# CSE 331
# Software Design & Implementation

Autumn 2024

Section 8 – Trees and ADTs

# Administrivia

- HW 8 written released tonight at 4:30 pm, due Thursday 11/21 at 11 pm
  - It is only 1 problem, try to get it done by Wednesday night

- HW 8 code released Wed 11/20 at 4:30 pm, due Monday 11/25 at 11 pm
  - Longer code section than recent weeks, so start doing it early and come to office hours

# Proof By Calculation (Review)

- The goal of proof by calculation is to *show* that an assertion is true *given* facts that you already know
- You should ***start*** the proof with the left side of the assertion and ***end*** the proof with the right side of the assertion. Each symbol (=, >, <, etc.) connecting each line of the proof is that line's relationship to the previous line on the proof
- Only modify one side

**Example:**

Suppose we have the facts: $x = 3$, $y = 4$, $z > 5$ and we want to use proof by calculation to prove $x^2 + y^2 < z^2$. Our proof by calculation would look like this:

| | | |
|---|---|---|
| $x^2 + y^2$ | $= 3^2 + y^2$ | since $x = 3$ |
| | $= 3^2 + 4^2$ | since $y = 4$ |
| | $= 25$ | |
| | $= 5^2$ | |
| | $< z^2$ | since $z > 5$ |

note that each line shows the relationship to the previous line ONLY

start with left side of assertion

end with right side of assertion

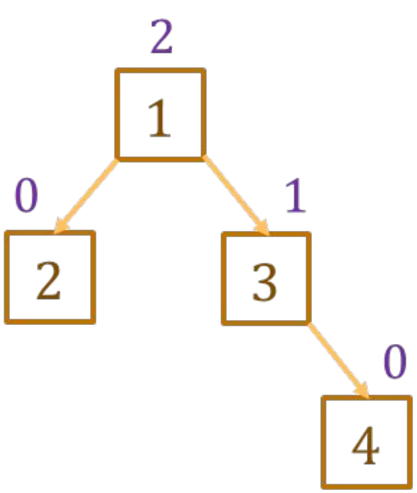
# Trees

- Trees are inductive data types with a constructor that has 2+ recursive arguments

- These come up all the time…
  - no constructors with recursive arguments = "generalized enums"
  - constructor with 1 recursive arguments = "generalized lists"
  - constructor with 2+ recursive arguments = "generalized trees"

# Height of a tree

- **Binary Tree:** a tree in which each node has at most 2 children
  - Not to be confused with Binary Search Tree, which also has the ordering property that (nodes in L) < x and (nodes in R) > x
- **type** Tree := empty | node(x: $\mathbb{Z}$, L: Tree, R: Tree)




Mathematical definition of height

$$\textbf{height : Tree} \rightarrow \mathbb{Z}$$

$$\begin{aligned} \text{height(empty)} &:= -1 \\ \text{height(node(x, L, R))} &:= 1 + \max(\text{height(L)}, \text{height(R)}) \end{aligned}$$

# Using Definitions in Calculations (example)

$$\textbf{height} : \textbf{Tree} \rightarrow \mathbb{Z}$$

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

$$\text{height}(\text{node}(x, L, R)) := 1 + \max(\text{height}(L), \text{height}(R))$$

- Suppose "T = node(1, empty, node(2, empty, empty))"
- Prove that height(T) = 1

height(T) = height(node(1, empty, node(2, empty, empty))) since T = …

= 1 + max(height(empty), height(node(2, empty, empty))) def of height

= 1 + max(-1, height(node(2, empty, empty))) def of height

= 1 + max(-1, 1+ max(height(empty), height(empty))) def of height

= 1 + max(-1, 1+ max(-1, -1)) def of height (x 2)

= 1 + max(-1, 1+ -1) def of max

= 1 + max(-1, 0)

= 1 + 0 def of max

= 1

# Task 1: One, Two, Tree…

The problem makes use of the following inductive type, representing a *left-leaning* binary tree

$$\begin{aligned} \textbf{type}\ \text{Tree} := &\quad \text{empty} \\ | &\quad \text{node}(\text{val} : \mathbb{Z},\ \text{left} : \text{Tree},\ \text{right} : \text{Tree}) \quad \text{with height(left)} \geqslant \text{height(right)} \end{aligned}$$

The "with" condition is an *invariant* of the node. Every node that is created must have this property,

$$\begin{aligned} \textbf{func}\ \text{height(empty)} &:= -1 \\ \text{height}(\text{node}(x, S, T)) &:= 1 + \text{height}(S) \quad \text{for any } x : \mathbb{Z} \text{ and } S, T : \text{Tree} \end{aligned}$$

Since height(S) >= height(T)

$$\text{size} : \text{Tree} \to \mathbb{N}$$

$$\begin{aligned} \text{size(empty)} &:= 0 \\ \text{size}(\text{node}(x, S, T)) &:= 1 + \text{size}(S) + \text{size}(T) \quad \text{for any } x : \mathbb{Z} \text{ and } S, T : \text{Tree} \end{aligned}$$

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

$$\text{size}(T) \leqslant 2^{\text{height}(T)+1} - 1$$

# Task 2: How do I Love Tree

a Path tells you how to get to a particular node where each step along the path (item in the list) would be a direction pointing you to keep going down the LEFT or RIGHT branch of the tree.

