Programming Languages and Techniques (CIS120)

Lecture 5

Datatypes and Trees

Recap: Lists, Recursion, & Tuples

Structural Recursion Over Lists

Structural recursion builds an answer from smaller components:

```
let rec f (l : … list) … : … =
  begin match l with
  | [] -> …
  | ( hd :: rest ) -> … f rest …
  end
```

The branch for `[ ]` calculates the value (`f [ ]`) directly.

- this is the *base case* of the recursion

The branch for `hd::rest` calculates

(`f(hd::rest)`) given `hd` and (`f rest`).

- this is the *inductive case* of the recursion

What is the result of this expression?

```
f [1; 2] [3;4
```

```
let rec f (l1:int list) (l2:int list) : int list =
    begin match l1 with
    | [] -> l2
    | x::xs -> x :: f xs l2
    end
```

f [1; 2] [3;4]

⇒ 1 :: (f [2] [3;4])

⇒ 1 :: 2 :: (f [] [3;4])

⇒ 1 :: 2 :: [3;4]

= [1;2;3;4]

What is the type of this expression?

```
[ 1 ]
```

1. int
2. int list
3. int list list
4. (int * int list) list
5. int * (int list)
6. (int * int) list
7. *none (expression is ill typed)*

Answer: 2

What is the type of this expression?

```
(1, [1])
```

1. int
2. int list
3. int list list
4. (int * int list) list
5. int * (int list)
6. (int * int) list
7. *none (expression is ill typed)*

Answer: 5

What is the type of this expression?

```
(1, [1], [[1]])
```

1. int
2. int list
3. int list list
4. (int * int list) list
5. int * (int list) * (int list list)
6. (int * int * int) list
7. *none (expression is ill typed)*

Answer: 5

What is the type of this expression?

```
[ (1,true); (0, false) ]
```

1. int * bool
2. int list * bool list
3. (int * bool) list
4. (int * bool) list list
5. *none (expression is ill typed)*

Answer: 3

What is the type of this expression?

```
(1 :: [], 2 :: [], 3 :: [])
```

1. int
2. int list
3. int list list
4. int list * int list * int list
5. int * int list * int list list
6. (int * int * int) list
7. *none (expression is ill typed)*

Answer: 4

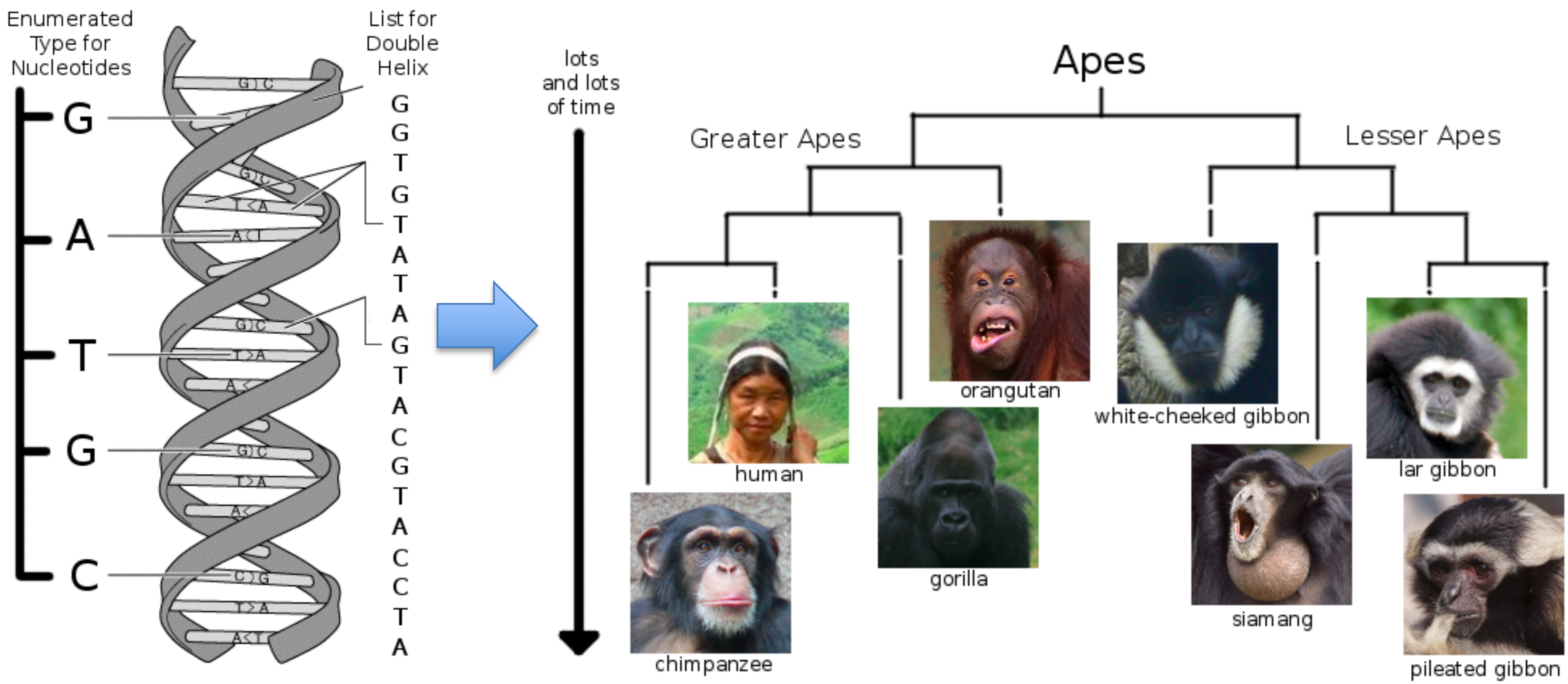
Datatypes and Trees

Datatypes

- Programming languages provide a variety of ways of creating and manipulating structured data
- We have already seen:
  - *primitive datatypes* (int, string, bool, …)
  - *lists* (int list,   string list, string list list, …)
  - *tuples* (int * int, int * string, …)
- Today:
  - *user-defined* datatypes

HW 2 Case Study: Evolutionary Trees

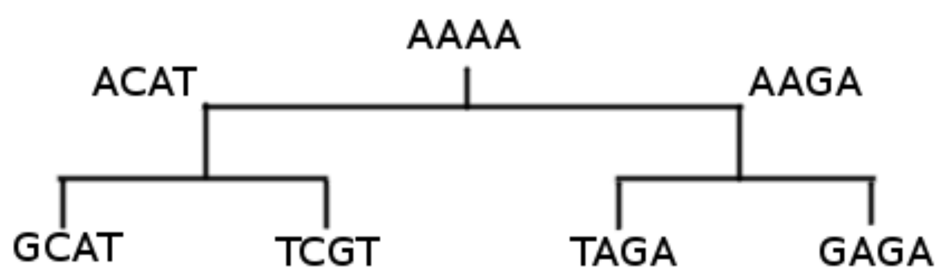
- Problem: reconstruct evolutionary trees* from DNA data.
  - What are the relevant abstractions?
  - How can we use the language features to define them?
  - How do the abstractions help shape the program?



*Interested? Check this out:
Dawkins*: The Ancestor's Tale: A Pilgrimage to the Dawn of Evolution*

DNA Computing Abstractions

- Nucleotide
  - Adenine (A), Guanine (G), Thymine (T), or Cytosine (C)
- Helix
  - a sequence of nucleotides: e.g. AGTCCGATTACAGAGA…
  - genetic code for a particular species (human, gorilla, etc)
- Phylogenetic tree
  - Binary tree with helices (species) at the nodes and leaves



Simple User-Defined Datatypes

- OCaml lets programmers define *new* datatypes

```
type day =
    | Sunday
    | Monday
    | Tuesday
    | Wednesday
    | Thursday
    | Friday
    | Saturday
```

‘type’ keyword

type name (must be lowercase)

```
type nucleotide =
| A
| C
| G
| T
```

constructor names (*tags*) (*must* be capitalized)

- The constructors *are* the values of the datatype
  - e.g. `A` is a nucleotide and `[A; G; C]` is a nucleotide list

Pattern Matching Simple Datatypes

- Datatype values can be analyzed by pattern matching:

```
let string_of_n (n:nucleotide) : string =
  begin match n with
  | A -> “adenine”
  | C -> “cytosine”
  | G -> “guanine”
  | T -> “thymine”
  end
```

- One case per constructor
  - you will get a warning if you leave out a case or list one twice
- As with lists, the pattern syntax follows that of the datatype values (i.e. the constructors)

