Discrete Structures for Computer Science

William Garrison
bill@cs.pitt.edu
6311 Sennott Square

Lecture #27: Recursion and Structural Induction

University of Pittsburgh

There are many uses of induction in computer science!

Proof by induction is often used to reason about:

- Algorithm properties (correctness, etc.)
- Properties of data structures
- Membership in certain sets
- Determining whether certain expressions are well-formed
- …

To begin looking at how we can use induction to prove the above types of statements, we first need to learn about recursion

Sometimes, it is difficult or messy to define some object explicitly

Recursive objects are defined in terms of (other instances of) themselves

We often see the recursive versions of the following types of objects:

- Functions
- Sequences
- Sets
- Data structures

Let's look at some examples…

Recursive functions are useful

When defining a recursive function whose domain is the set of natural numbers, we have two steps:

1. Basis step: Define the behavior of $f(0)$
2. Recursive step: Compute $f(n+1)$ using $f(0), \ldots, f(n)$

Doesn't this look a little bit like strong induction?

Example: Let $f(0) = 3$, $f(n+1) = 2f(n) + 3$

- $f(1) = 2f(0) + 3 = 2(3) + 3 = 9$
- $f(2) = 2f(1) + 3 = 2(9) + 3 = 21$
- $f(3) = 2f(2) + 3 = 2(21) + 3 = 45$
- $f(4) = 2f(3) + 3 = 2(45) + 3 = 93$
- …

Some functions can be defined more precisely using recursion

Example: Define the factorial function F(n) recursively

1. Basis step: $F(0) = 1$
2. Recursive step: $F(n+1) = (n+1) \times F(n)$

Note: $F(4) = 4 \times F(3)$
$= 4 \times 3 \times F(2)$
$= 4 \times 3 \times 2 \times F(1)$
$= 4 \times 3 \times 2 \times 1 \times F(0)$
$= 4 \times 3 \times 2 \times 1 \times 1 = 24$

The recursive definition avoids using the "…" shorthand!

Compare the above definition our old definition:

- $F(n) = n \times (n-1) \times \ldots \times 2 \times 1$

It should be no surprise that we can also define recursive sequences

Example: The Fibonacci numbers, $\{f_n\}$, are defined as follows:

- $f_0 = 0$
- $f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}$

This is like strong induction, since we need more than f_{n-1} to compute f_n.

Calculate: f_2, f_3, f_4, and f_5

- $f_2 = f_1 + f_0 = 0 + 1 = 1$
- $f_3 = f_2 + f_1 = 1 + 1 = 2$
- $f_4 = f_3 + f_2 = 2 + 1 = 3$
- $f_5 = f_4 + f_3 = 3 + 2 = 5$

This gives us the sequence $\{f_n\} = 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

Recursion is used heavily in the study of strings

Let: Σ be defined as an alphabet

- Binary strings: $\Sigma = \{0, 1\}$
- Lower case letters: $\Sigma = \{a, b, c, \ldots, z\}$

We can define the set Σ^* containing all strings over the alphabet Σ as follows:

λ is the empty string containing no characters

1. Basis step: $\lambda \in \Sigma^*$
2. Recursive step: If $w \in \Sigma^*$ and $x \in \Sigma$, then $wx \in \Sigma^*$

Example: If $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\lambda, 0, 1, 01, 11, \ldots\}$

This recursive definition allows us to easily define important string operations

Definition: The concatenation of two strings can be defined as follows:

1. Basis step: if $w \in \Sigma^*$, then $w \diamond \lambda = w$
2. Recursive step: if $w_1 \in \Sigma^*$, $w_2 \in \Sigma^*$, and $x \in \Sigma$, then $w_1 \diamond (w_2 x) = (w_1 \diamond w_2)x$

Example: Concatenate the strings "Hello" and "World"

1. Hello⋄World = (Hello⋄Worl)d
2. = (Hello⋄Wor)ld
3. = (Hello⋄Wo)rld
4. = (Hello⋄W)orld
5. = (Hello⋄λ)World
6. = HelloWorld

This recursive definition allows us to easily define important string operations

Definition: The length $l(w)$ of a string can be defined as follows:

