Programming Languages and Techniques (CIS1200)

Lecture 5

Datatypes and Trees

CIS 1200 Announcements

- HW01 is due TOMORROW at midnight
  - Mandatory in-person check-in (15-20 minutes) with your recitation TAs after you submit
  - look for them to coordinate

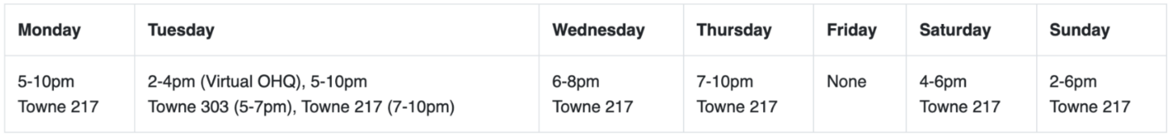
- Dr Weirich's Office Hours: Tuesdays 2:30-3:30 in Levine 510, or by appointment

- Attendance grading (via PollEverywhere) starts today

How to get help at Office Hours

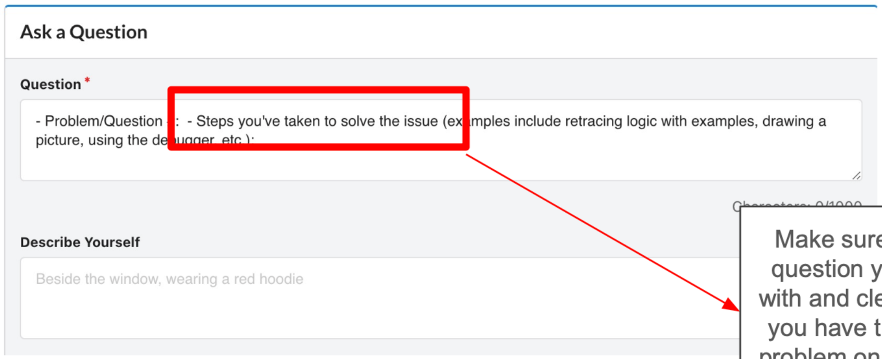
1. Go to office hours location (schedule available on course website)

TA Office Hours Schedule



Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
5-10pm Towne 217	2-4pm (Virtual OHQ), 5-10pm Towne 303 (5-7pm), Towne 217 (7-10pm)	6-8pm Towne 217	7-10pm Towne 217	None	4-6pm Towne 217	2-6pm Towne 217

2. Enter the office hours queue via OHQ
   a. Can be accessed via the course website, you may need to click "add course" and search "CIS 1200"
3. Fill out the template



Ask a Question

Question *

- Problem/Question : - Steps you've taken to solve the issue (examples include retracing logic with examples, drawing a picture, using the debugger, etc.):

Characters: 0/1000

Describe Yourself

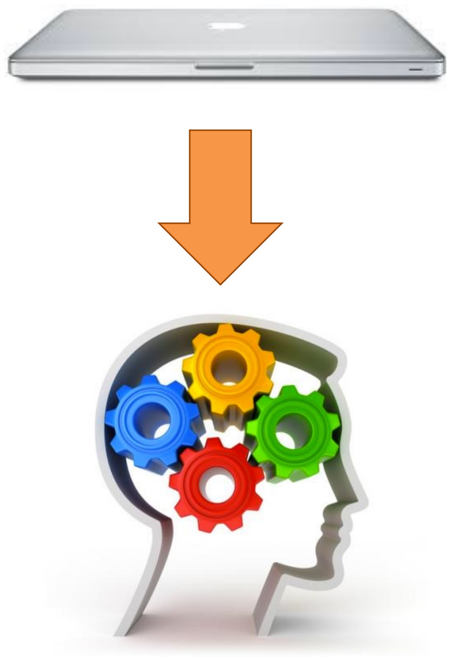
Beside the window, wearing a red hoodie

Make sure to mark which question you're struggling with and clearly explain how you have tried to solve the problem on your own before you will be helped!

