Correctness of getLast (2/2)

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

- Use both RI and AF to check correctness

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

= this.last **by RI**

Correctness of ADT implementation

- Check that the constructor…
  - creates a concrete state satisfying RI
  - creates the abstract state required by the spec

- Check the correctness of each method…
  - check value returned is the one stated by the spec
  - may need to use both RI and AF

ADTs: the Good and the Bad

- Provides data abstraction
  - can change data structures without breaking clients

- Comes at a cost
  - more work to specify and check correctness

- Not everything needs to be an ADT
  - don’t be like Java and make everything a class

- Prefer concrete types for most things
  - concrete types are easier to think about
  - introduce ADTs when the first *change* occurs

Worked Example: Immutable Queues

Immutable Queue Interface

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

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

observer

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

producer

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

producer

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

Implementing a Queue with a List ("Easiest")

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

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

- Easiest implementation is concrete = abstract state
  - just store the abstract state in a field

- Still requires extra work to check correctness...
  - abstraction barrier comes with a cost

Implementing a Queue with a List: Size

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

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

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

- Correctness of `size`:

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

nothing is "straight from the spec" anymore

Implementing a Queue with a List: Constructor

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

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

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

- **Correctness of** `constructor`**:**

$$\begin{aligned} \text{items} &= \text{this.items} && \textit{(from code)} \\ &= \text{obj} && \textbf{AF} \end{aligned}$$

Implementing a Queue with a List: Enqueue

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

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

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

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

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

Implementing a Queue with a List: Dequeue

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

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

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

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

Summary of `ListQueue`

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

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

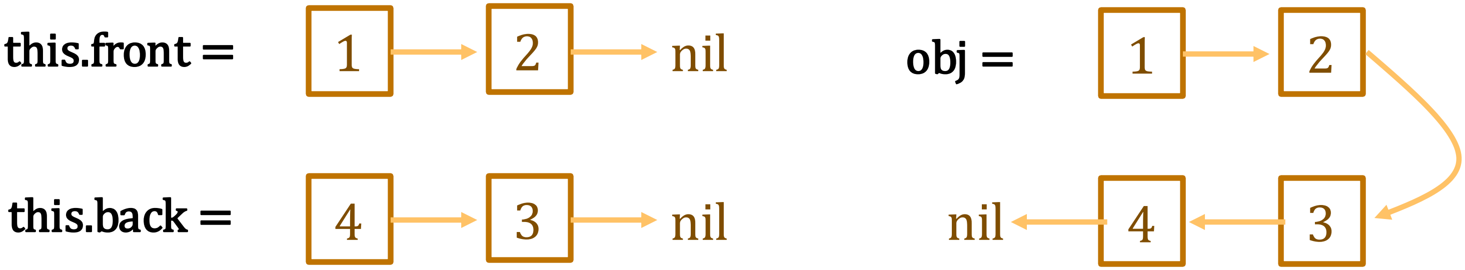
    code uses fields, but postcondition uses "obj"
  - this is the cost of the abstraction barrier

Implementing a Queue with Two Lists

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

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

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



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

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

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

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

$$\text{obj} \ =$$

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

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

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

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

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

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

Two-Queue List: Constructor (for now)

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

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

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

- **Will implement this later…**

Two-Queue List: Size (1/2)

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

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

- **Correctness of `size`:**

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

Two-Queue List: Size (2/2)

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

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

- **Correctness of `size`:**

$$
\begin{aligned}
\text{len}(\text{obj}) &= \text{len}(\text{this.front} \mathbin{+\!\!+} \text{rev}(\text{this.back})) && \textbf{by AF} \\
&= \text{len}(\text{this.front}) + \text{len}(\text{rev}(\text{this.back})) && \textbf{by earlier Ex.} \\
&= \text{len}(\text{this.front}) + \text{len}(\text{this.back}) && \textbf{by another induction}
\end{aligned}
$$

Two-Queue List: Enqueue (1/2)

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

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

- **Correctness of `enqueue`:**

ret value =

Two-Queue List: Enqueue (2/2)

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

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

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

$\text{ret value} = (x :: \text{this.front}) \mathbin{+\!\!+} \text{rev}(\text{this.back})$	**spec of constructor**
$= x :: (\text{this.front} \mathbin{+\!\!+} \text{rev}(\text{this.back}))$	**def of** concat
$= x :: \text{obj}$	**AF**
$= [\,] \mathbin{+\!\!+} (x :: \text{obj})$	**def of** concat
$= [x] \mathbin{+\!\!+} \text{obj}$	**def of** concat

Two-Queue List: Dequeue (1/2)

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

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

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

Two-Queue List: Dequeue (2/2)

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

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

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

Two-Queue List: Constructor (1/3)

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

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

RI: this.front = nil
or this.back ≠ nil

holds since this.front = nil

holds since this.back ≠ nil

- **Need to check that RI holds at end of constructor**

Two-Queue List: Constructor (2/3)

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

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

obj = nil ⧺ rev(rev(front)) ??

obj = front ⧺ rev(back)

- Need to check this creates correct abstract state

Two-Queue List: Constructor (3/3)

