CSE 331: Software Design & Implementation Spring 2025

Quiz Section 8: Trees – Solutions

Task 1 – One, Two, Tree...

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

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

The "with" condition is an *invariant* of the node. Every node that is created must have this property, and we are allowed to use the fact that it holds in our reasoning.

The height of a tree is defined recursively by

$$\mathsf{height} : \mathsf{Tree} \to \mathbb{Z}$$

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

In a general binary tree, the height of a non-empty tree is the length of the *longest* path to a leaf. With a left-leaning tree, we know the longest path is the one that always travels toward the left child.

We can define the size of a tree, the number of values stored in it, as follows:

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

$$\begin{aligned} \mathsf{size}(\mathsf{empty}) &:= 0 \\ \mathsf{size}(\mathsf{node}(x, S, R)) &:= 1 + \mathsf{size}(S) + \mathsf{size}(R) \end{aligned}$$

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

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

Define $P(T)$ to be the claim that $\mathsf{size}(T) \leqslant 2^{\mathsf{height}(T)+1} - 1$. We will prove this by structural induction.

Base Case (empty). In this case, we can see that

$$\begin{aligned} & \mathsf{size}(\mathsf{empty}) \\ &\quad = 0 && \text{Def of size} \\ &\quad = 1 - 1 \\ &\quad = 2^0 - 1 \\ &\quad = 2^{-1+1} - 1 \\ &\quad = 2^{\mathsf{height}(\mathsf{empty})+1} - 1 && \text{Def of height} \end{aligned}$$

Inductive Hypothesis. Suppose that P holds for trees S and R.

Inductive Step. We need to show $P(\text{node}(x, S, R))$ for any integer x.

Let x be any integer. Then, we can see that

$$\begin{aligned}
&\text{size}(\text{node}(x, S, R)) && \\
&\quad = 1 + \text{size}(S) + \text{size}(R) && \text{Def of size} \\
&\quad \leqslant 1 + 2^{\text{height}(S)+1} - 1 + 2^{\text{height}(R)+1} - 1 && \text{Inductive Hypothesis} \\
&\quad = 2^{\text{height}(S)+1} + 2^{\text{height}(R)+1} - 1 && \\
&\quad \leqslant 2 \cdot 2^{\text{height}(S)+1} - 1 && \text{since height}(S) \geqslant \text{height}(R) \\
&\quad = 2 \cdot 2^{\text{height}(\text{node}(x,S,R))} - 1 && \text{Def of height} \\
&\quad = 2^{\text{height}(\text{node}(x,S,R))+1} - 1 &&
\end{aligned}$$

Conclusion. $P(T)$ holds for any left-leaning tree T by structural induction.

Task 2 – How Do I Love Tree, Let Me Count the Ways

The following is the definition of a binary search tree:

$$\begin{aligned} \textbf{type}\ \mathsf{BST} :=\ & \mathsf{empty} \\ \mid\ & \mathsf{node}(x : \mathbb{Z},\ S : \mathsf{BST},\ R : \mathsf{BST}) \end{aligned}$$

Suppose that we wanted to have a way to refer to a specific node in a BST. One way to do so would be to give directions from the root to that node. If we define these types:

$$\begin{aligned} \textbf{type}\ \mathsf{Dir} &:= \mathsf{LEFT}\ \mid\ \mathsf{RIGHT} \\ \textbf{type}\ \mathsf{Path} &:= \mathsf{List}\langle\mathsf{Dir}\rangle \end{aligned}$$

then 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.

For example, LEFT :: RIGHT :: nil says to select the "LEFT"child of the parent and then the "RIGHT" child of that node, giving us a grand-child of the root node.

(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.

$$\mathsf{find} : (\mathsf{Path},\ \mathsf{BST}) \to \mathsf{BST}$$

$$\begin{aligned} \mathsf{find}(\mathsf{nil}, T) &:= T \\ \mathsf{find}(d :: L, \mathsf{empty}) &:= \mathsf{undefined} \\ \mathsf{find}(\mathsf{LEFT} :: L, \mathsf{node}(x, S, R)) &:= \mathsf{find}(L, S) \\ \mathsf{find}(\mathsf{RIGHT} :: L, \mathsf{node}(x, S, R)) &:= \mathsf{find}(L, R) \end{aligned}$$

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

$$\mathsf{remove} : (\mathsf{Path},\ \mathsf{BST}) \to \mathsf{BST}$$

$$\begin{aligned} \textbf{func}\ \mathsf{remove}(\mathsf{nil}, T) &:= \mathsf{empty} \\ \mathsf{remove}(d :: L, \mathsf{empty}) &:= \mathsf{undefined} \\ \mathsf{remove}(\mathsf{LEFT} :: L, \mathsf{node}(x, S, R)) &:= \mathsf{node}(x, \mathsf{remove}(L, S), R) \\ \mathsf{remove}(\mathsf{RIGHT} :: L, \mathsf{node}(x, S, R)) &:= \mathsf{node}(x, S, \mathsf{remove}(L, R)) \end{aligned}$$

Task 3 – Let's Blow This Point

Suppose we had the following interface for a Point class that represents a point in $\mathbb{R}^2$ (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;
}
```

The following is an implementation of the Point interface:

```
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) in the form r = "..." and abstraction function (AF) in the form obj = "..." for the SimplePoint class.

RI: r = Math.sqrt(this.x * this.x + this.y * this.y)
AF: obj = (this.x, this.y)

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

We can see that:

$$\begin{aligned} &\text{Math.sqrt(obj.x*obj.x + obj.y*obj.y)} \\ &= \text{Math.sqrt(this.x*this.x + this.y*this.y)} && \text{by AF} \\ &= \text{this.r} && \text{by RI} \end{aligned}$$

Our function returns this.r, so we know that it is correct.

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

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

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

Suppose we have the following implementation of the rotate method:

/** @returns rotate(obj, θ) */

```
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.

We can see that:

$$\begin{aligned} \text{rotate}(obj, \theta) &= \text{rotate((this.x, this.y)}, \theta) && \text{by AF} \\ &= (\text{this.x} \cdot cos(\theta) - \text{this.y} \cdot sin(\theta), \text{this.x} \cdot sin(\theta) + \text{this.y} \cdot cos(\theta)) && \text{def of rotate} \\ &= (\text{newX}, \text{this.x} \cdot sin(\theta) + \text{this.y} \cdot cos(\theta)) && \text{def of newX} \\ &= (\text{newX}, \text{newY}) && \text{def of newY} \end{aligned}$$

Our function returns new SimplePoint(newX, newY), so we know that it is correct.