1. Basis step: $l(\lambda) = 0$
2. Recursive step: $l(wx) = l(w) + 1$ if $w \in \Sigma^*$ and $x \in \Sigma$

Example:
$$\begin{aligned} l(1001) &= l(100) + 1 \\ &= l(10) + 1 + 1 \\ &= l(1) + 1 + 1 + 1 \\ &= l(\lambda) + 1 + 1 + 1 + 1 \\ &= 0 + 1 + 1 + 1 + 1 \\ &= 4 \end{aligned}$$

We can define sets of well-formed formulae recursively

This is often used to specify the operations permissible in a given formal language (e.g., a programming language)

Example: Defining propositional logic

1. Basis step: $\top$, $\bot$, and *s* are well-formed propositional logic statements (where *s* is a propositional variable)
2. Recursive step: If E and F are well-formed statements, so are
   - $(\neg E)$
   - $(E \wedge F)$
   - $(E \vee F)$
   - $(E \rightarrow F)$
   - $(E \leftrightarrow F)$

Example

Question: Is $((p \wedge q) \rightarrow (((\neg r) \vee q) \wedge t))$ well-formed?

- Basis tells us that p, q, r, t are well-formed
- 1st application: $(p \wedge q)$, $(\neg r)$ are well-formed
- 2nd application: $((\neg r) \vee q)$ is well-formed
- 3rd application: $(((\neg r) \vee q) \wedge t)$ is well-formed
- 4th application: $((p \wedge q) \rightarrow (((\neg r) \vee q) \wedge t))$ is well-formed

In-class exercises

Problem 1: Consider $f: \mathbf{Z} \times \mathbf{N} \rightarrow \mathbf{Z}$ where $f(x, y) = x^y$. Construct a recursive definition for $f(x, y)$ that does not use exponentiation.

Problem 2: Top Hat

Like other forms of induction, structural induction requires that we consider two cases

Basis step: Show that the result holds for the objects specified in the basis case of the recursive definition

Recursive step: Show that if the result holds for the objects used to construct new elements using the recursive step of the definition (the inputs), then it holds for the new object (the output) as well.

To see how this works, let's revisit string length…

Recall from earlier…

Definition: The length $l(w)$ of a string can be defined as follows:

1. Basis step: $l(\lambda) = 0$
2. Recursive step: $l(wx) = l(w) + 1$ if $w \in \Sigma^*$ and $x \in \Sigma$

Example: $l(1001) = l(100) + 1$

$$= l(10) + 1 + 1$$

$$= l(1) + 1 + 1 + 1$$

$$= l(\lambda) + 1 + 1 + 1 + 1$$

$$= 0 + 1 + 1 + 1 + 1$$

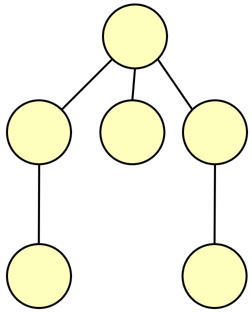
$$= 4$$

Prove that $l(x \diamond y) = l(x) + l(y)$ for $x,y \in \Sigma^*$

$P(n) \equiv l(x \diamond y) = l(x) + l(y)$ whenever $x \in \Sigma^*$ and $l(y) = n$
Base case: $P(0)$: $l(x \diamond \lambda) = l(x) = l(x) + 0 = l(x) + l(\lambda)$ ✔
I.H.: Assume that $P(k)$ holds for an arbitrary integer k
Inductive step: We will now show that $P(k) \rightarrow P(k+1)$ ■ Consider the string $x \diamond ya$, where $x,y \in \Sigma^*$, $a \in \Sigma$ and $l(y) = k$ ■ $l(x \diamond ya) = l(x \diamond y) + 1$ by the recursive definition of $l()$ ■ $= l(x) + l(y) + 1$ by the I.H. ■ Since $l(ya) = l(y) + 1$ by the recursive defintion of $l()$, we have that $l(x \diamond ya) = l(x) + l(ya)$, where ya is a string of size $k+1$
Conclusion: Since we have proved the base case and the inductive case, the claim holds for all n by structural induction ❑