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

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

$$\begin{aligned}
\text{obj} &= \text{nil} \mathbin{+\!\!+} \text{rev}(\text{rev}(\text{front})) && \textbf{AF} \\
&= \text{nil} \mathbin{+\!\!+} \text{front} && \textbf{because } \mathbf{L = rev(rev(L))*} \\
&= \text{front} && \textbf{def of } \text{concat} \\
&= \text{front} \mathbin{+\!\!+} \text{nil} && \\
&= \text{front} \mathbin{+\!\!+} \text{rev}(\text{nil}) && \textbf{def of } \text{rev} \\
&= \text{front} \mathbin{+\!\!+} \text{rev}(\text{back}) && \textbf{since } \text{back} = \text{nil}
\end{aligned}$$

CSE 331
Spring 2025

More Inductive ADTs & Proofs

Matt Wang

& Ali, Alice, Andrew, Anmol, Antonio, Connor, Edison, Helena, Jonathan, Katherine, Lauren, Lawrence, Mayee, Omar, Riva, Saan, and Yusong

Weekly Wack (JS) Wednesday

```
typeof "str"
// returns 'string'


"str" instanceof String
// returns false


class Foo extends Function {
  constructor(val) {
    super()
    this.prototype.val = val
  }
}

new new Foo(":))")().val
// returns ':))'
```

Recall: Inductive Data Types

- Describe a set by ways of creating its elements
  - each is a “constructor”

    type $\mathrm{T} := \mathrm{A} \mid \mathrm{B} \mid \mathrm{C}(\mathrm{x} : \mathbb{Z}) \mid \mathrm{D}(\mathrm{x} : \mathbb{S}^*, \mathrm{t} : \mathrm{T}) \mid \mathrm{E}(\mathrm{s} : \mathrm{T}, \mathrm{t} : \mathrm{T})$

  - constructors taking arguments of type T are "recursive"

    $\mathrm{A}, \mathrm{B}, \mathrm{C}$ have no recursive arguments

    D has one recursive argument

    E has two recursive arguments

Categorizing Inductive Data Types

- Generalized "enum":
  - no constructors with recursive arguments

$$\textbf{type}\ \mathrm{T}\ :=\ \mathrm{A}\ |\ \mathrm{B}\ |\ \mathrm{C}(x : \mathbb{Z})$$

- Generalized "list":
  - constructor with 1 recursive arguments

$$\textbf{type}\ \mathrm{T}\ :=\ \mathrm{A}\ |\ \mathrm{B}\ |\ \mathrm{C}(x : \mathbb{Z})\ |\ \mathrm{D}(x : \mathbb{S}^*, \mathbf{t} : \mathbf{T})$$

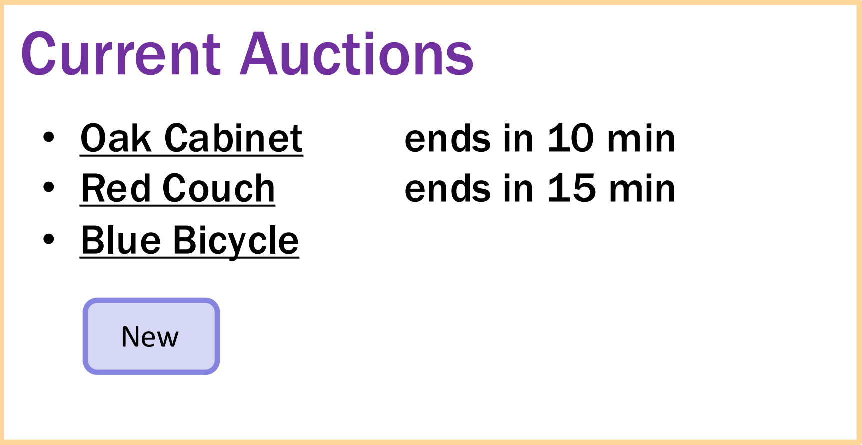
- Generalized "tree":
  - constructor with 2+ recursive arguments

$$\textbf{type}\ \mathrm{T}\ :=\ \mathrm{A}\ |\ \mathrm{B}\ |\ \mathrm{C}(x : \mathbb{Z})\ |\ \mathrm{D}(x : \mathbb{S}^*, t : \mathrm{T})\ |\ \mathrm{E}(\mathbf{s} : \mathbf{T}, \mathbf{t} : \mathbf{T})$$

Enums

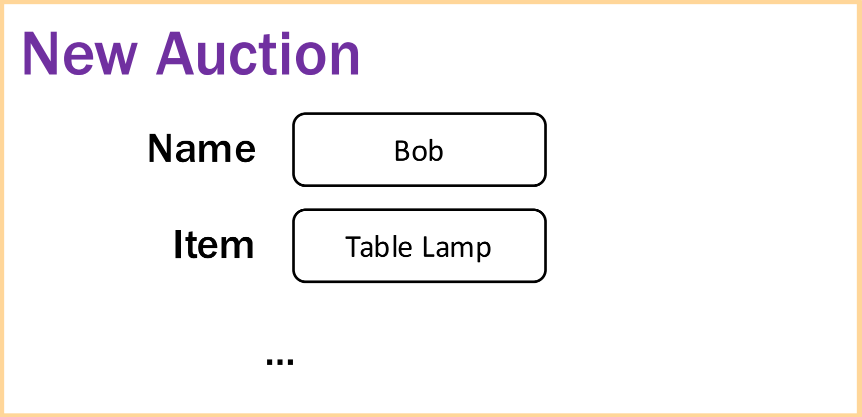
Enums Example: Auction pages

- Auction site has three different “pages”