4. Wait to be seen

Reminder: No Laptops during Lecture

- Laptops *closed*… minds *open*
  - Although this is a computer science class, the use of electronic devices – laptops, phones, etc., during lecture (*except for participating in quizzes*) is *prohibited*
- Why?
  - Device users tend to surf/chat/email/ game/text/tweet/etc.
  - They also distract those around them
  - More effective to take notes *by hand*
  - You'll get plenty of time in front of your computer while working on the homework :-)



Recap: Lists, Recursion, & Tuples

A General Pattern:
Structural Recursion Over Lists

Structural recursion builds an answer from smaller components:

```
let rec f (l : … list) … : … =
  begin match l with
  | [] -> …               (* BASE CASE *)

  | ( hd :: rest ) ->     (* INDUCTIVE CASE *)
        … (f rest) …
  end
```

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

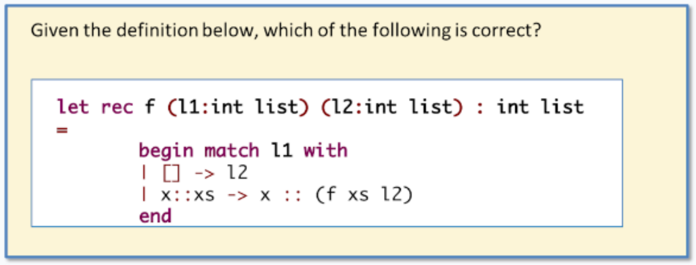
- the *base case* of the recursion

The branch for `hd::rest` calculates `f (hd::rest)` given `hd` and `(f rest)`.

- the *inductive case* of the recursion

5: Given the definition below, which of the following is correct?

0



Given the definition below, which of the following is correct?

```
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]) = [3;4;1;2]   0%

(f [1;2] [3;4]) = [1;2;3;4]   0%

(f [1;2] [3;4]) = [4;3;2;1]   0%

(f [1;2] [3;4]) = [1;3;2;4]   0%

none of the above   0%

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]
```

5: What is the type of this expression?

✓ 0

What is the type of this expression?

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

int	0%
int list	0%
int list list	0%
(int * int list) list	0%
int * (int list) * (int list list)	0%
(int * int * int) list	0%
none (expression is ill typed)	0%

What is the type of this expression?

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

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

Answer: 4

5: What is the type of this expression?

✓ 0

What is the type of this expression?

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

int * bool

0%

int list * bool list

0%

(int * bool) list

0%

(int * bool) list list

0%

none (expression is ill typed)

0%

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

Topics for Today

Defining your own types

Types for Structured Data

- Like most programming languages, OCaml offers a variety of ways of creating and manipulating *structured* data

- We have already seen
  - *primitive types* (`int`, `string`, `bool`, …)
  - *lists* (`int list`, `string list`, `string list list`, …)
  - *tuples* (`int * int`, `int * string`, …)

- Today
  - *type abbreviations*
  - *user-defined* datatypes

Type Abbreviations

A Handy Feature: Type Abbreviations

OCaml lets us define a new *name* for an existing type

```
type cents = int
type dollars = int
type money = dollars * cents
```

type name

definition in terms of existing types

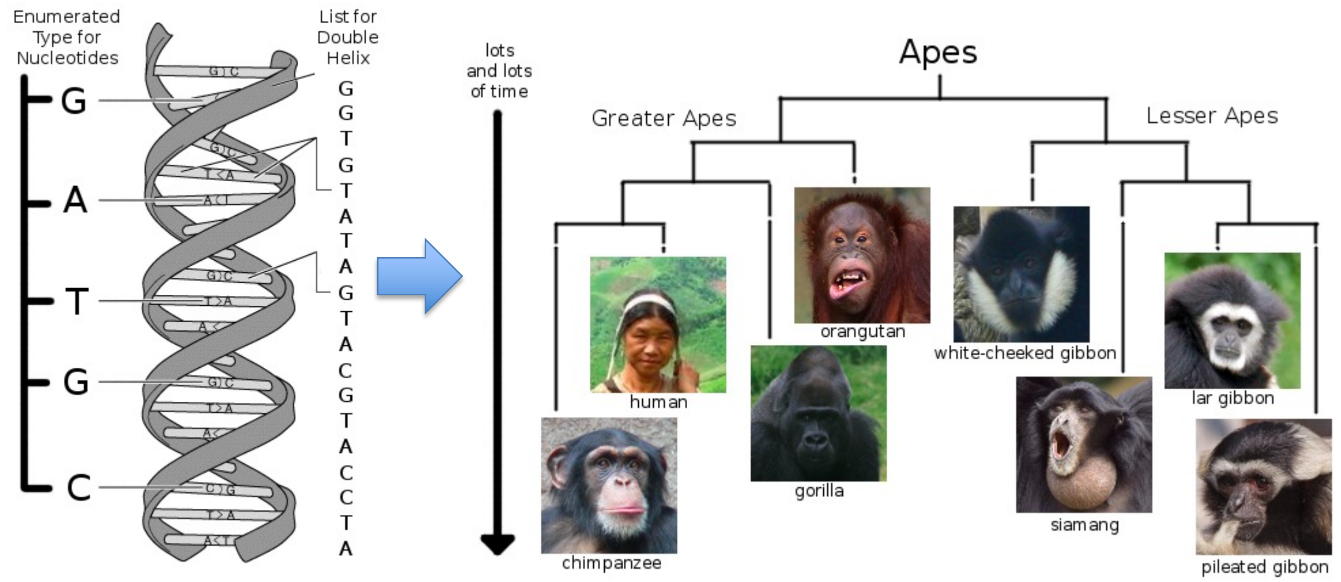
- A type abbreviation is **interchangeable** with its definition
- Abbreviations are useful for **documenting** important concepts

```
let profit (attendees:int) : money = …
```