Many common data structures used in computer science have recursive definitions

Example: Rooted Trees

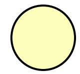


Base step: A single node is a rooted tree

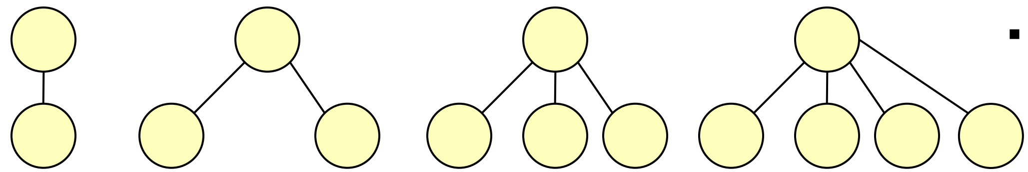
Recursive step: If T_1, T_2, …, T_n are disjoint rooted trees with roots r_1, r_2, …, r_n then introducing a new root r connected to r_1, r_2, …, r_n forms a new rooted tree.

Example Rooted Trees

Base case:

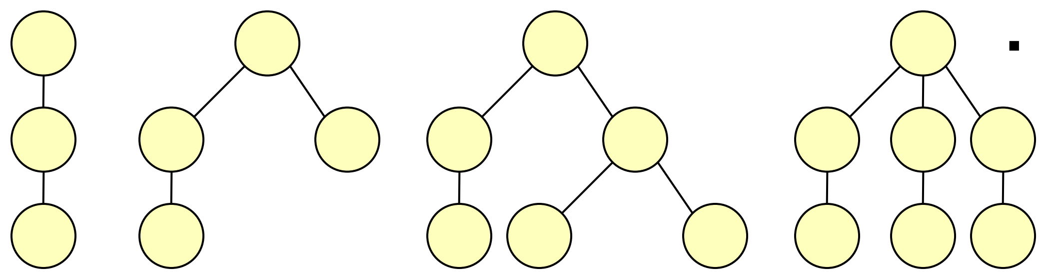


One application:



…

Two applications:

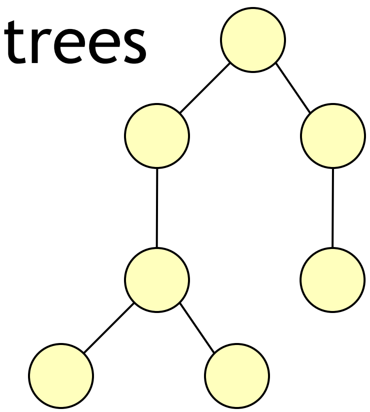


…

…

Many common data structures used in computer science have recursive definitions

Example: Extended binary trees



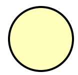
Base step: The empty set is an extended binary tree

Recursive step: If T_1 and T_2 are disjoint extended binary trees with roots r_1 and r_2, then introducing a new root r connected to r_1 and r_2 forms a new extended binary tree.

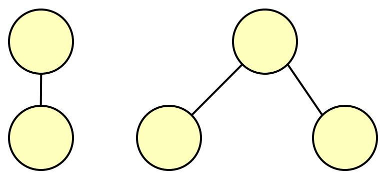
Example Extended Binary Trees

Base case: $\emptyset$

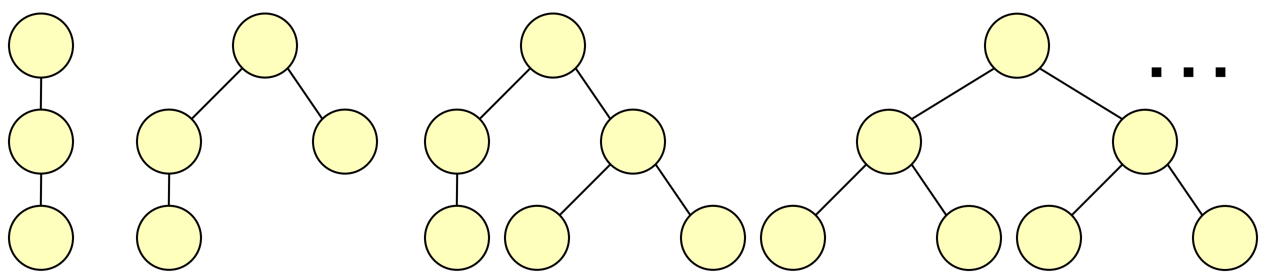
Step 1:



Step 2:



Step 3:

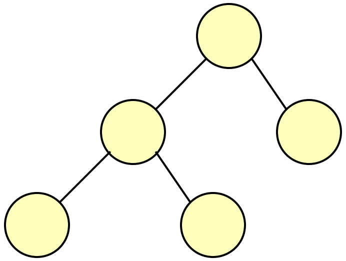


…

…

Many common data structures used in computer science have recursive definitions

Example: Full binary trees

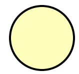


Base step: A single root node r is a full binary tree

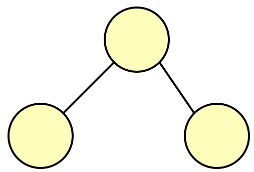
Recursive step: If T_1 and T_2 are disjoint full binary trees with roots r_1 and r_2, then introducing a new root r connected to r_1 and r_2 forms a new full binary tree.

Example Full Binary Trees

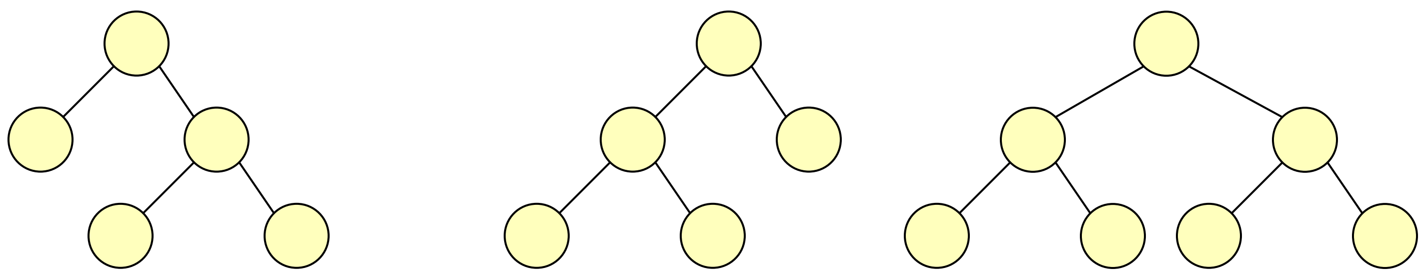
Base case:



Step 1:



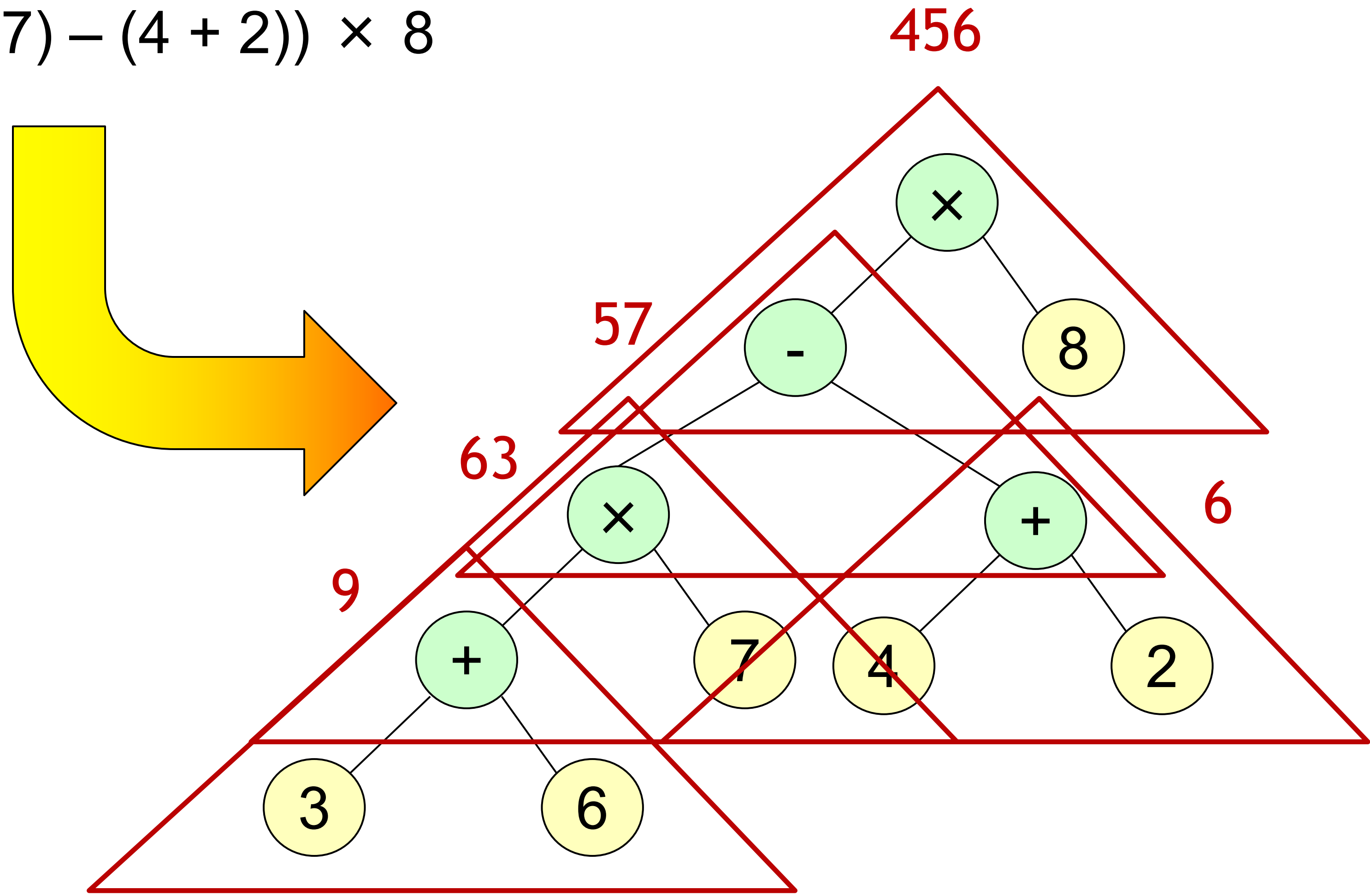
Step 2:



...

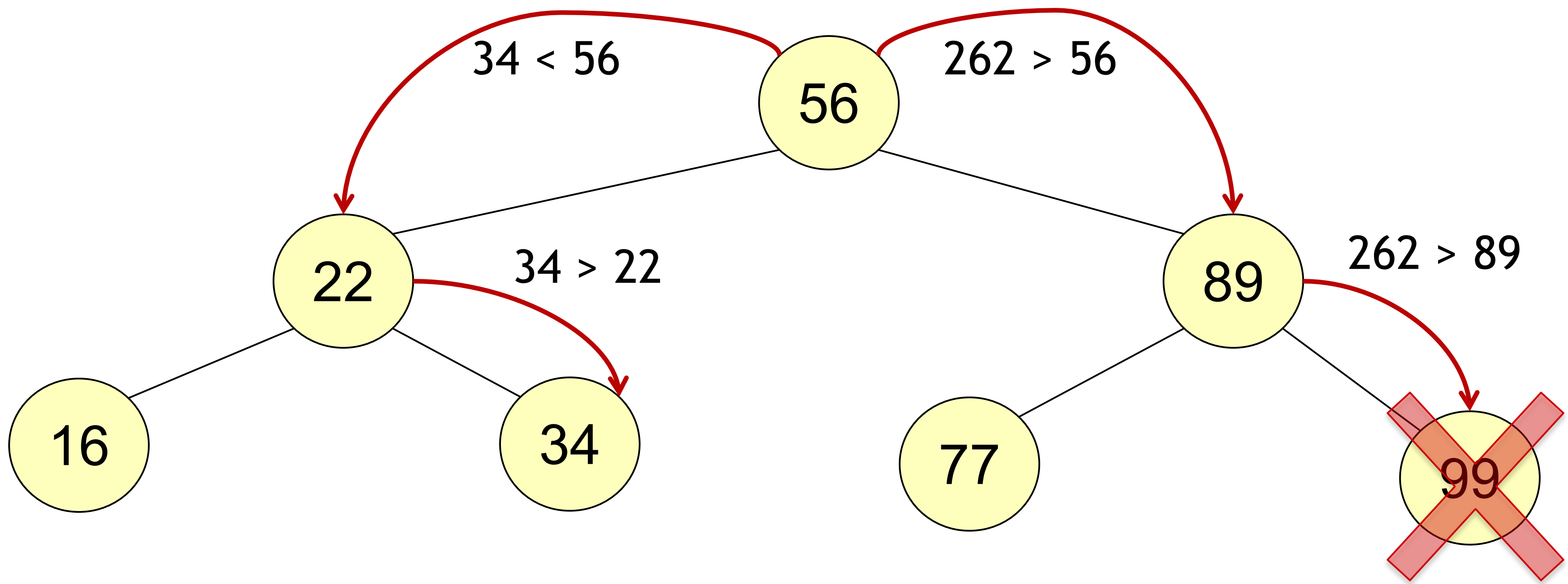
Trees are used to parse expressions

$(((3 + 6) \times 7) - (4 + 2)) \times 8$



Trees are used to enable fast searches

Consider the set S = {56, 22, 34, 89, 99, 77, 16}



Question: Is 34 ∈ S?

YES!

Question: Is 262 ∈ S?

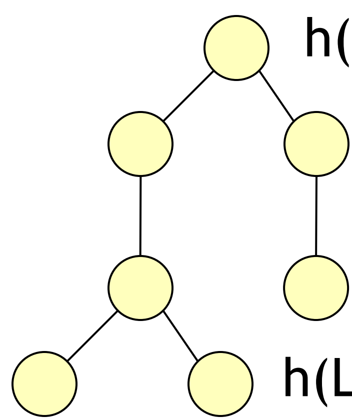
NO!

As with other recursively defined objects, we can define many properties of trees recursively

Definition: Given a tree T, we can define the height of T recursively, as follows:

1. Basis step: If T consists only of the root node r, then $h(T) = 0$
2. Recursive step: If T consists of a root r that connects to subtrees $T_1, \ldots, T_n$, then $h(T) = 1 + \max(h(T_1), \ldots, h(T_n))$

Example: What is the height of this tree T?



$h(T) = 1 + \max(h(L), h(R))$

$h(L) = 1 + h(L_1)$

$h(R) = 1 + h(R_1)$