Current Auctions

- Oak Cabinet ends in 10 min
- Red Couch ends in 15 min
- Blue Bicycle

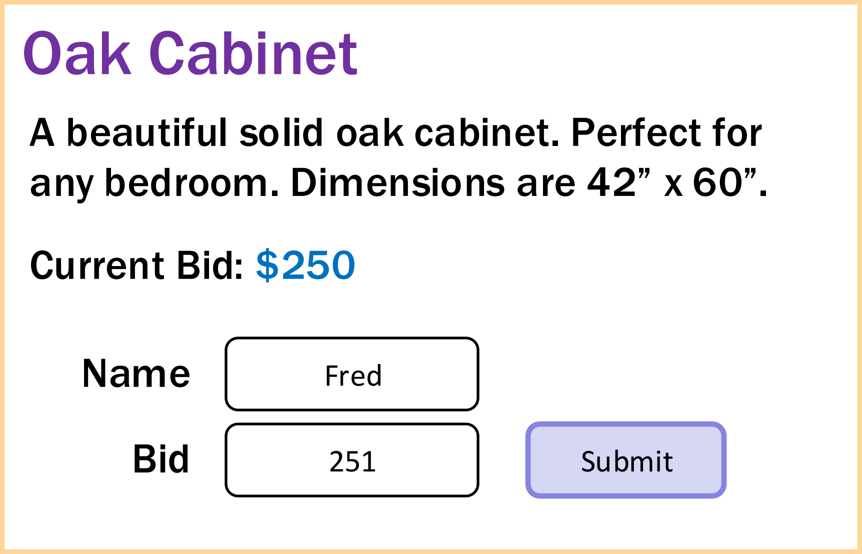


New Auction

Name: Bob

Item: Table Lamp

…



Oak Cabinet

A beautiful solid oak cabinet. Perfect for any bedroom. Dimensions are 42” x 60”.

Current Bid: $250

Name: Fred

Bid: 251

App component needs to show one of these components.

Must keep track of which one we are currently showing.

Auction `App.tsx` – Pages as Enums

```
type Page = {kind: "list"}
          | {kind: "new"}
          | {kind: "details", name: string};

type AppState = {page: Page};
```

- **Page is an inductive data type:**

  type Page := list | new | details(name: $\mathbb{S}$*)

  – **App keeps track of the current page**
  – **note that "**details**" has an argument (which auction's details)**

Auction `App.tsx` – Rendering Enum Pages

```
type Page = {kind: "list"}
          | {kind: "new"}
          | {kind: "details", name: string};

type AppState = {page: Page};

class App extends Component<{}, AppState> {
  render = (): JSX.Element => {
    if (this.state.page.kind === "list") {
      return <AuctionList/>;
    } else if (this.state.page.kind === "new") {
      return <NewAuction/>;
    } else {
      return <AuctionDetails
                  name={this.state.page.name}/>;
    }
  };
```

Lists

Generalized Lists

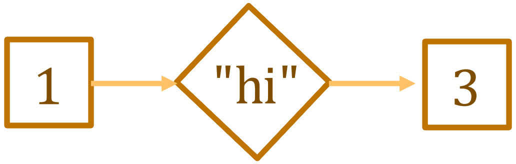
- **Lists can have multiple recursive constructors**

type ShapeList := nil | square(x: $\mathbb{Z}$, L: ShapeList) | diamond(y: $\mathbb{S}^*$, L: ShapeList)

  - two different ways to add to the front

- **Still not much more complicated**

square(1, diamond("hi", square(3, nil))) =



Trees

Trees in the Wild

- Trees are the most general case…

- Some prominent examples of trees:
  - HTML: used to describe UI
  - JSON: used to describe just about any data

Proofs for Trees...

type $\mathrm{T} := \mathrm{A}$

$\quad\mid \mathrm{B}$

$\quad\mid \mathrm{C}(\mathrm{x} : \mathbb{Z})$

$\quad\mid \mathrm{D}(\mathrm{x} : \mathbb{S}^*, \mathrm{t} : \mathrm{T})$

$\quad\mid \mathrm{E}(\mathrm{s} : \mathrm{T}, \mathrm{t} : \mathrm{T})$

- **To prove $\mathrm{P}(\mathrm{t})$ for all $\mathrm{t} : \mathrm{T}$:**

  prove $\mathrm{P}(\mathrm{A})$

  prove $\mathrm{P}(\mathrm{B})$

  prove $\mathrm{P}(\mathrm{C}(\mathrm{x}))$

  prove $\mathrm{P}(\mathrm{D}(\mathrm{x}, \mathrm{t}))$

  prove $\mathrm{P}(\mathrm{E}(\mathrm{s}, \mathrm{t}))$

  – (this is proof by cases)