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

$$\textbf{type } \text{Dir} := \text{LEFT} \mid \text{RIGHT}$$

$$\textbf{type } \text{Path} := \text{List}\langle \text{Dir} \rangle$$

(a) Define a function "find($p$ : Path, $T$ : BST)" that returns the node (a BST) at the path from the root of $T$ or undefined if there is no such node.

(b) Define a function "remove($p$ : Path, $T$ : BST)" that returns $T$ except with the node at the given path replaced by empty.

# Specifications for ADTs – Review

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

- 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 string or a record (concrete)

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

# 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 – Example

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

Talk about functions in terms of the abstract state (obj)

Hide the representation details (i.e. real fields) from the client

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

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

# Task 3: Let's Blow This Point

Suppose we had the following interface and implementation to represent a point in 2D space:

```
/** Represents a point with coordinates in (x,y) space. */
interface Point {
    /** @returns the x coordinate of the point */
    getX: () => number;

    /** @returns the y coordinate of the point */
    getY: () => number;

    /**
     * Returns the distance of this point to the origin.
     * @returns Math.sqrt(obj.x*obj.x + obj.y*obj.y)
     */
    distToOrigin: () => number;
}
```

```
class SimplePoint implements Point {
    // RI: <TODO>
    // AF: <TODO>
    readonly x: number;
    readonly y: number;
    readonly r: number;

    // Creates a point with the given coordinates
    constructor(x: number, y: number) {
     this.x = x;
     this.y = y;
     this.r = Math.sqrt(x*x + y*y);
    }

    getX = (): number => this.x;
    getY = (): number => this.y;
    distToOrigin = (): number => this.r;
}
```

(a) Define the representation invariant (RI) and abstraction function (AF) for the SimplePoint class.

# Task 3: Let’s Blow This Point

```
/** Represents a point with coordinates in (x,y) space. */
interface Point {
    /** @returns the x coordinate of the point */
    getX: () => number;

    /** @returns the y coordinate of the point */
    getY: () => number;

    /**
     * Returns the distance of this point to the origin.
     * @returns Math.sqrt(obj.x*obj.x + obj.y*obj.y)
     */
    distToOrigin: () => number;
}
```

```
class SimplePoint implements Point {
    // RI: <TODO>
    // AF: <TODO>
    readonly x: number;
    readonly y: number;
    readonly r: number;

    // Creates a point with the given coordinates
    constructor(x: number, y: number) {
     this.x = x;
     this.y = y;
     this.r = Math.sqrt(x*x + y*y);
    }

    getX = (): number => this.x;
    getY = (): number => this.y;
    distToOrigin = (): number => this.r;
}
```

(b) Use the RI or AF to prove that the distToOrigin method of the SimplePoint class is correct.

# Task 3: Let's Blow This Point

(c) The following problem will make use of this math definition that rotates a point around the origin $(x, y)$ by an angle $\theta$:

$$\text{rotate} : (\text{Point}, \ \mathbb{R}) \rightarrow \text{Point}$$

$$\text{rotate}((x, y), \theta) \quad = \quad (x \cdot \cos(\theta) - y \cdot \sin(\theta), \ x \cdot \sin(\theta) + y \cdot \cos(\theta))$$

Suppose we have the following implementation of the rotate method:

/** @returns rotate(obj, $\theta$) */

```
rotate = (theta: number): Point => {
    const newX = this.x * Math.cos(theta) - this.y * Math.sin(theta);
    const newY = this.x * Math.sin(theta) + this.y * Math.cos(theta);
    return new SimplePoint(newX, newY);
}
```

Prove that the rotate method is correct using the RI or AF.

# Task 4: Going Back and Length

$$\text{len} : \text{List} \to \mathbb{N}$$

$$\text{len}(\text{nil}) := 0$$
$$\text{len}(x :: L) := 1 + \text{len}(L)$$

$$\text{rev} : \text{List} \to \text{List}$$

$$\text{rev}(\text{nil}) := \text{nil}$$
$$\text{rev}(x :: L) := \text{rev}(L) +\!+ [x]$$

Lemma 1:
len(rev($L$) :: $x$) =
len(rev($L$)) + len($x$::nil)
for any list $L$ and element $x$

4. Prove by Structural Induction that len(rev($L$)) = len($L$) for any list $L$. You may use Lemma 1 in your proof.

*ok to work from top and bottom as long as only modifying right side!