Datatypes and Trees

HW 2 Case Study: Evolutionary Trees

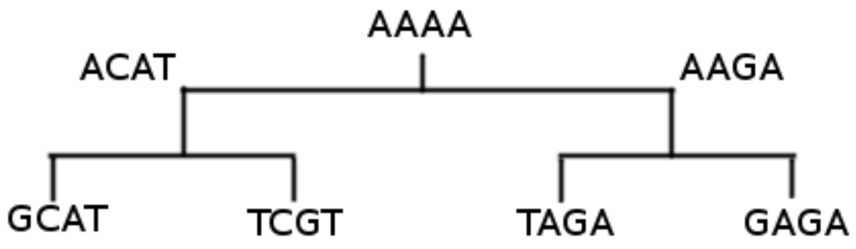
- Problem: reconstruct evolutionary trees* from DNA data.
  - What are the relevant abstractions?
  - How can we use the language features to define them?
  - How do the abstractions help shape the program?



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

DNA Computing Abstractions

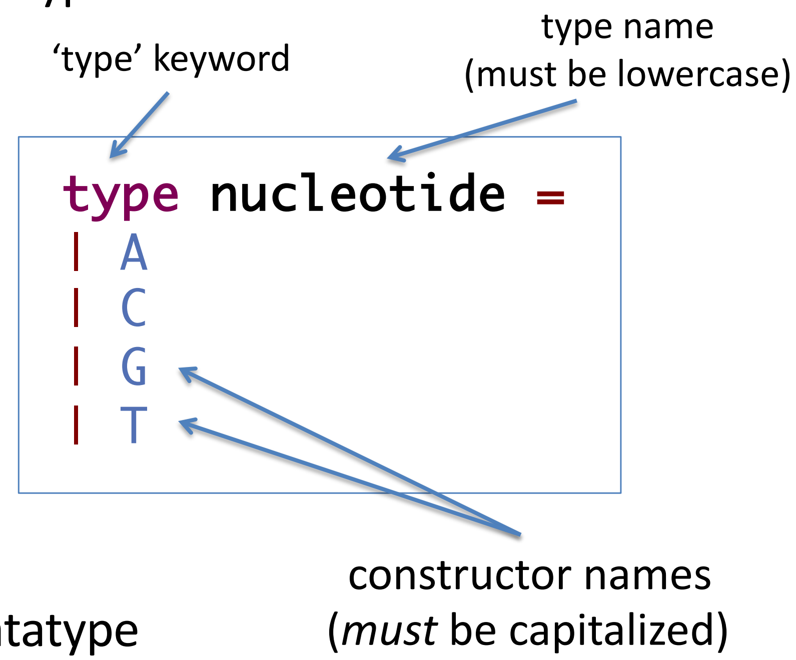
- Nucleotide
  - Adenine (A), Guanine (G), Thymine (T), or Cytosine (C)
- Helix
  - a sequence of nucleotides: e.g. AGTCCGATTACAGAGA…
  - genetic code for a particular species (human, gorilla, etc)
- Phylogenetic tree
  - Binary tree with a helix (i.e. species) at all leaves and internal nodes



Simple User-Defined Datatypes

OCaml lets programmers define *new* datatypes

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



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

‘type’ keyword
type name (must be lowercase)
constructor names (*must* be capitalized)

The *constructors* are the values of the datatype

```
Sunday : day
A : nucleotide
```

Pattern Matching on 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, warning if you miss or repeat a case
- Like lists, the pattern syntax follows the datatype values (*i.e.*, the patterns for nucleotides are the constructors `A`, `C`, `G`, and `T`)