A Point About Abstraction

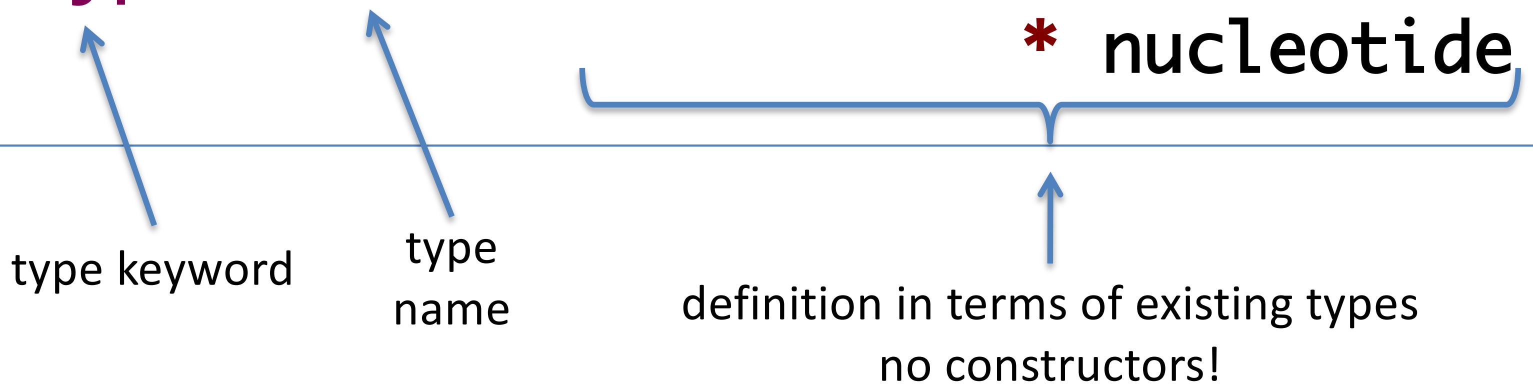
- We *could* represent data like this by using integers:
  - Sunday = 0, Monday = 1, Tuesday = 2, etc.
- But:
  - Integers support different operations than days do:
    Wednesday - Monday = Tuesday (?!)
  - There are *more* integers than days (What day is 17?)
- Confusing integers with days can lead to bugs
  - Many "scripting" languages (PHP, Javascript, Perl, Python,…) violate such abstractions (true == 1 == "1"), leading to pain and misery…

Most modern languages (Java, C#, C++, Rust, Swift,…) provide user-defined types for these reasons

Type Abbreviations

- OCaml also lets us *name* types without making new abstractions:

```
type helix = nucleotide list
type codon = nucleotide * nucleotide
                        * nucleotide
```



- i.e. a `codon` is the same thing a triple of `nucleotides`

```
let x : codon = (A,C,C)
```

- Can make code easier to read & write

Data-Carrying Constructors

- Datatype constructors can also carry values

```
type measurement =
  | Missing
  | NucCount   of nucleotide * int
  | CodonCount of codon * int
```

keyword 'of'

Constructors may take a tuple of arguments

- Values of type '**measurement**' include:

```
Missing
NucCount(A, 3)
CodonCount((A,G,T), 17)
```

Pattern Matching Datatypes

- Pattern matching notation combines syntax of tuples and simple datatype constructors:

```
let get_count (m:measurement) : int =
  begin match m with
  | Missing            -> 0
  | NucCount(_, n)     -> n
  | CodonCount(_, n)   -> n
  end
```

- Datatype patterns *bind* identifiers (e.g. 'n') just like lists and tuples

```
type nucleotide = | A | C | G | T
type helix = nucleotide list
```

What is the type of this expression?

```
[A;C]
```

1. nucleotide
2. helix
3. nucleotide list
4. string * string
5. nucleotide * nucleotide
6. *none (expression is ill typed)*

Answer: both 2 and 3

```
type nucleotide = | A | C | G | T
type helix = nucleotide list
```

What is the type of this expression?

```
(A, “A”)
```

1. nucleotide
2. nucleotide list
3. helix
4. nucleotide * string
5. string * string
6. *none (expression is ill typed)*

Answer: 4

Recursive User-defined Datatypes

- Datatype definitions can mention themselves *recursively*:

```
type tree =
  | Leaf of helix
  | Node of tree * helix * tree
```

base constructor (nonrecursive)

Node carries a tuple of values

recursive occurrences of datatype being defined

Syntax for User-defined Types

```
type tree =
  | Leaf of helix
  | Node of tree * helix * tree
```

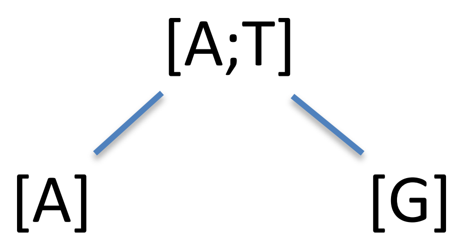
- Example values of type `tree`

```
let t1 = Leaf [A;G]
let t2 = Node (Leaf [G], [A;T], Leaf [A])
let t3 =
  Node (Leaf [T],
        [T;T],
        Node (Leaf [G;C], [G], Leaf []))
```

Constructors
(note capitalization)

```
type tree =
  | Leaf of helix
  | Node of tree * helix * tree
```

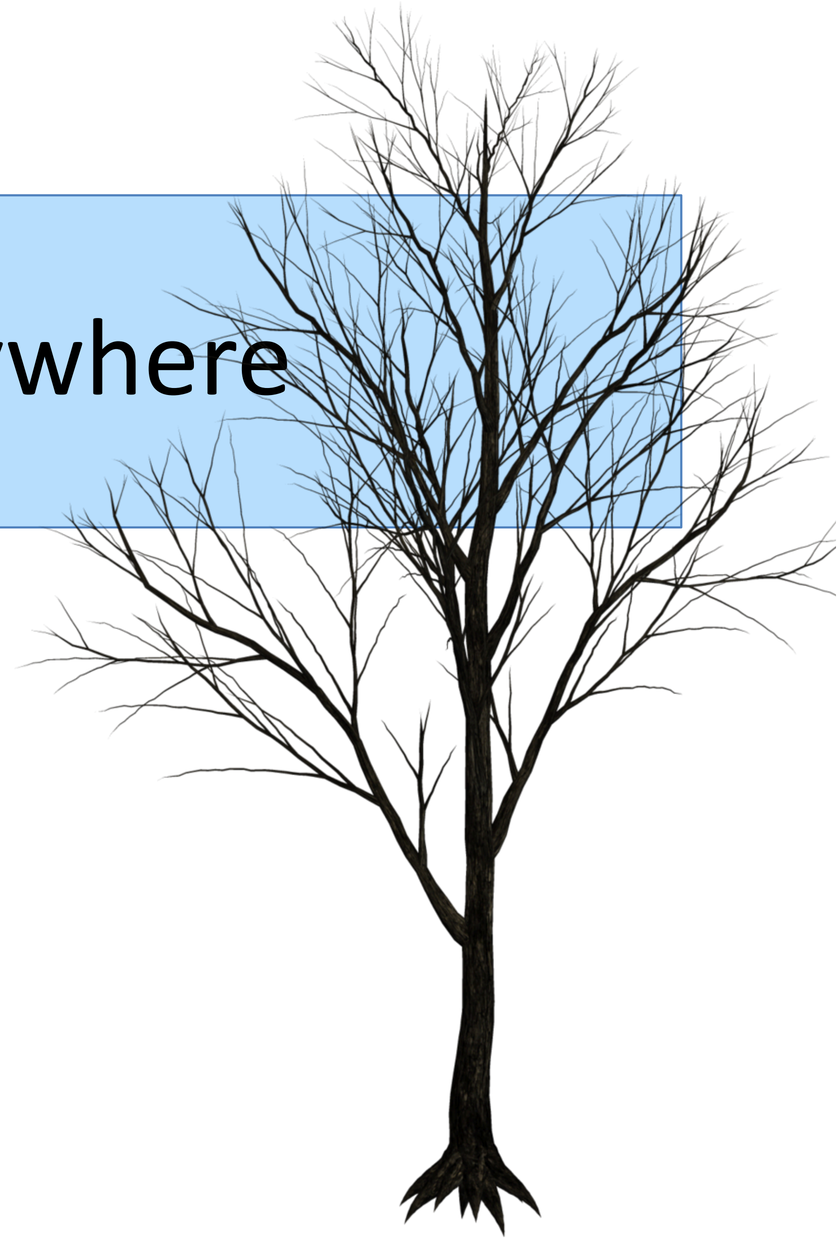
How would you construct this tree in OCaml?