Proofs for Trees… Use Structural Induction!

$$
\begin{aligned}
\textbf{type}\ \mathrm{T}\ :=\ & \mathrm{A} \\
|\ & \mathrm{B} \\
|\ & \mathrm{C}(x : \mathbb{Z}) \\
|\ & \mathrm{D}(x : \mathbb{S}^*, t : \mathrm{T}) \\
|\ & \mathrm{E}(s : \mathrm{T}, t : \mathrm{T})
\end{aligned}
$$

- **To prove $P(t)$ for all $t : T$:**

  prove $P(A)$

  prove $P(B)$

  prove $P(C(x))$

  prove $P(D(x, t))$ **assuming** $P(t)$

  prove $P(E(s, t))$ **assuming** $P(s)$ **and** $P(t)$

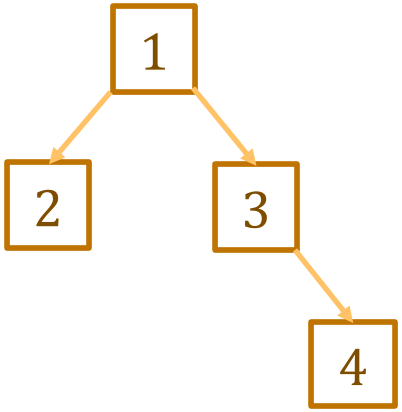
  - **this is structural induction!**

Inductive Binary Trees

type Tree := empty | node(x : ℤ, L : Tree, R : Tree)

- **Inductive definition of binary trees of integers**

node(1, node(2, empty, empty), node(3, empty, node(4, empty, empty))))



Functions on Binary Trees: num-nodes

type Tree := empty | node(x: $\mathbb{Z}$, L: Tree, R: Tree)

num-nodes : Tree → $\mathbb{N}$

num-nodes(empty) := 0

num-nodes(node(x, L, R)) := 1 + num-nodes(L) + num-nodes(R)

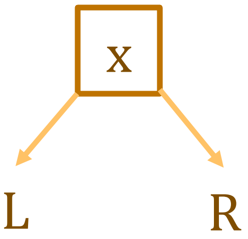
- **How many nodes are in the tree?**

Functions on Binary Trees: num-edges

```
type Tree :=  empty |  node(x: ℤ, L: Tree, R: Tree)

num-edges : Tree → ℕ
num-edges(empty)              := -1
num-edges(node(x, L, R))      := 2 + num-edges(L) + num-edges(R)
```

- How many edges are in the tree?
  - "edge" is a move from one node to another

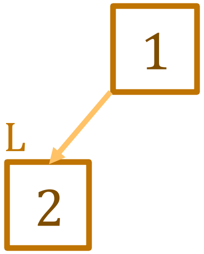


Tracing Through num-edges

num-edges : Tree → ℕ
num-edges(empty) := -1
num-edges(node(x, L, R)) := 2 + num-edges(L) + num-edges(R)

- **Why a "-1" here?**

num-edges(node(x, L, empty))
= 2 + num-edges(L) + num-edges(empty)
= 2 + num-edges(L) + -1
= 1 + num-edges(L)



num-edges(node(x, empty, empty))
= 2 + num-edges(empty) + num-edges(empty)
= 2 + -1 + -1
= 0

Proving Claims on Trees Example (Base Case)

Let $\mathrm{P(T)}$ **be the claim "**$\text{num-nodes}(\mathrm{T}) = \text{num-edges}(\mathrm{T}) + 1$**"**

Prove $\mathrm{P(T)}$ **holds for <u>any</u> tree** T **by structural induction**

Base Case: prove $\mathrm{P(empty)}$

num-nodes(empty)	
$= 0$	**def of** num-nodes
$= \text{-}1 + 1$	
$= \text{num-edges(empty)} + 1$	**def of** num-edges

$\text{num-nodes(empty)} := 0$

$\text{num-edges(empty)} := \text{-}1$

Proving Claims on Trees Example (Induction Setup)

Let $\mathrm{P(T)}$ **be the claim** "$\text{num-nodes}(\mathrm{T}) = \text{num-edges}(\mathrm{T}) + 1$"

Inductive Hypothesis: assume P(L) and P(R)

- assume P for both subtrees

Inductive Step: prove $\mathrm{P(node(x, L, R))}$

- use known facts and definitions and Inductive Hypotheses

Proving Claims on Trees Example (Inductive Step)

Let P(T) **be the claim** "$\text{num-nodes(T)} = \text{num-edges(T)} + 1$"

Inductive Step: prove P(node(x, L, R))

$\text{num-nodes(node(x, L, R))}$	
$= 1 + \text{num-nodes(L)} + \text{num-nodes(R)}$	**def of** num-nodes
$= 1 + \text{num-edges(L)} + 1 + \text{num-nodes(R)}$	**Ind. Hyp.**
$= 1 + \text{num-edges(L)} + 1 + \text{num-edges(R)} + 1$	**Ind. Hyp.**
$= 2 + \text{num-edges(L)} + \text{num-edges(R)} + 1$	
$= \text{num-edges(node(x, L, R))} + 1$	**def of** num-edges