A Point About *Abstraction*

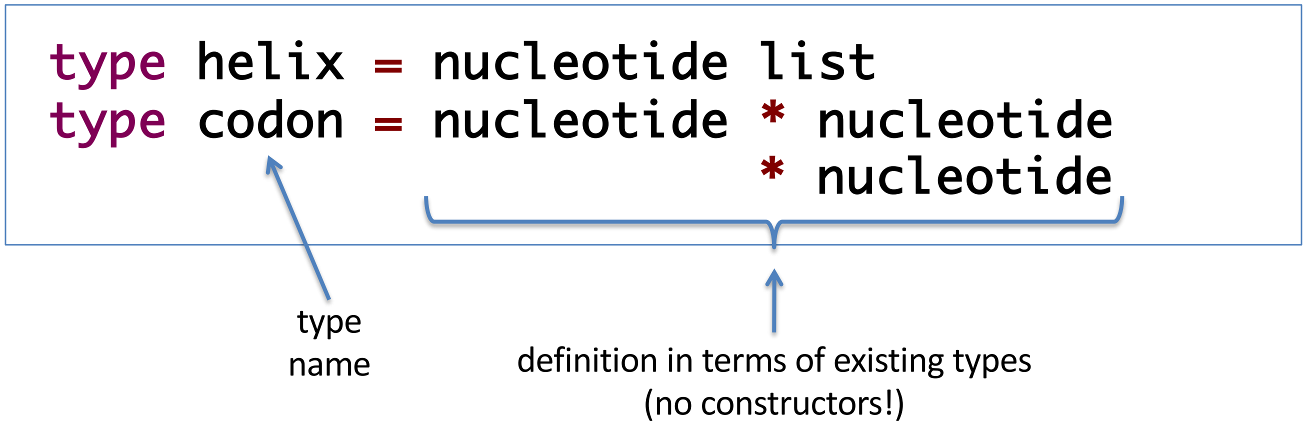
- We *could* represent weekdays by using integers:
  - `Sunday = 0`, `Monday = 1`, `Tuesday = 2`, *etc.*
- But…!
  - Integers support different operations than days do:

    ```
    Wednesday - Monday   = Tuesday     (?!?)
    Wednesday * Tuesday = Saturday    (?!?)
    ```
  - There are *more* integers than days (What day is 17? -3?)
- Confusing integers with days can lead to bugs
  - Many "scripting" languages (PHP, Javascript, Perl, Python,…) do confuse values of different types (true == 1 == "1"), leading to much misery when debugging…
- For these reasons, most modern languages (Java, C#, C++, Rust, Swift,…) provide *user-defined types*

Type Abbreviations II

Abbreviations let us give *names* to complex types but do not introduce new abstractions

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



- *i.e.*, a `helix` is the **same** as a list of `nucleotides`

  ```
  let x : helix = [A;C;C] in length x
  ```

- Can make code easier to read & write, but does not provide all the benefits of user-defined types

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 on 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
```

- Defining a datatype also defines its patterns
- Datatype patterns *bind* identifiers (*e.g.*, ‘n’) just like for lists and tuples

5: What is the type of this expression?

✓ 0

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

What is the type of this expression?

```
(A, "A")
```

nucleotide	0%
nucleotide list	0%
helix	0%
nucleotide * string	0%
string * string	0%
none (expression is ill typed)	0%