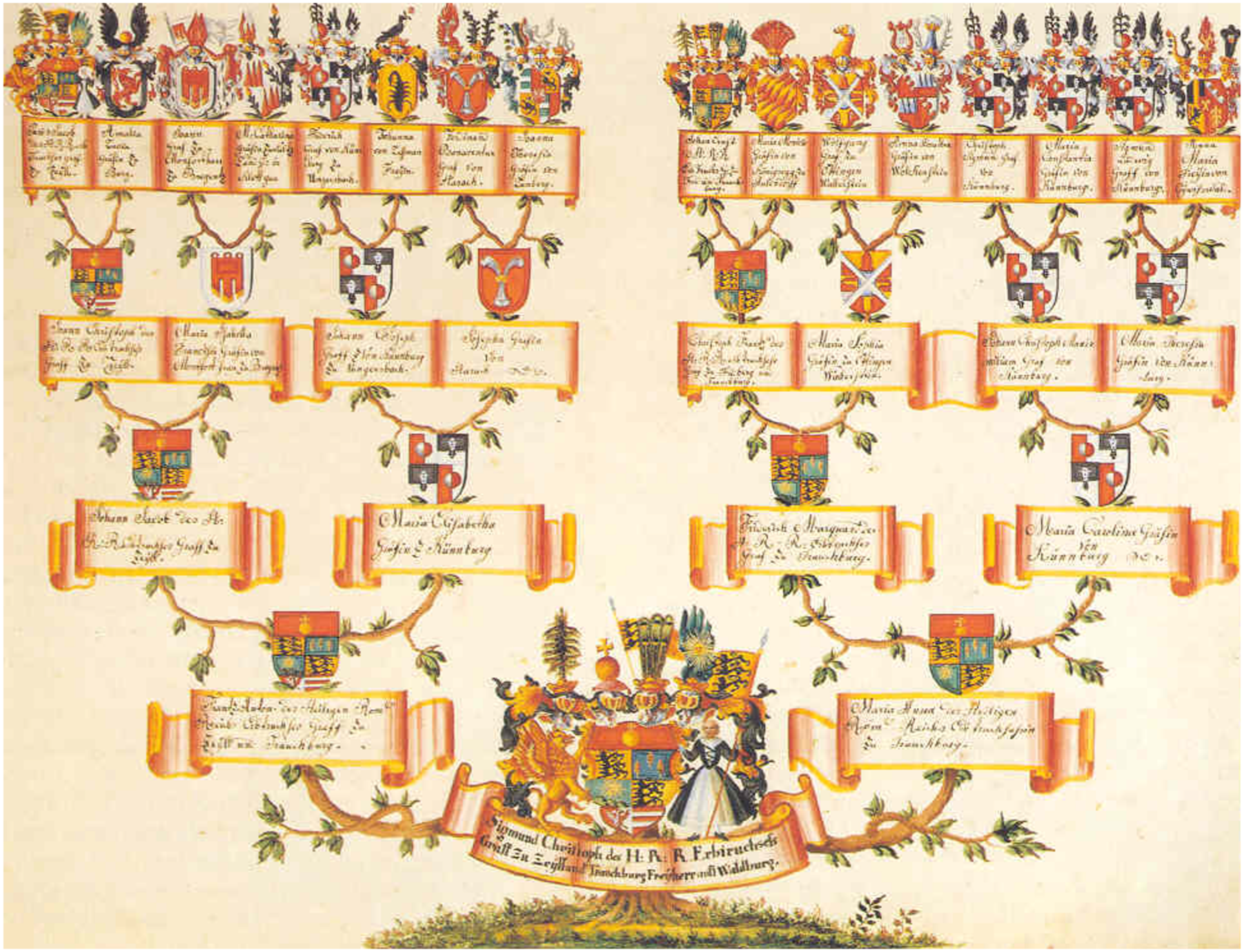
1. `Leaf [A;T]`
2. `Node (Leaf [G], [A;T], Leaf [A])`
3. `Node (Leaf [A], [A;T], Leaf [G])`
4. `Node (Leaf [T], [A;T],`
   `Node (Leaf [G;C], [G], Leaf []))`
5. None of the above