$\text{num-nodes(node(x, L, R))} := 1 + \text{num-nodes(L)} + \text{num-nodes(R)}$

$\text{num-edges(node(x, L, R)} \;\; := 2 + \text{num-edges(L)} + \text{num-edges(R)}$

Common ADTs as Lists

- 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

- All of these are specified as lists
  - maps are "association lists" (lists of pairs)

Association Lists (1/3)

- A list of pairs $\text{List}\langle(\text{K},\text{V})\rangle$ is an "association list"
  - can be used to describe a map from keys to values
  - set the value associated with a key:

$$\text{set-value} : (\text{K}, \text{V}, \text{List}\langle(\text{K}, \text{V})\rangle) \rightarrow \text{List}\langle(\text{K}, \text{V})\rangle$$

$$\text{set-value}(\text{x}, \text{v}, \text{L}) \ := \ (\text{x}, \text{v}) :: \text{L}$$

  - first pair with that key has the current value
  - could choose to remove any later pairs with this key

saves memory and makes debugging harder (hooray!)

Association Lists (2/3)

- A list of pairs $\text{List}\langle(\text{K},\text{V})\rangle$ is an "association list"
  - can be used to describe a map from keys to values
  - retrieve the (first) value associated with a key:

$$
\begin{aligned}
&\text{get-value} : (\text{K}, \text{List}\langle(\text{K}, \text{V})\rangle) \to \text{V} && \\
&\text{get-value}(\text{x}, \text{nil}) && := \text{undefined} && \\
&\text{get-value}(\text{x}, (\text{y}, \text{v}) :: \text{L}) && := \text{v} && \textbf{if } \text{x} = \text{y} \\
&\text{get-value}(\text{x}, (\text{y}, \text{v}) :: \text{L}) && := \text{get-value}(\text{x}, \text{L}) && \textbf{if } \text{x} \neq \text{y}
\end{aligned}
$$

$$
\begin{aligned}
&\text{contains-key} : (\text{K}, \text{List}\langle(\text{K}, \text{V})\rangle) \to \mathbb{B} && \\
&\text{contains-key}(\text{x}, \text{nil}) && := \text{false} && \\
&\text{contains-key}(\text{x}, (\text{y}, \text{v}) :: \text{L}) && := \text{true} && \textbf{if } \text{x} = \text{y} \\
&\text{contains-key}(\text{x}, (\text{y}, \text{v}) :: \text{L}) && := \text{contains-key}(\text{x}, \text{L}) && \textbf{if } \text{x} \neq \text{y}
\end{aligned}
$$

Notice anything about these functions?

Association Lists (3/3)

Two association lists are "the same" if they return the same values for each key

- Can see that get/set work as expected:
  - get the value just set (v):

$\text{get-value}(\text{x}, \text{set-value}(\text{x}, \text{v}, \text{L}))$

$\quad = \text{get-value}(\text{x}, (\text{x}, \text{v}) :: \text{L})$ **def of** set-value

$\quad = \text{v}$ **def of** get-value

  - get the value of a key not just set ($\text{x} \neq \text{y}$):

$\text{get-value}(\text{y}, \text{set-value}(\text{x}, \text{v}, \text{L}))$

$\quad = \text{get-value}(\text{y}, (\text{x}, \text{v}) :: \text{L})$ **def of** set-value

$\quad = \text{get-value}(\text{y}, \text{L})$ **def of** get-value **(since** $\text{x} \neq \text{y}$**)**

$\text{get-value}(\text{x}, (\text{y}, \text{v}) :: \text{L}) \;:=\; \text{v}$ **if** $\text{x} = \text{y}$

$\text{set-value}(\text{x}, \text{v}, \text{L}) \;:=\; (\text{x}, \text{v}) :: \text{L}$

$\text{get-value}(\text{x}, (\text{y}, \text{v}) :: \text{L}) \;:=\; \text{get-value}(\text{x}, \text{L})$ **if** $\text{x} \neq \text{y}$

Immutable Map

- An "association list" also called a "map"

```
            // List of (key, value) pairs
            interface Map<K, V> {

              // @returns contains-key(x, obj)
observer      containsKey(x: K): boolean;

              // @requires contains-key(x, obj)
              // @returns get-value(x, obj)
observer      getValue(x: K): V;

              // @returns set-value(x, v, obj)
producer      setValue(x: K, v: V): Map<K, V>;
            }
```

Mutable Map Teaser (more next week)

- An "association list" also called a "map"

```
// List of (key, value) pairs
interface Map<K, V> {
```

observer

```
  // @returns contains-key(x, obj)
  containsKey(x: K): boolean;
```

observer

```
  // @requires contains-key(x, obj)
  // @returns get-value(x, obj)
  getValue(x: K): V;
```

mutator

```
  // @modifies obj
  // @effects obj = set-value(x, v, obj)
  setValue(x: K, v: V): void;
}
```

This version saves some memory and …
makes debugging harder and…

Introduces possible aliasing bugs!