```
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

5. What is the type of the expression [A;C]?

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

nucleotide	0%
helix	0%
nucleotide list	0%
string * string	0%
nucleotide * nucleotide	0%
none (expression is ill typed)	0%

```
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)*

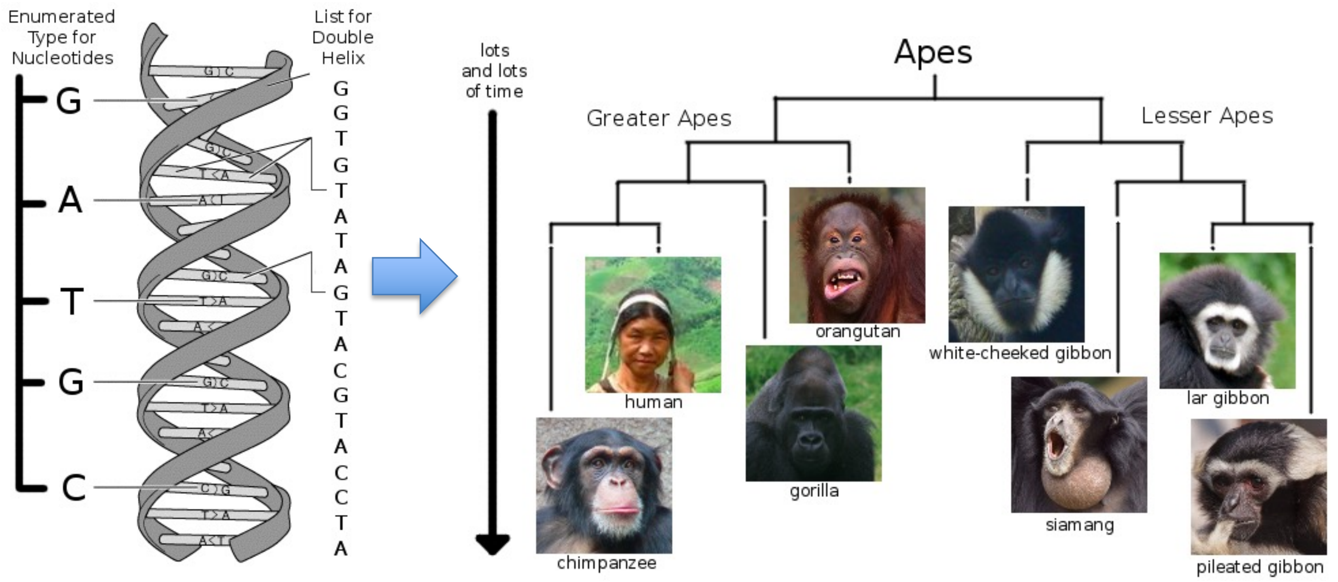
Answer: both 2 and 3

Defining a *datatype* adds a fresh type as a first-class concept to your program.

- *Constructors* explain the shape/structure of the values.
- *Patterns* explain how to inspect/name the components of those values.
- *Abstraction* means that the new type can't be confused with other, existing types.

Trees

- We now know how to define types for nucleotides, codons, DNA helices, *etc.*
- What about the evolutionary tree itself?



Recursive User-defined Datatypes

Datatype definitions can mention themselves *recursively*:

base constructor