Answer: 3

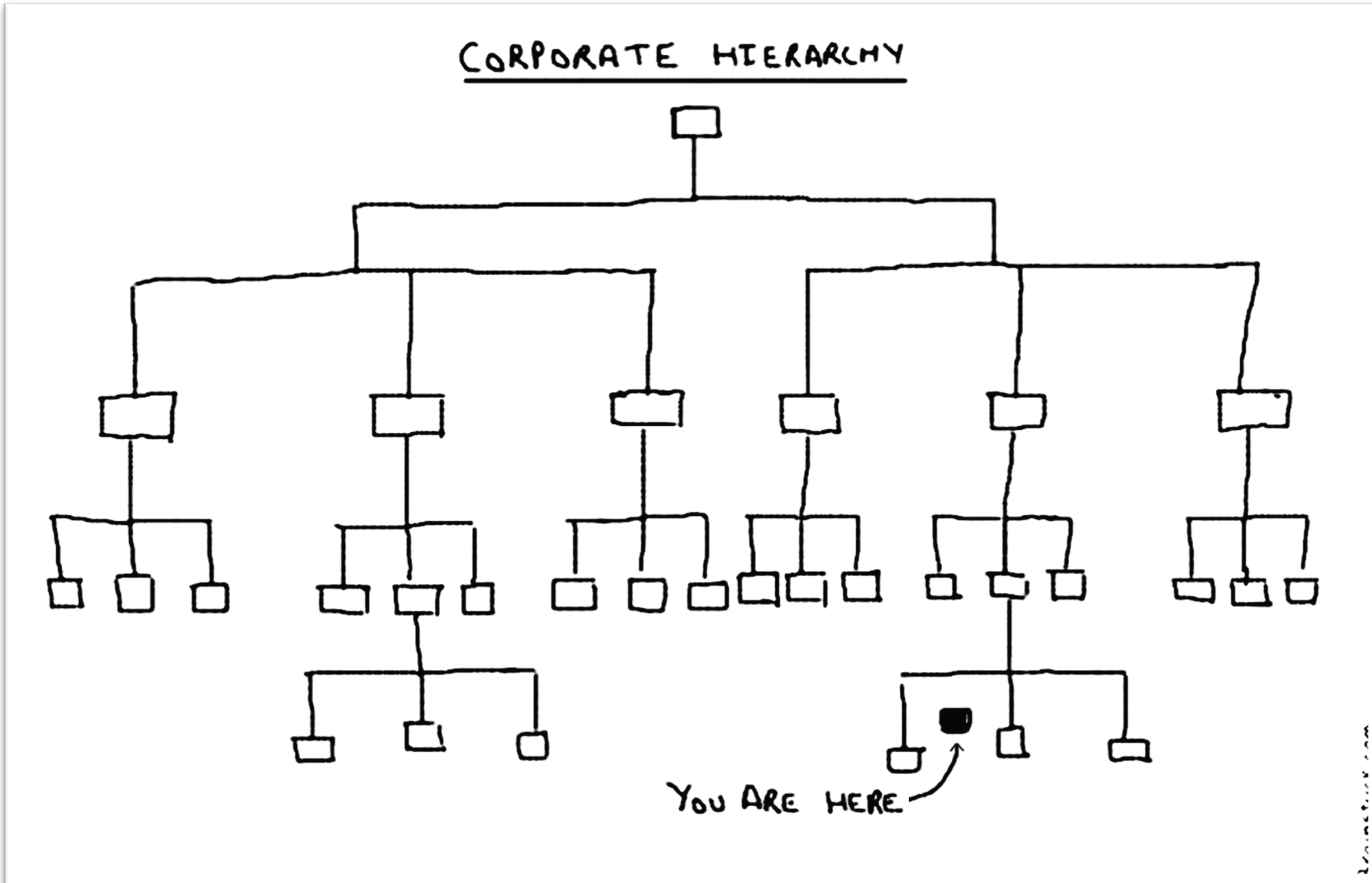
Trees are everywhere



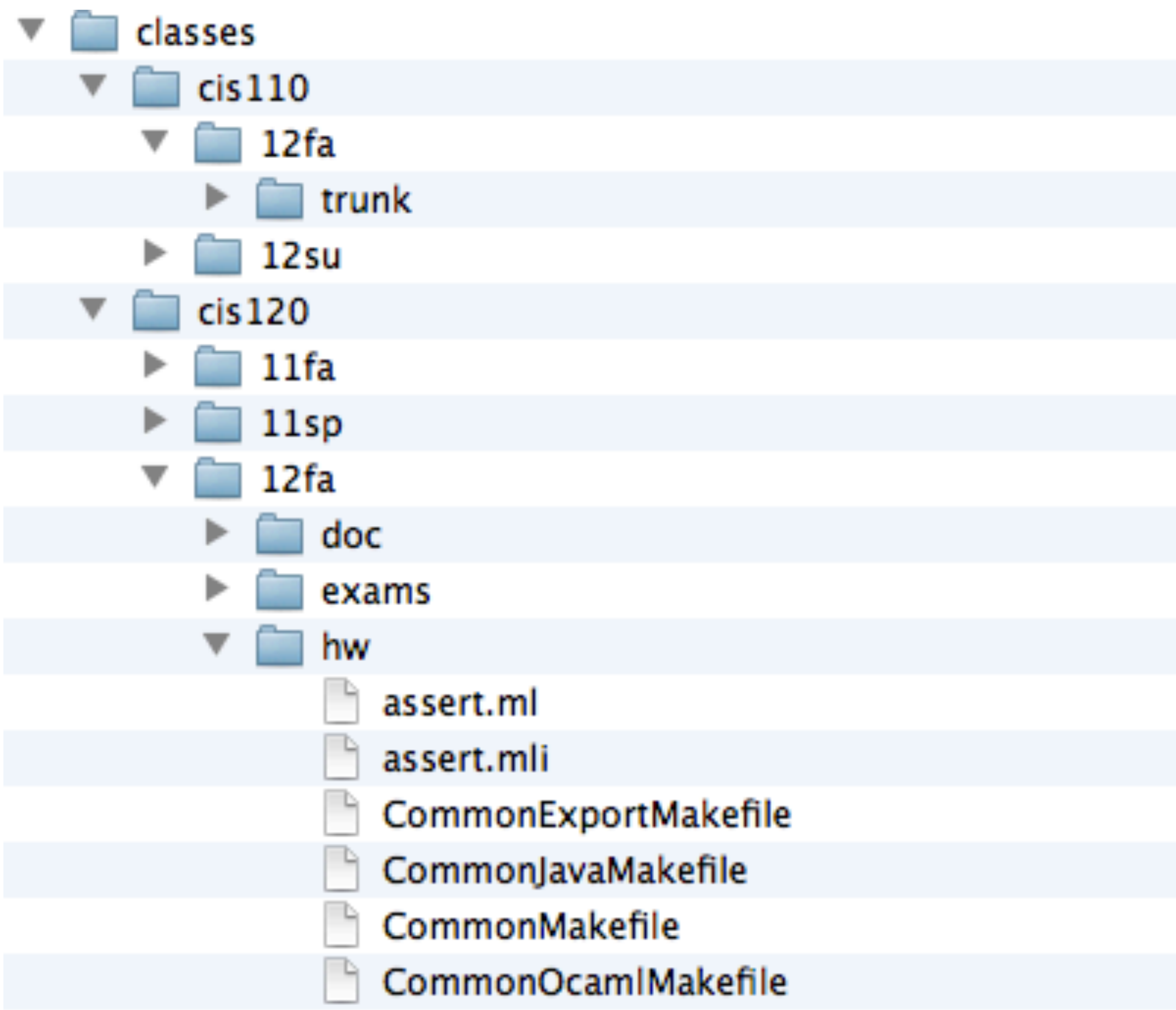
Family trees



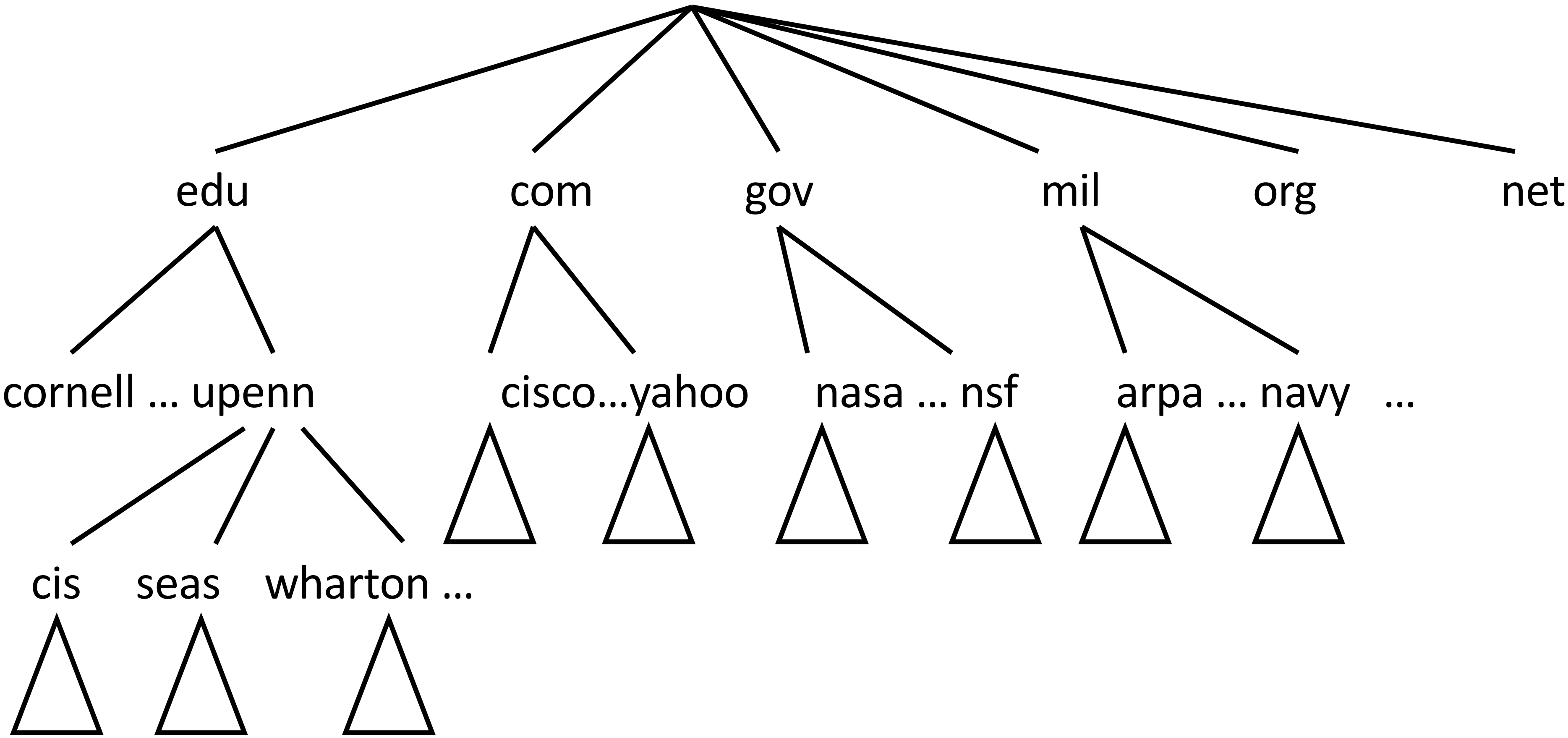
Organizational charts



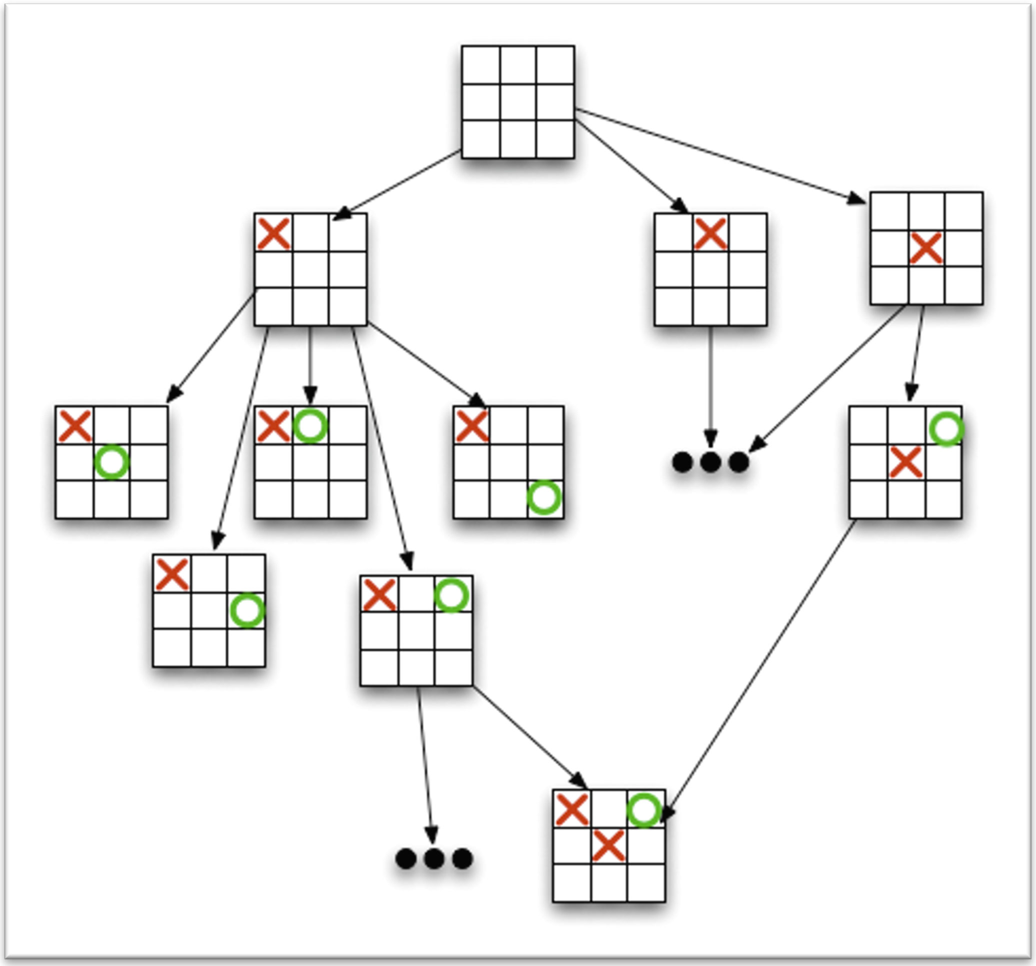
Filesystem Directory Structure



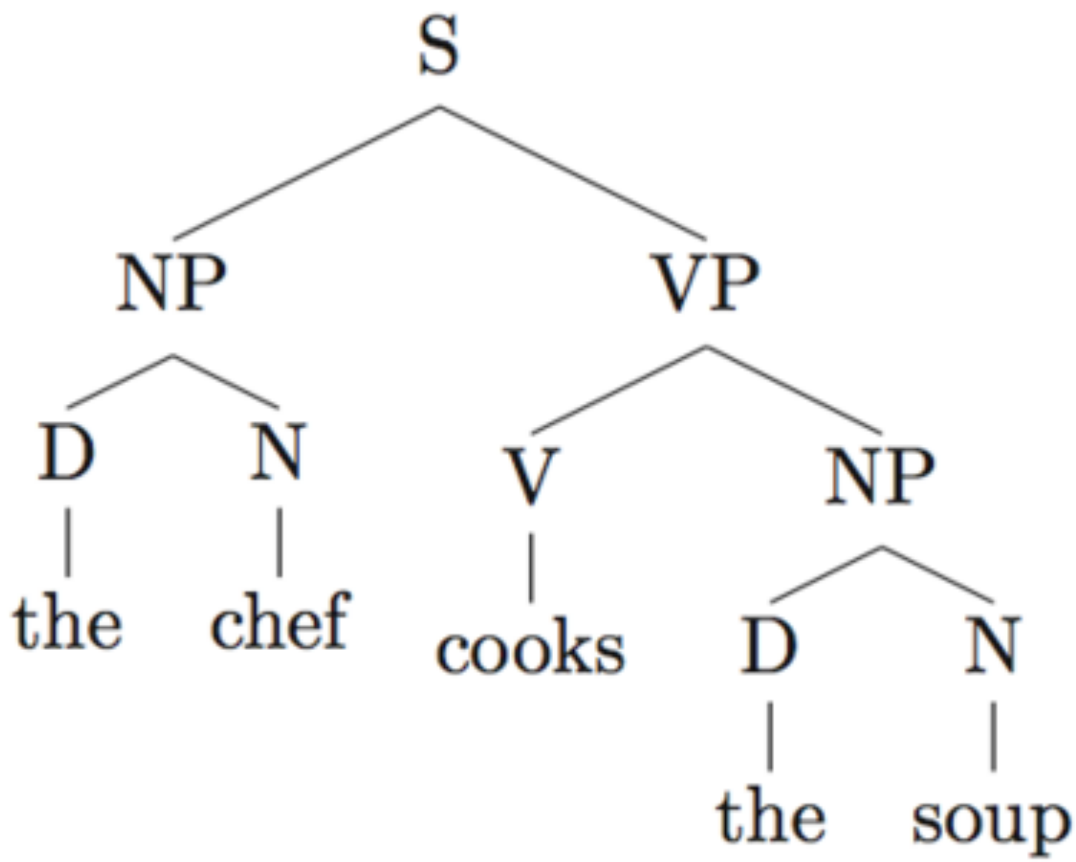
Domain Name Hierarchy



Game trees



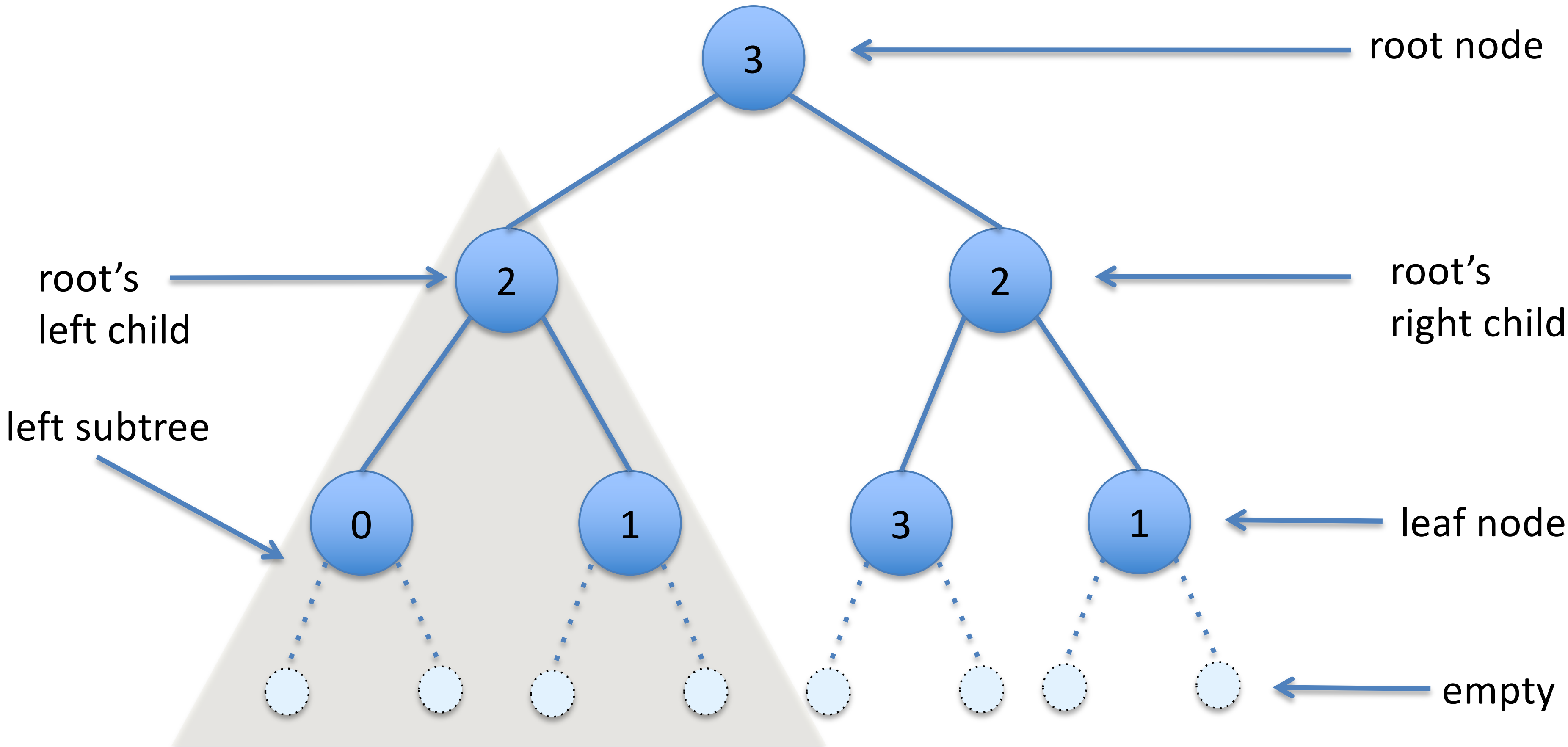
Natural-Language Parse Trees



Binary Trees

A particular form of tree-structured data

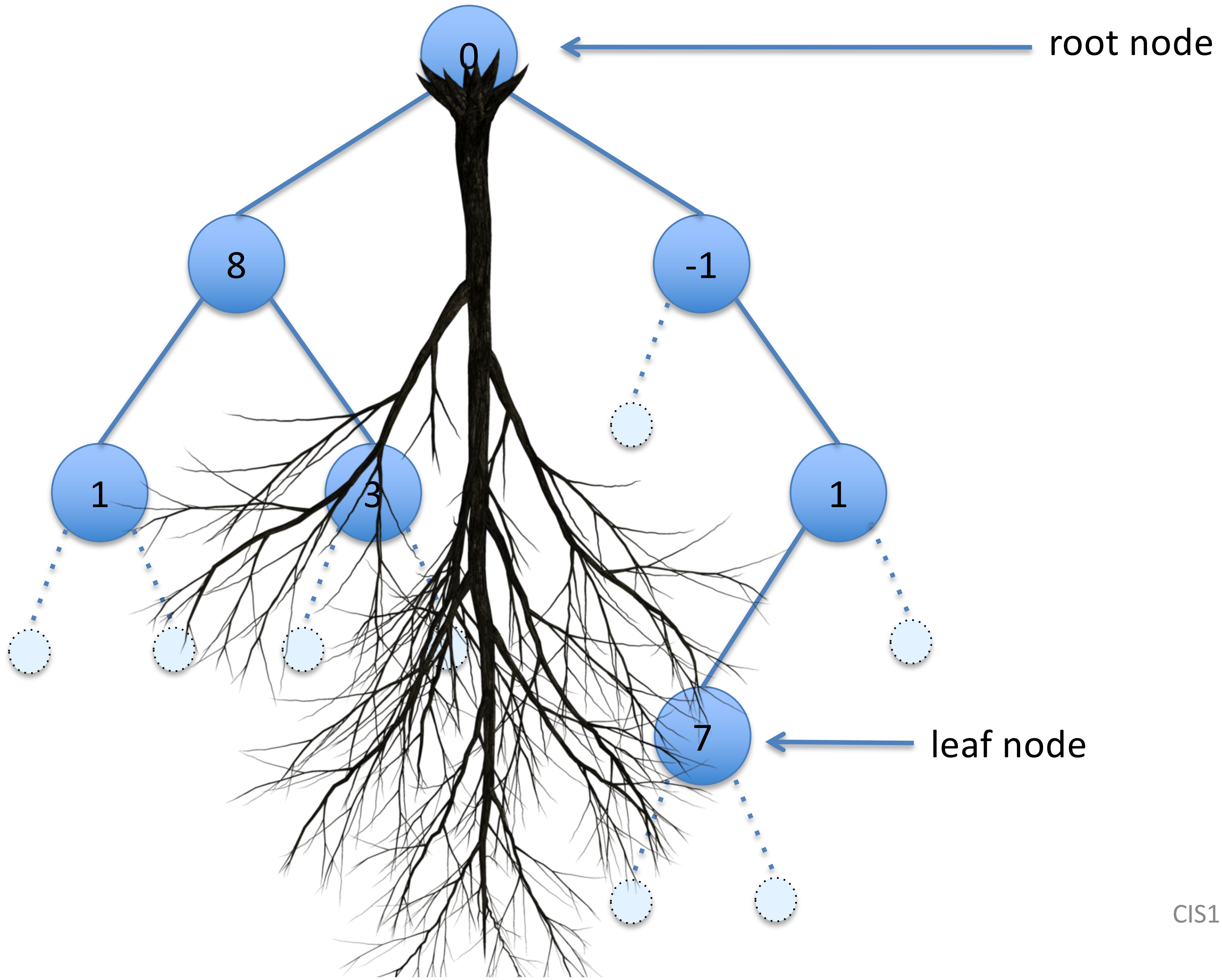
Binary Trees



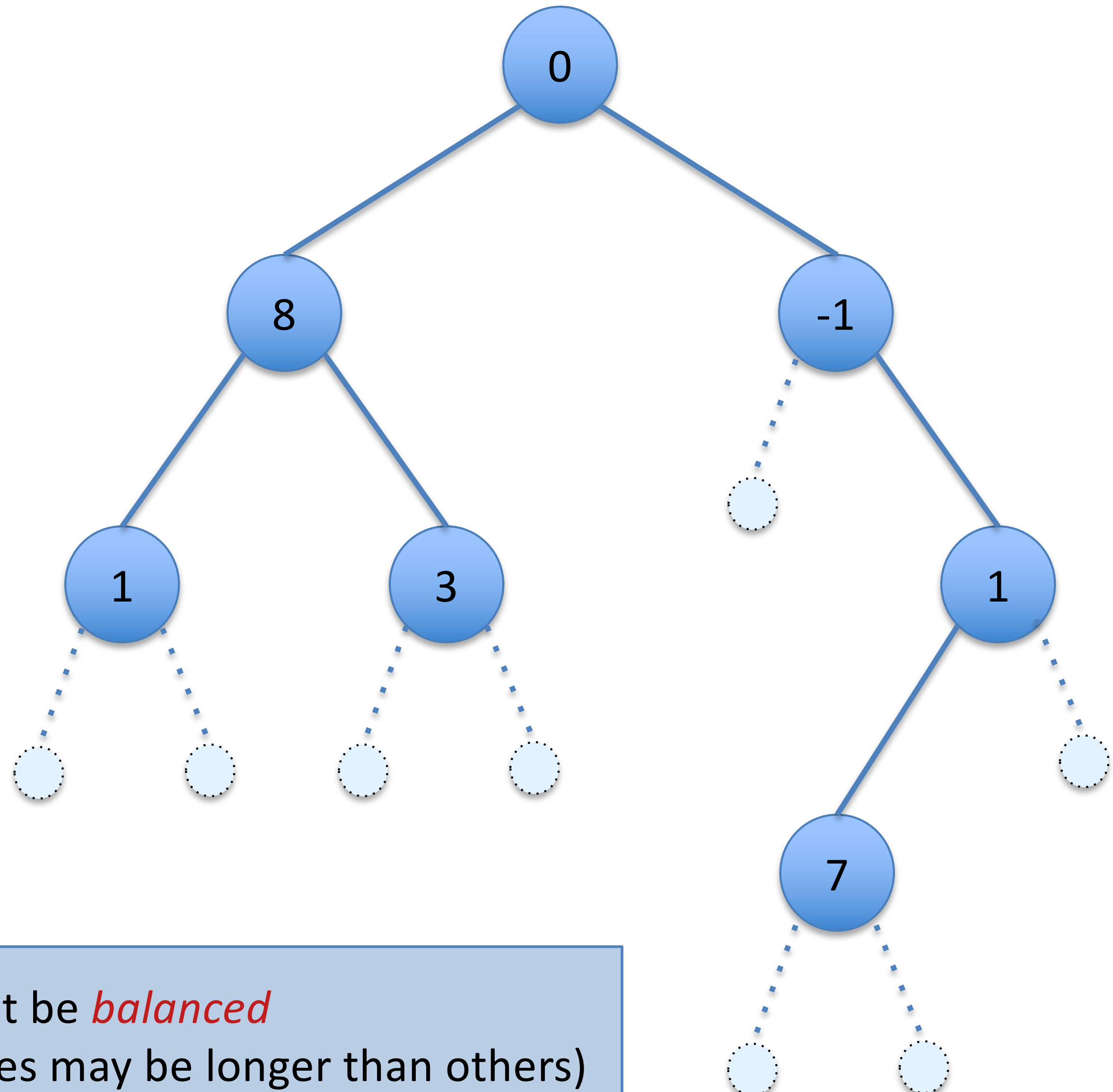