Common ADTs as Trees

- 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
- All of these are specified as lists
  - maps are "association lists" (lists of pairs)
- Set and Map can be implemented with trees

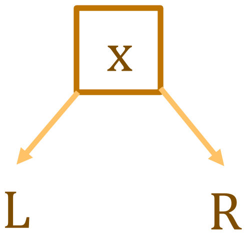
Defining Binary Search Trees (BSTs)

$$\textbf{type}\ \text{BST} := \text{empty} \mid \text{node}(x : \mathbb{Z}, v : \mathbb{Z}, L : \text{BST}, R : \text{BST})$$

- stores a value "v" as well as a key "x"

- BSTs add an extra rep invariant to every node

$$\text{contains-key}(y, L) \rightarrow (y < x)$$
$$\text{contains-key}(z, R) \rightarrow (x < z)$$



Binary Search Trees: contains-key

$$\textbf{type}\ \mathrm{BST} := \ \mathrm{empty} \mid \ \mathrm{node}(x : \mathbb{Z}, v : \mathbb{Z}, L : \mathrm{BST}, R : \mathrm{BST})$$

- **See if a key is in the tree:**

$$\text{contains-key} : (\mathbb{Z}, \mathrm{BST}) \to \mathbb{B}$$

$$\begin{array}{lll}
\text{contains-key}(x, \mathrm{empty}) & := \mathrm{false} & \\
\text{contains-key}(x, \mathrm{node}(y, w, L, R)) & := \mathrm{true} & \textbf{if } x = y \\
\text{contains-key}(x, \mathrm{node}(y, w, L, R)) & := \text{contains-key}(x, L) & \textbf{if } x < y \\
\text{contains-key}(x, \mathrm{node}(y, w, L, R)) & := \text{contains-key}(x, R) & \textbf{if } y < x
\end{array}$$

Binary Search Trees: get-value

type BST := empty | node(x : ℤ, v : ℤ, L : BST, R : BST)

- Get the value associated with a key in the tree:

```
get-value : (ℤ, BST) → ℤ

get-value(x, empty)             := undefined
get-value(x, node(y, w, L, R))  := w                 if x = y
get-value(x, node(y, w, L, R))  := get-value(x, L)   if x < y
get-value(x, node(y, w, L, R))  := get-value(x, R)   if y < x
```

Binary Search Trees: set-value (1/2)*

$\textbf{type}\ \mathrm{BST} := \mathrm{empty} \mid \mathrm{node}(x : \mathbb{Z}, v : \mathbb{Z}, L : \mathrm{BST}, R : \mathrm{BST})$

- **Set a (key, value) in the tree:**

$$\text{set-value} : (\mathbb{Z}, \mathbb{Z}, \mathrm{BST}) \to \mathrm{BST}$$

$$\begin{array}{lll} \text{set-value}(x, v, \text{empty}) & := \text{node}(x, v, \text{empty}, \text{empty}) & \\ \text{set-value}(x, v, \text{node}(y, w, L, R)) & := \text{node}(x, v, L, R) & \textbf{if } x = y \\ \text{set-value}(x, v, \text{node}(y, w, L, R)) & := \text{node}(y, w, \text{set-value}(x, v, L), R) & \textbf{if } x < y \\ \text{set-value}(x, v, \text{node}(y, w, L, R)) & := \text{node}(y, w, L, \text{set-value}(x, v, R)) & \textbf{if } y < x \end{array}$$

  - **add a new node if the key is not present**
  - **replace the value if the key is present***

Binary Search Trees: set-value (2/2)*

type BST := empty | node(x : ℤ, v : ℤ, L : BST, R : BST)

- **Set a (key, value) in the tree:**

  set-value : (ℤ, ℤ, BST) → BST

  set-value(x, v, empty) := node(x, v, empty, empty)
  set-value(x, v, node(y, w, L, R)) := node(x, v, L, R) **if** $x = y$
  set-value(x, v, node(y, w, L, R)) := node(y, w, set-value(x, v, L), R) **if** $x < y$
  set-value(x, v, node(y, w, L, R)) := node(y, w, L, set-value(x, v, R)) **if** $y < x$

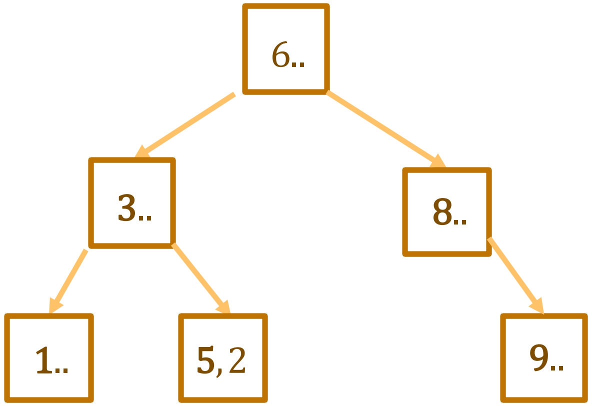
  - **note that this does <u>not</u> mutate the existing tree**
  - **the old tree is still around and unchanged**