$h(L_1) = 1 + \max(h(L_{11}), h(L_{12}))$

$h(R_1) = 0$

$h(L_{11}) = 0$

$h(L_{12}) = 0$

If T is a full binary tree, then the number of nodes in T (denoted n(T)) is less than or equal to $2^{h(T)+1}-1$

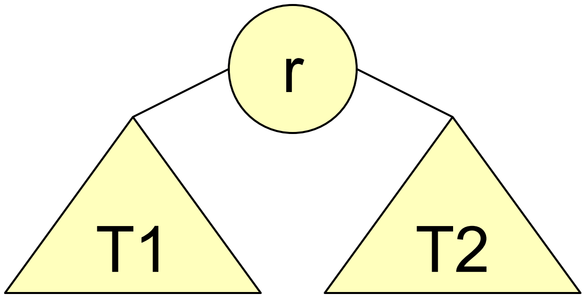
Claim: $n(T) \le 2^{h(T)+1}-1$

Base case: T contains only a root node. In this case n(T) = 1 and h(T) = 0. Note that $2^{0+1}-1 = 1$, so the claim holds.

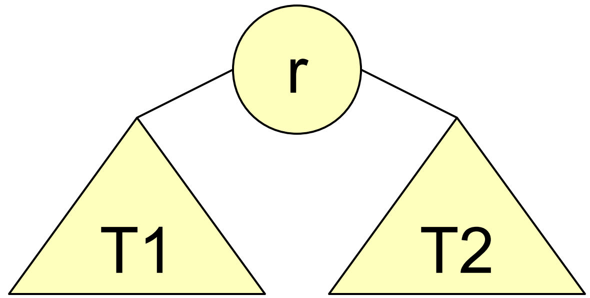
I.H. (Strong): Assume that claim holds for a tree of height ≤k, arb. k

Inductive step: Show that the claim holds for trees of height k+1

- Let T1 and T2 be disjoint full binary trees of height ≤k
- By the I.H., $n(T1) \le 2^{h(T1)+1}-1$ and $n(T2) \le 2^{h(T2)+1}-1$
- Let r be a unique root element, and let T be the tree formed using r as a root, T1 as the left subtree of r, and T2 as the right subtree of r