A binary tree is either *empty*, or a *node* with at most two children, both of which are also binary trees.

A *leaf* is a node whose children are both empty.

Trees are Drawn Upside Down



Another Example Tree



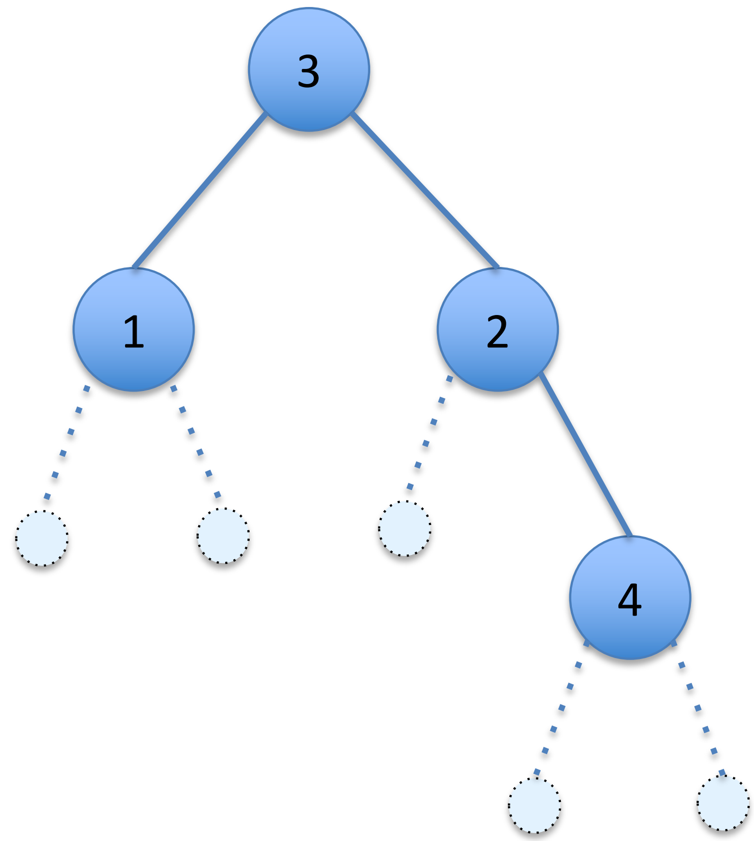
Trees need not be *balanced*
(some branches may be longer than others)

Binary Trees in OCaml

```
type tree =
| Empty
| Node of tree * int * tree
```

```
let t : tree =
  Node (Node (Empty, 1, Empty),
    3,
    Node (Empty, 2,
      Node (Empty, 4, Empty)))
```

=



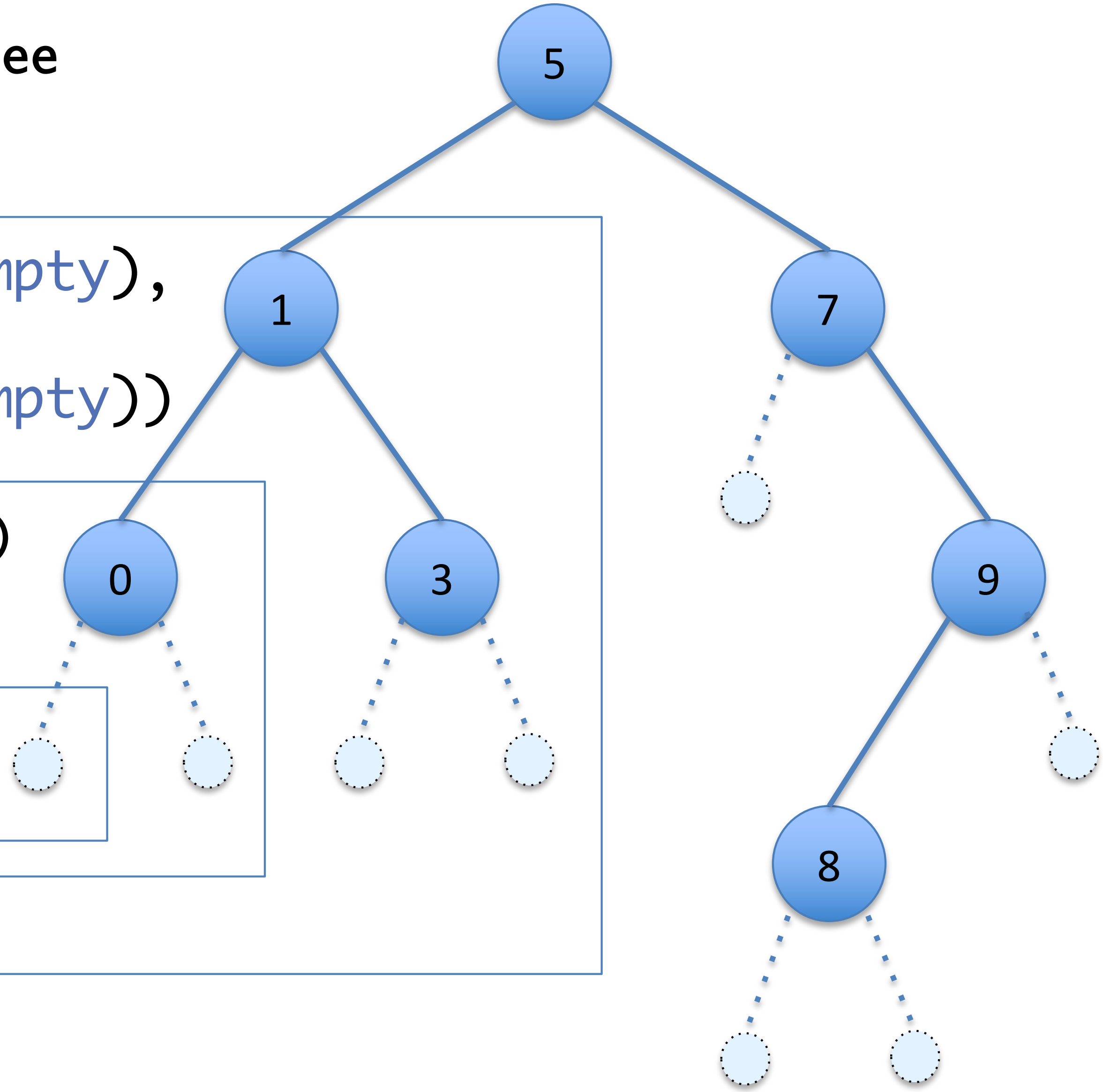
Representing trees

```
type tree =
| Empty
| Node of tree * int * tree
```

```
Node (Node (Empty, 0, Empty),
      1,
      Node (Empty, 3, Empty))
```

```
Node (Empty, 0, Empty)
```

```
Empty
```



Coding with binary trees

see tree.ml

treeExamples.ml

Structural Recursion Over *Trees*

Structural recursion builds an answer from smaller components:

```
let rec f (t : tree) … : … =
  begin match t with
  | Empty -> …
  | Node(l,x,r) -> … (f l) … x … (f r) …
  end
```

The branch for `Empty` calculates the value (`f Empty`) directly.