Think, Pair, Share: Tree Tea of Washington

type BST := empty | node(x : ℤ, v : ℤ, L : BST, R : BST)

sli.do #cse331

```
Consider  L = node(6, ...,
                node(3, ...,
                      node(1, ..., empty, empty),
                      node(5, 2, empty, empty)),
                node(8, ...,
                      empty,
                node(9, ..., empty, empty)))
```



After calling set-value(5, 7, L), **which nodes need to be recreated?**

1. **just node 5**
2. **nodes 6, 3, 5**
3. **nodes 6, 3, 1, 5**
4. **all nodes**

set-value(x, v, empty)	:=	node(x, v, empty, empty)	
set-value(x, v, node(y, w, L, R))	:=	node(x, v, L, R)	if x = y
set-value(x, v, node(y, w, L, R))	:=	node(y, w, set-value(x, v, L), R)	if x < y
set-value(x, v, node(y, w, L, R))	:=	node(y, w, L, set-value(x, v, R))	if y < x

Visualizing BST set-value

set-value(5, 7, node(6, a, L_1, R_1))

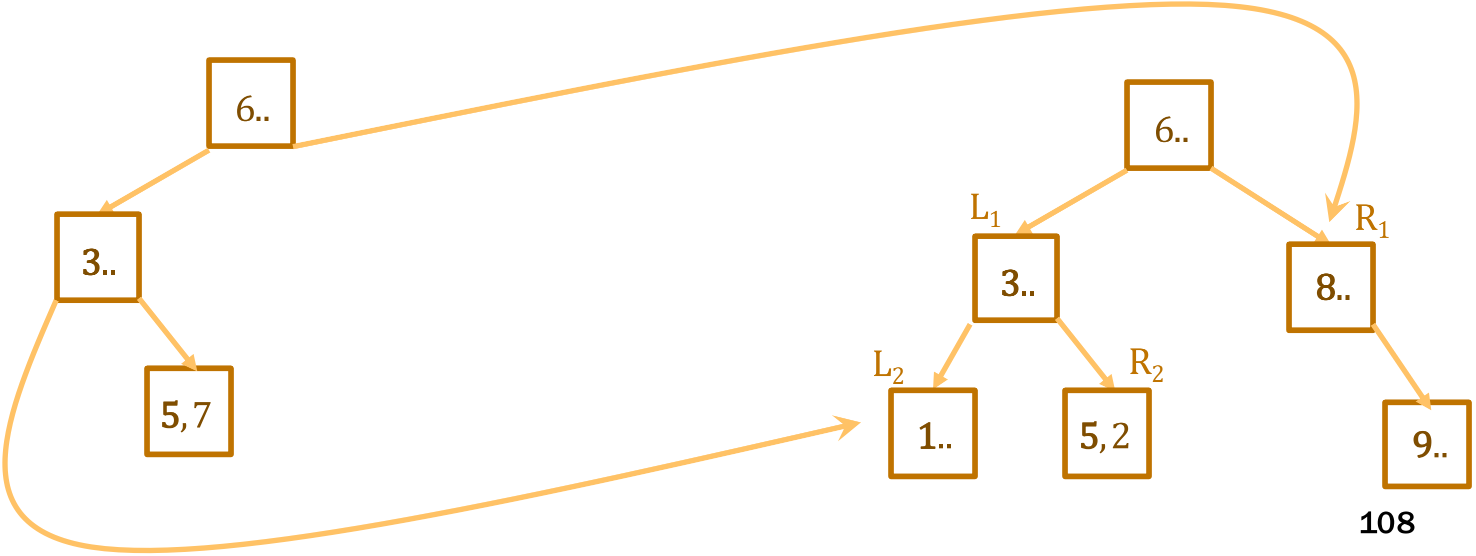
= node(6, a, set-value(5, 7, node(3, b, L_2, R_2)), R_1) **def of** set-value ($5 < 6$)

= node(6, a, node(3, b, L_2, set-value(5, 7, node(5, 2, empty, empty))), R_1) … ($5 > 3$)

= node(6, a, node(3, b, L_2, node(5, 7, empty, empty))), R_1) **def of** set-value

- **only copies the <u>path to</u> 5 in the tree**

  only O(log n) extra memory for a balanced tree



Reasoning about BSTs

- Use reasoning to make sure this works…
  - easier to reason than to *debug and then reason*
  - get the value just set (v):

$$\text{get-value}(x, \text{set-value}(x, v, T)) \;=\; v \;\;??$$

  - get the value of a key not just set ($x \neq y$):

$$\text{get-value}(y, \text{set-value}(x, v, T)) \;=\; \text{get-value}(y, T) \;\;??$$

  - how do we prove this for all $T : \text{BST}$?

