• HW5 released later today, due Wednesday (5/1) at **11pm**
  – Please! start early and be prepared for a challenge!
  – Give yourself time to come to OH and ask questions on Ed
  – Working on the same issue for hours when you’re stuck won’t help, ask for help!

• Can resubmit as many times as you’d like until the deadline.
  – Use the autograder as a tool if you’re not sure if your code/tests have bugs
Abstraction Barrier – Review

- Specifications acts as the “barrier” between each side
  - improves understandability, changeability, and modularity
- Clients can only depend on the spec
- Implementer can write any code that satisfies the spec
Specifications for ADTs – Review

- New Terminology for specifying ADTs:
  - **Concrete State / Representation (Code)**
    - Actual fields of the record and the data stored
    - Ex: `{ list: List, last: bigint | undefined }`
  - **Abstract State / Representation (Math)**
    - How clients should understand the object
    - Ex: List(nil or cons)
Specifications for ADTs – Review

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

• Talk about functions in terms of the abstract state

• Hide the representation details (i.e. real fields) from the client
Documenting ADTs – Review

Abstract Function (AF) – defines what abstract state the field values represent
  – Maps field values → the object they represent
  – Output is math, this is a mathematical function

Representation Invariants (RI) – facts about the field values that must always be true
  – Constructor must always make sure RI is true at runtime
  – Can assume RI is true when reasoning about methods
  – AF only needs to make sense when RI holds
  – Must ensure that RI always holds
Documenting ADTs – Review

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

    // @ returns last(obj)
    getLast = (): bigint | undefined => {
        return this.last;
    }
}
```

Prove correctness of \( \text{last(obj)} = \text{this.last} \) using both

\[
\text{Last(obj)} = \text{last(this.list)} \quad \text{by AF}
\]
\[
= \text{this.last} \quad \text{by RI}
\]
Defining Interfaces

Typescript

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

Java

interface FastList {
    int getLast() throws EmptyList;
    List<Integer> toList();
}
The prefix `readonly` is used to make a property read-only
- Value cannot be changed
- Protects variables from unwanted mutations
- Should be our default

```typescript
class FastLastListImpl implements FastList {
    readonly last: bigint | undefined;
    readonly list: List<bigint>;
}
```
Abstract Data Class – Example

class FastLastListImpl implements FastList {
    readonly last: bigint | undefined;
    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 new record using “new”:

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

new FastLastListImpl(list);
Run `npm run start` in `sec-highlight` to check it out!

input the points:

```
100 100
100 300
300 100
300 300
```
Questions 1 & 2 – Recap

• From concrete implementation → ADT, writing specs shouldn’t be too hard
  – the specs already exist
  – just need to adjust what objects they’re operating on: parameters → ‘obj’
  – and add appropriate AF and RI

• Only did 1 in this example, but we’re able to have multiple classes implement the same interface, all with the same spec
  – Implementation can be switched out as needed, but expected inputs and behavior (spec) will be consistent
**Question 4**

Prove by structural induction that, for any left-leaning tree $T$, we have:

\[
\text{size}(T) \leq 2^{\text{height}(T)+1} - 1
\]

**Hints:**

1) Define the tree in your IH according to the definition of tree \`node(x, S, T)` so you can access the left and right trees

2) Remember the exponent rule: $x^y \times x = x^{y+1}$

```
func size(empty) := 0
size(node(x, S, T)) := 1 + size(S) + size(T)

func height(empty) := -1
height(node(x, S, T)) := 1 + height(S) for any $x : \mathbb{Z}$ and $S, T : \text{Tree}$
```
sep takes a list \( L \) and a value \( x \), and returns two lists, \( A \) containing all values \( \leq x \) and \( B \) containing all values \( \geq x \).

\[
\begin{array}{ll}
\text{func} & \text{sep}(\text{nil}, x) := (\text{nil}, \text{nil}) \\
& \text{sep}(\text{cons}(y, L), x) := (\text{cons}(y, A), B) \quad \text{if } y \leq x \\
& \text{sep}(\text{cons}(y, L), x) := (A, \text{cons}(y, B)) \quad \text{if } x < y \\
\end{array}
\]

\[
\text{where } (A, B) := \text{sep}(L, x)
\]

Note: in the recursive case, you:
- make a call to \( \text{sep}(L, x) \)
- take the return value of that call \( (A, B) \)
- \text{cons}(y \text{ on to } A \text{ or } B \text{ and returns } (A, \text{cons}(y, B)) \text{ or } (\text{cons}(y, A), B) \)
- Making an additional step to make our recursive result cleaner and avoid multiple recursive calls
Question 3

Prove by induction on $L$ that $\text{len}(A) + \text{len}(B) = \text{len}(L)$, where $(A, B) = \text{sep}(L, x)$