- this is the *base case* of the recursion

The branch for `Node(l,x,r)` calculates

`(f(Node(l,x,r))` given `x` and `(f l)` and `(f r)`.

- this is the *inductive case* of the recursion

Tree vs. List Recursion

```
let rec f (t : tree) … : … =
  begin match t with
  | Empty -> …
  | Node(l,x,r) -> … (f l) … (f r) …
  end
```

Two recursive calls, for left and right sub trees, versus one for lists.

```
let rec f (l : … list) … : … =
  begin match l with
  | [] -> …
  | ( hd :: rest ) -> … f rest …
  end
```

Trees as Containers

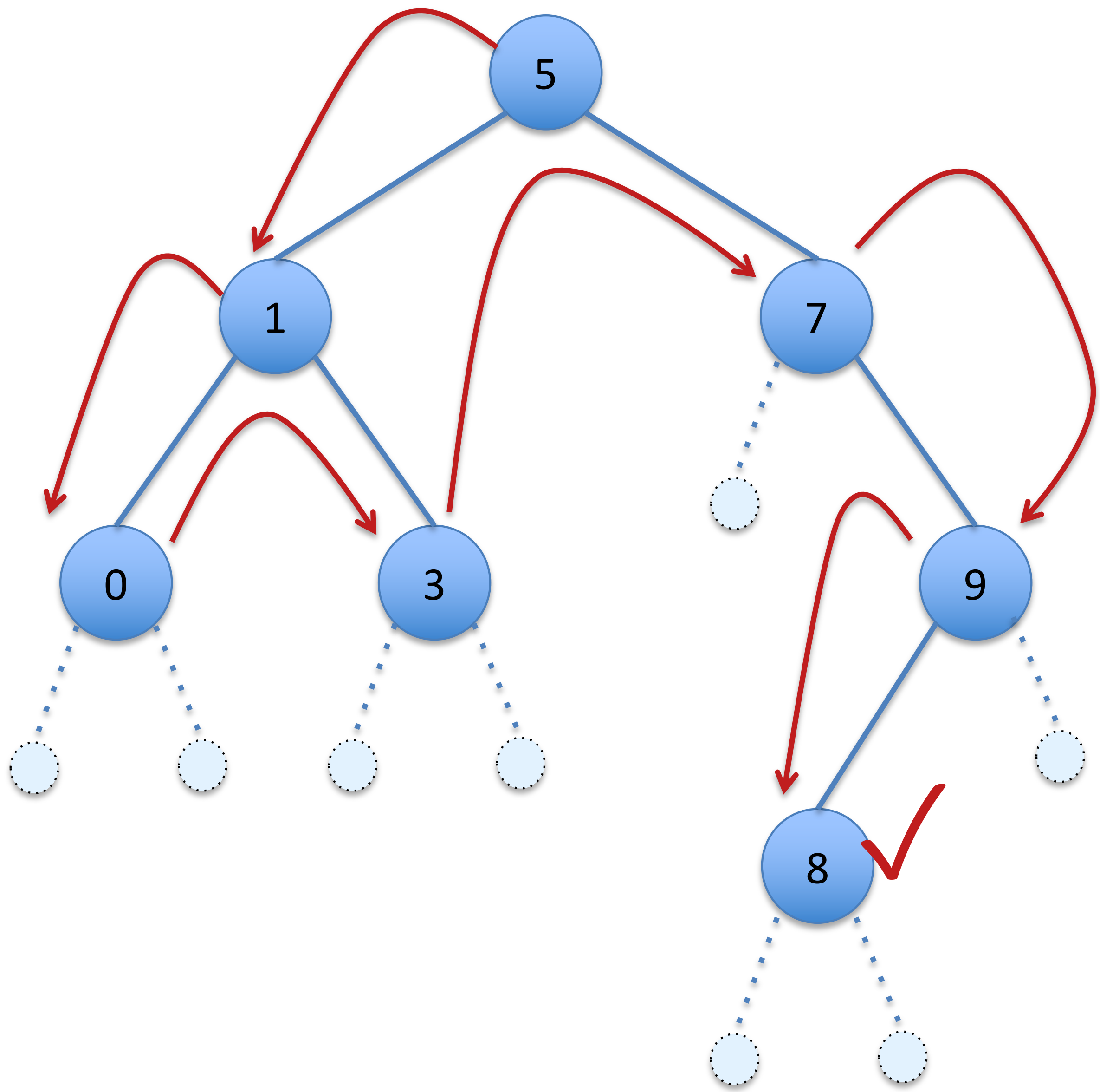
- Like lists, trees aggregate ordered data
- As we did for lists, we can write a function to determine whether a tree *contains* a particular element

Searching for Data in a Tree

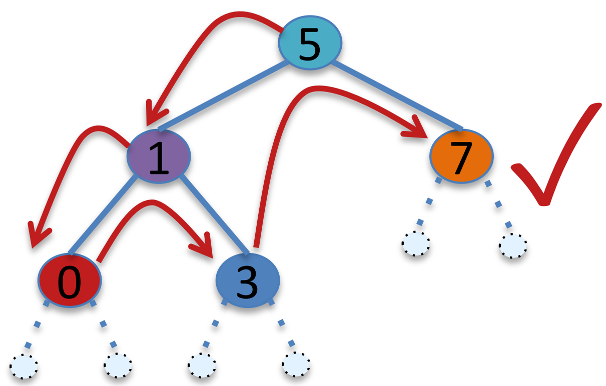
```
let rec contains (t:tree) (n:int) : bool =
  begin match t with
  | Empty -> false
  | Node(lt,x,rt) ->
        x = n
     || contains lt n
     || contains rt n
  end
```

- This function searches through the tree, looking for n
- In the worst case, it might have to traverse the *entire tree*

Search during (contains t 8)



Searching for Data in a Tree




```
let rec contains (t:tree) (n:int) : bool =
  begin match t with
  | Empty -> false
  | Node(lt,x,rt) -> x = n ||
                (contains lt n) || (contains rt n)
  end
```

```
contains (Node(Node(Node (Empty, 0, Empty), 1, Node(Empty, 3, Empty)),
                  5, Node (Empty, 7, Empty))) 7
```

```
5 = 7
|| contains (Node(Node (Empty, 0, Empty), 1, Node(Empty, 3, Empty))) 7
|| contains (Node (Empty, 7, Empty)) 7
```

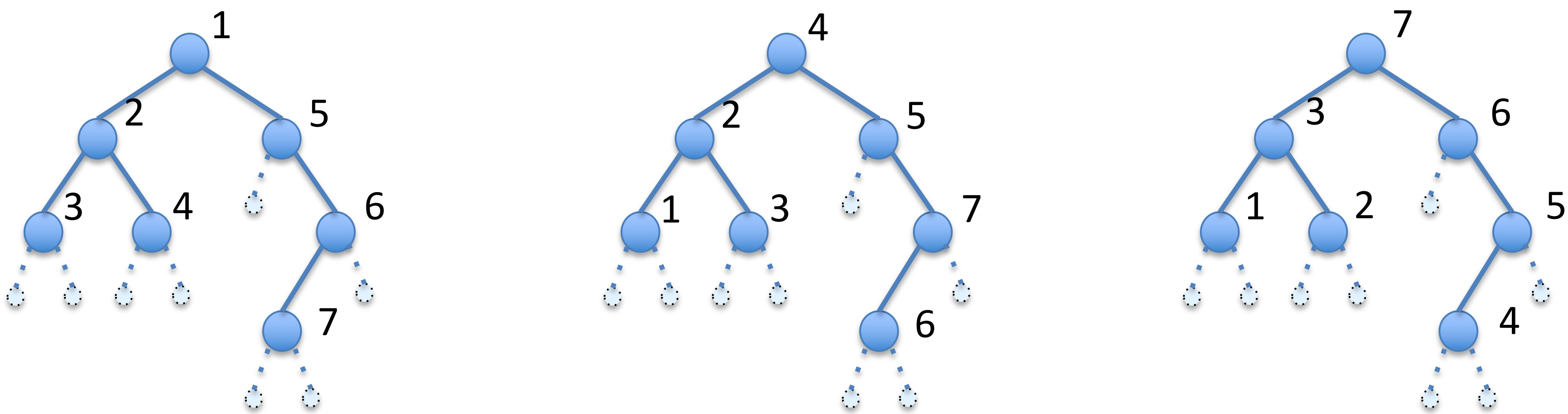
```
(1 = 7 || contains (Node (Empty, 0, Empty))  7
        || contains (Node(Empty, 3, Empty)) 7)
|| contains (Node (Empty, 7, Empty)) 7
```

```
((0 = 7 || contains Empty 7 || contains Empty 7)
         || contains (Node(Empty, 3, Empty)) 7)
|| contains (Node (Empty, 7, Empty)) 7
```

```
contains (Node(Empty, 3, Empty)) 7
|| contains (Node (Empty, 7, Empty)) 7
```

```
contains (Node (Empty, 7, Empty)) 7
```