    last time, it was just a calculation

Structural Induction on BSTs: Base Case

Let P(T) **be the claim "**get-value(x, set-value(x, v, T)) = v**"**

Prove P(T) **holds for <u>any</u> BST** T **by structural induction**

Base Case: prove P(empty)

get-value(x, set-value(x, v, empty))	
= get-value(x, node(x, v, empty, empty))	**def of** set-value
= v	**def of** get-value

set-value(x, v, empty) := node(x, v, empty, empty)		
set-value(x, v, node(y, w, L, R)) := node(x, v, L, R)	if x = y	get-value(x, node(y, w, L, R)) := v
set-value(x, v, node(y, w, L, R)) := node(y, w, set-value(x, v, L), R)	if x < y	get-value(x, node(y, w, L, R)) := get-value(x, L)
set-value(x, v, node(y, w, L, R)) := node(y, w, L, set-value(x, v, R))	if y < x	get-value(x, node(y, w, L, R)) := get-value(x, R)

Structural Induction on BSTs: Induction Setup

$\text{P(T) := "get-value(x, set-value(x, v, T)) = v"}$

Inductive Hypothesis: assume P(L) and P(R)

– assume P for both subtrees

Inductive Step: prove $\text{P(node(y, w, L, R))}$

– use known facts and definitions and Inductive Hypotheses

Structural Induction on BSTs: Inductive Step (1/4)

P(T) := "get-value(x, set-value(x, v, T)) = v"

Inductive Step: prove P(node(y, w, L, R))

get-value(x, set-value(x, v, node(y, w, L, R)))
  = ??

Don't know which rule of definition applies!

Need to continue by cases.

set-value(x, v, empty)	:=	node(x, v, empty, empty)			
set-value(x, v, node(y, w, L, R))	:=	node(x, v, L, R)	if $x = y$	get-value(x, node(y, w, L, R)) :=	v
set-value(x, v, node(y, w, L, R))	:=	node(y, w, set-value(x, v, L), R)	if $x < y$	get-value(x, node(y, w, L, R)) :=	get-value(x, L)
set-value(x, v, node(y, w, L, R))	:=	node(y, w, L, set-value(x, v, R))	if $y < x$	get-value(x, node(y, w, L, R)) :=	get-value(x, R)

Structural Induction on BSTs: Inductive Step (2/4)

P(T) := "get-value(x, set-value(x, v, T)) = v"

Inductive Step: prove P(node(y, w, L, R))

Suppose that $x = y$**.**

get-value(x, set-value(x, v, node(y, w, L, R)))	
= get-value(x, node(x, v, L, R))	**def of** set-value (**since** $x=y$)
= v	**def of** get-value

set-value(x, v, empty)	:= node(x, v, empty, empty)		
set-value(x, v, node(y, w, L, R))	:= node(x, v, L, R)	if $x = y$	get-value(x, node(y, w, L, R)) := v
set-value(x, v, node(y, w, L, R))	:= node(y, w, set-value(x, v, L), R)	if $x < y$	get-value(x, node(y, w, L, R)) := get-value(x, L)
set-value(x, v, node(y, w, L, R))	:= node(y, w, L, set-value(x, v, R))	if $y < x$	get-value(x, node(y, w, L, R)) := get-value(x, R)

Structural Induction on BSTs: Inductive Step (3/4)

P(T) := "get-value(x, set-value(x, v, T)) = v"

Inductive Step: **prove** P(node(y, w, L, R))

Suppose that x < y.

get-value(x, set-value(x, v, node(y, w, L, R)))	
= get-value(x, node(y, w, set-value(x, v, L), R))	**def of** set-value (**since** x<y)
= get-value(x, set-value(x, v, L))	**def of** get-value (**since** x<y)
= v	**Ind. Hyp.**

Suppose that x > y. **... (Analogous)**

set-value(x, v, empty)	:= node(x, v, empty, empty)			
set-value(x, v, node(y, w, L, R))	:= node(x, v, L, R)	if x = y	get-value(x, node(y, w, L, R))	:= v
set-value(x, v, node(y, w, L, R))	:= node(y, w, set-value(x, v, L), R)	if x < y	get-value(x, node(y, w, L, R))	:= get-value(x, L)
set-value(x, v, node(y, w, L, R))	:= node(y, w, L, set-value(x, v, R))	if y < x	get-value(x, node(y, w, L, R))	:= get-value(x, R)

Structural Induction on BSTs: Inductive Step (4/4)

P(T) := "get-value(x, set-value(x, v, T)) = v"

Inductive Step: prove P(node(y, w, L, R))

Suppose that x > y.

get-value(x, set-value(x, v, node(y, w, L, R)))	
= get-value(x, node(y, w, L, set-value(x, v, R)))	**def of** set-value (**since** x>y)
= get-value(x, set-value(x, v, R))	**def of** get-value (**since** x>y)
= v	**Ind. Hyp.**

set-value(x, v, empty) := node(x, v, empty, empty)		
set-value(x, v, node(y, w, L, R)) := node(x, v, L, R)	if x = y	get-value(x, node(y, w, L, R)) := v
set-value(x, v, node(y, w, L, R)) := node(y, w, set-value(x, v, L), R)	if x < y	get-value(x, node(y, w, L, R)) := get-value(x, L)
set-value(x, v, node(y, w, L, R)) := node(y, w, L, set-value(x, v, R))	if y < x	get-value(x, node(y, w, L, R)) := get-value(x, R)