```
type labeled_tree =
 | LLeaf of helix
 | LNode of labeled_tree * helix * labeled_tree
```

recursive constructor

LNode carries a tuple of values

recursive occurrences of datatype being defined

Tree Values

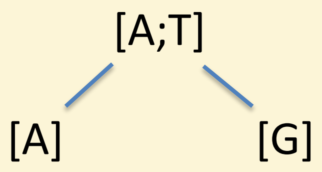
```
type labeled_tree =
 | LLeaf of helix
 | LNode of labeled_tree * helix * labeled_tree
```

Example values of type `labeled_tree`:

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

Constructors of `labeled_tree`

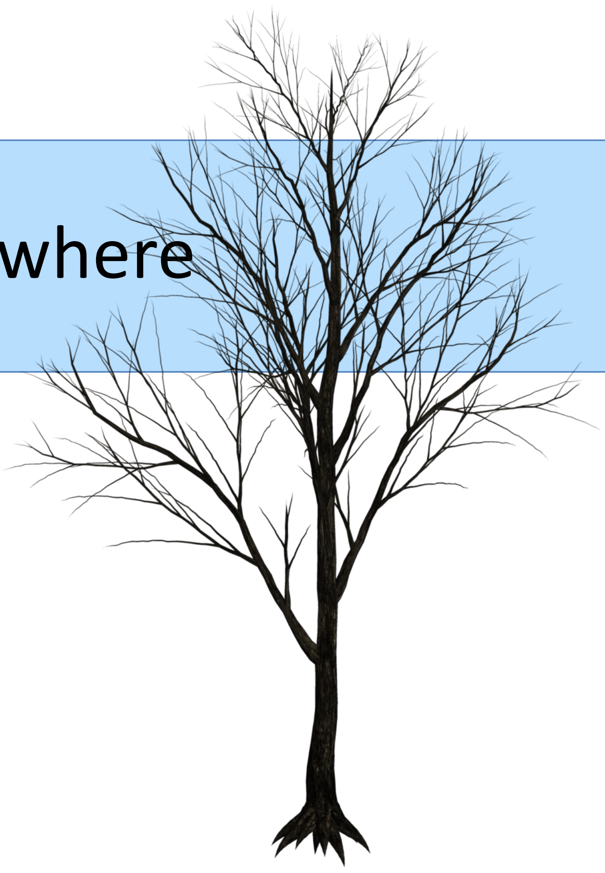
How would you construct this tree in OCaml?



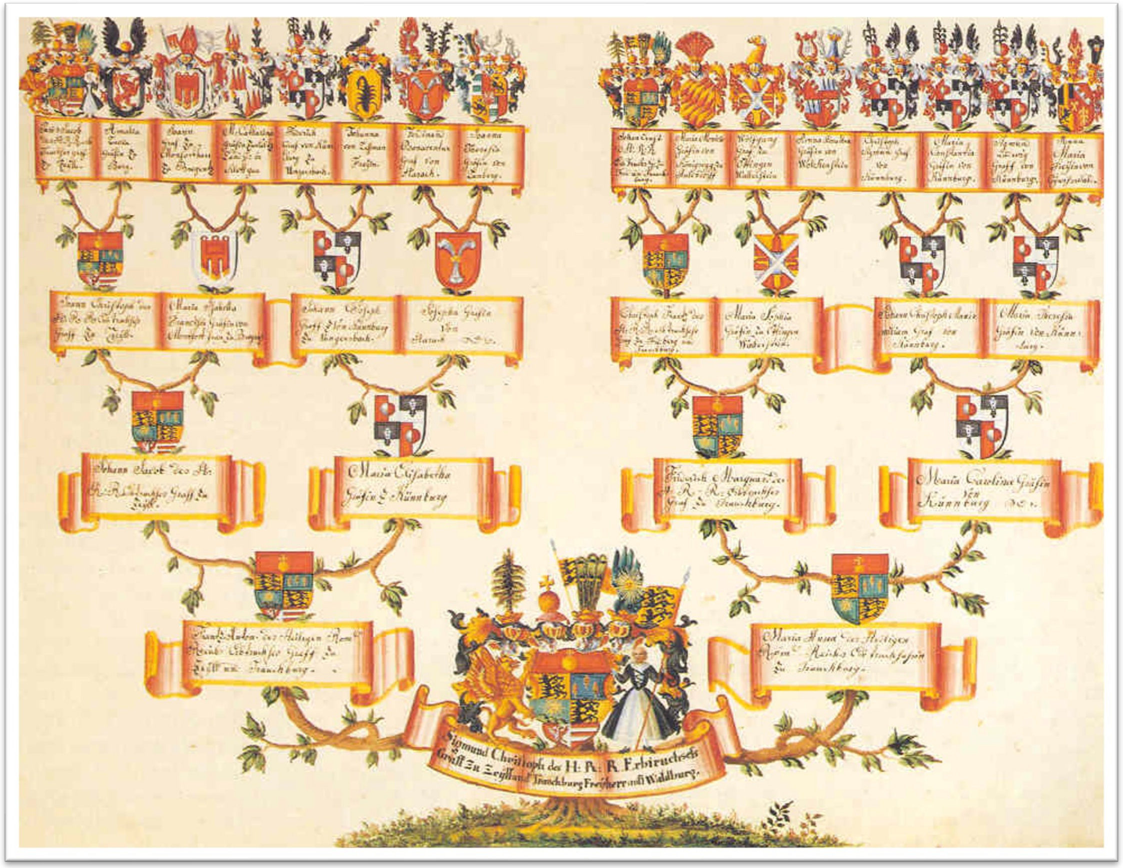
```
1. LLeaf [A;T]
2. LNode (LLeaf [G], [A;T], LLeaf [A])
3. LNode (LLeaf [A], [A;T], LLeaf [G])
4. LNode (LLeaf [T], [A;T],
      LNode (LLeaf [G;C], [G], LLeaf []))
5. None of the above
```

Answer: 3

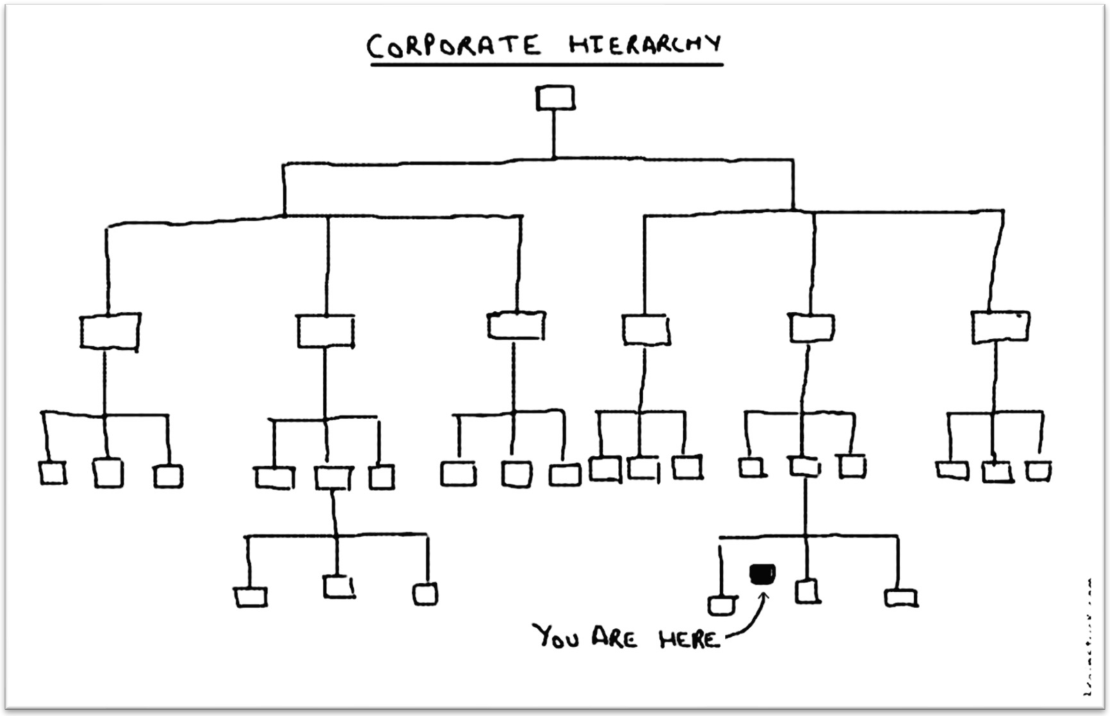
Trees are everywhere



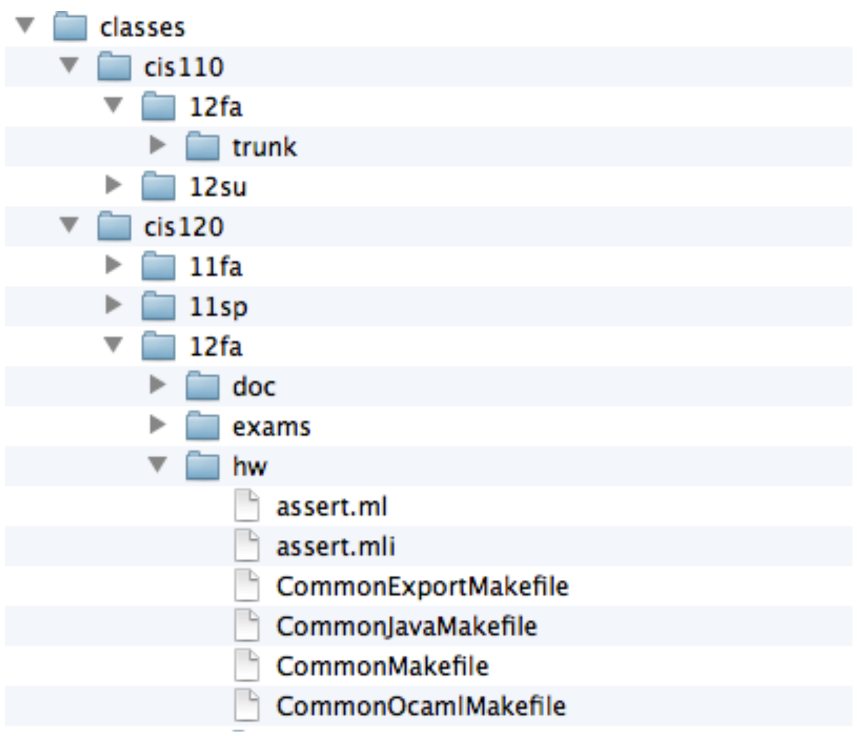