Recursive Tree Traversals



Pre-Order
Root – Left – Right

In Order
Left – Root – Right

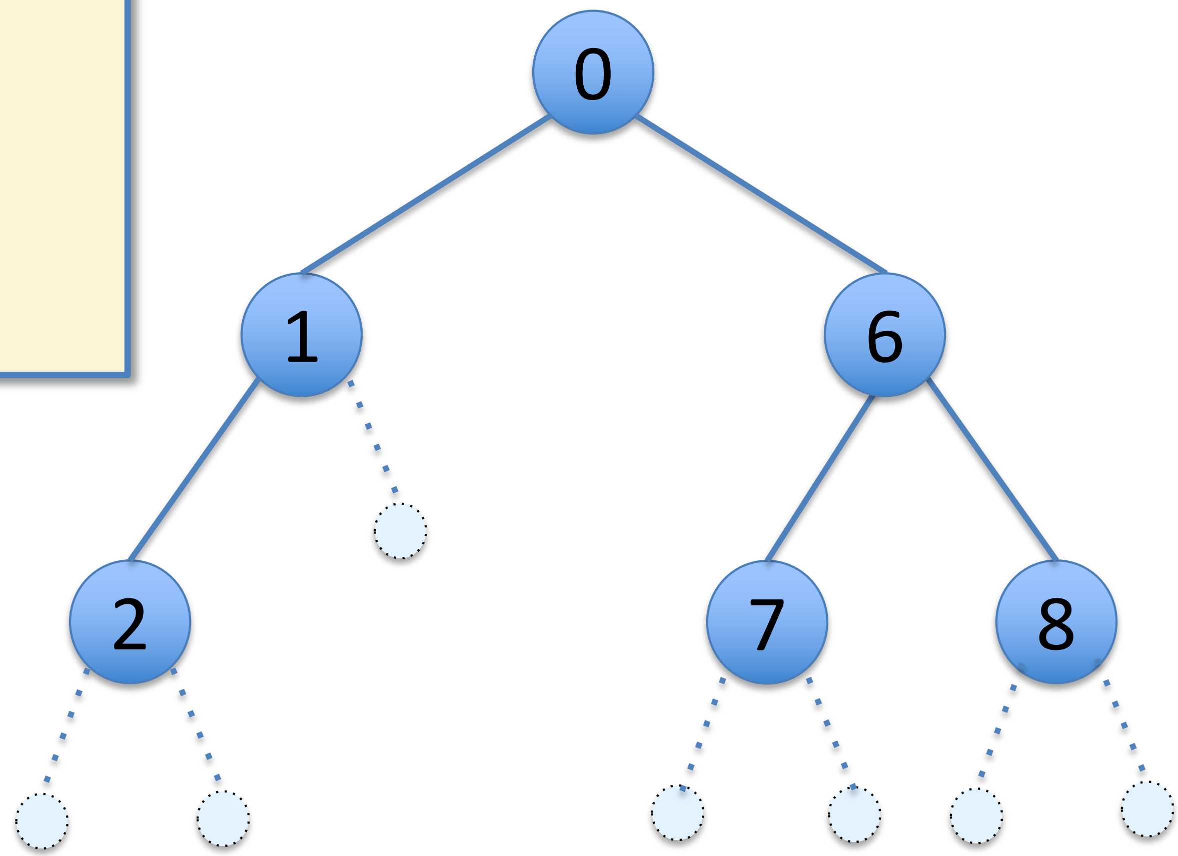
Post-Order
Left – Right – Root

```
(* Code for Pre-Order Traversal *)
let rec f (t:tree) : … =
  begin match t with
  | Empty -> …
  | Node(l, x, r) ->
    let root = … x … in (* process root *)
    let left =  f l  in (* recursively process left subtree *)
    let right = f r  in (* recursively process right subtree *)
    combine root left right
  end
```

Other traversals vary the order in which these are computed…

In what sequence will the nodes of this tree be visited by a post-order traversal?

1. [0;1;6;2;7;8]
2. [0;1;2;6;7;8]
3. [2;1;0;7;6;8]
4. [7;8;6;2;1;0]
5. [2;1;7;8;6;0]



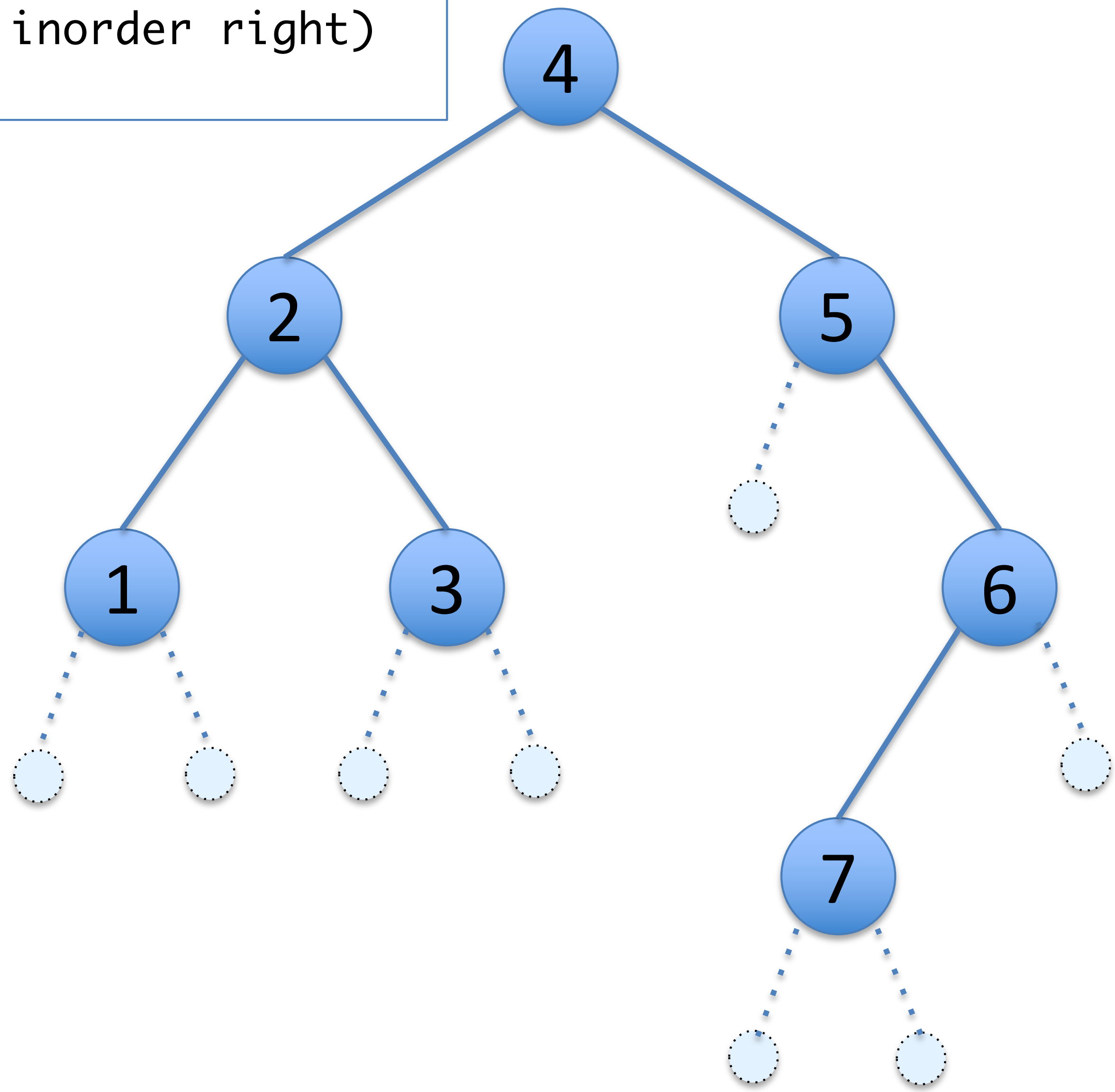
Post-Order
Left – Right – Root

Answer: 5

```
let rec inorder (t:tree) : int list =
  begin match t with
    | Empty -> []
    | Node (left, x, right) ->
        inorder left @ (x :: inorder right)
end
```

What is the result of applying this function on this tree?

1. []
2. [1;2;3;4;5;6;7]
3. [1;2;3;4;5;7;6]
4. [4;2;1;3;5;6;7]
5. [4]
6. [1;1;1;1;1;1;1]
7. none of the above



Answer: 3