If T is a full binary tree, then the number of nodes in T (denoted n(T)) is less than or equal to $2^{h(T)+1}-1$



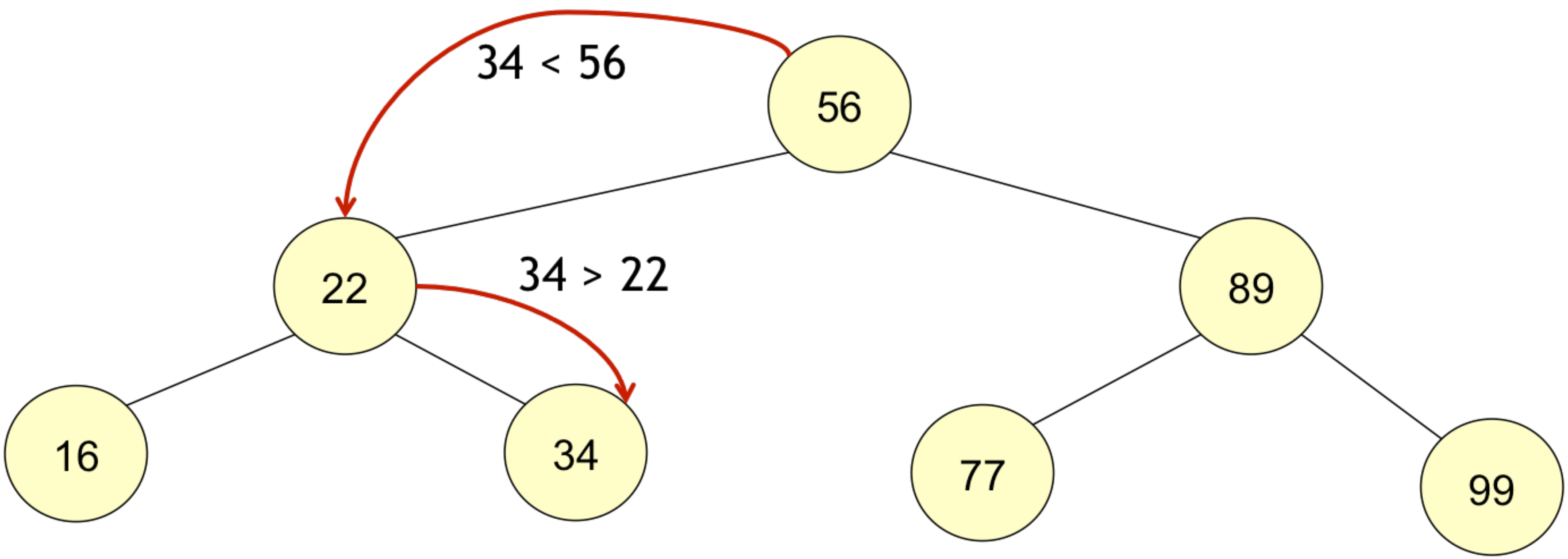
Inductive step (cont.): We have that

- $n(T) = 1 + n(T1) + n(T2)$ — by recursive formula of n(T)
- $\leq 1 + 2^{h(T1)+1} - 1 + 2^{h(T2)+1} - 1$ — by I.H.
- $\leq 2^{h(T1)+1} + 2^{h(T2)+1} - 1$
- $\leq 2 \times \max(2^{h(T1)+1}, 2^{h(T2)+1}) - 1$ — sum of 2 terms ≤ twice larger term
- $\leq 2 \times 2^{\max(h(T1),\, h(T2))+1} - 1$ — $\max(2^x, 2^y) = 2^{\max(x,y)}$
- $\leq 2 \times 2^{h(T)} - 1$ — by recursive def'n of h(T)
- $\leq 2^{h(T)+1} - 1$

Conclusion: Since we have proved the base case and the inductive case, the claim holds by structural induction ❑

In-class exercises

Problem 3: Use structural induction to prove that checking whether some number is contained in a binary search tree T involves at most h(T)+1 comparison operations.



Final Thoughts

- Structural induction can be used to prove properties of recursive
  - Functions
  - Sequences
  - Sets
  - Data structures
- Next time, our final review session!