Family trees



Organizational charts

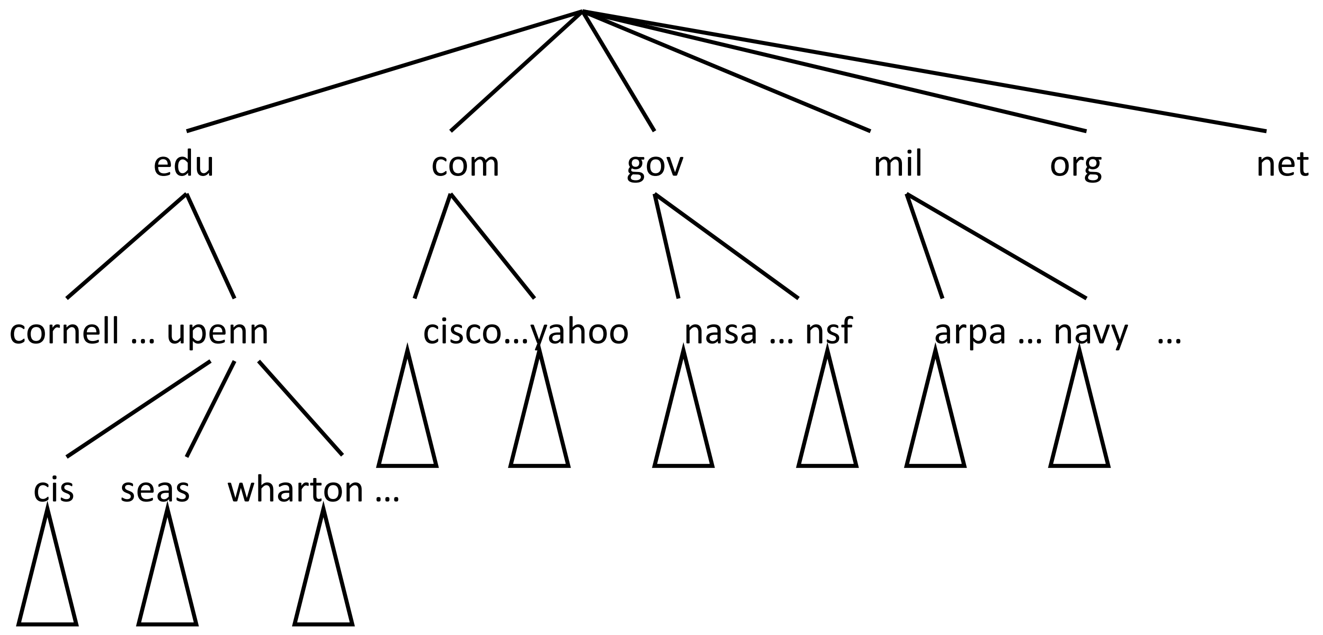


Filesystem Folder Structure

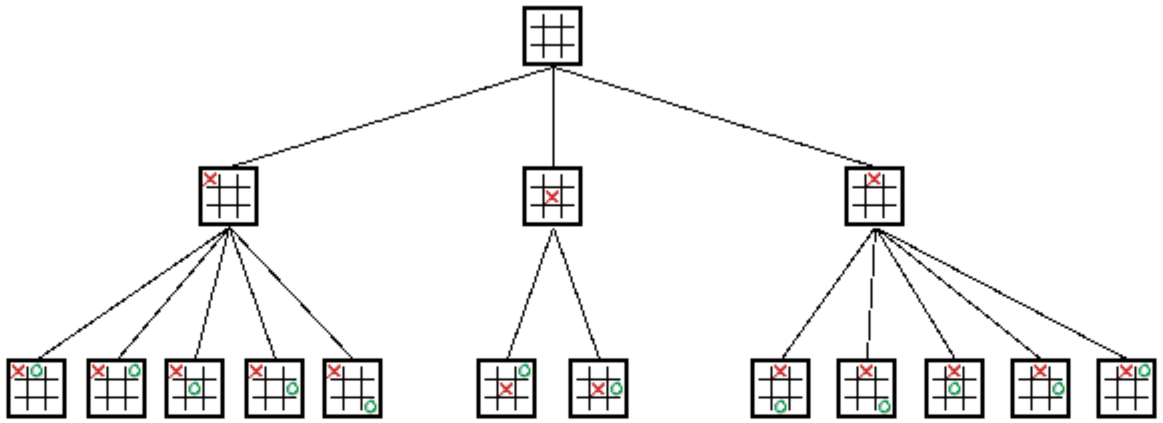


```
▼ classes
  ▼ cis110
    ▼ 12fa
      ▶ trunk
    ▶ 12su
  ▼ cis120
    ▶ 11fa
    ▶ 11sp
    ▼ 12fa
      ▶ doc
      ▶ exams
      ▼ hw
          assert.ml
          assert.mli
          CommonExportMakefile
          CommonJavaMakefile
          CommonMakefile
          CommonOcamlMakefile
```

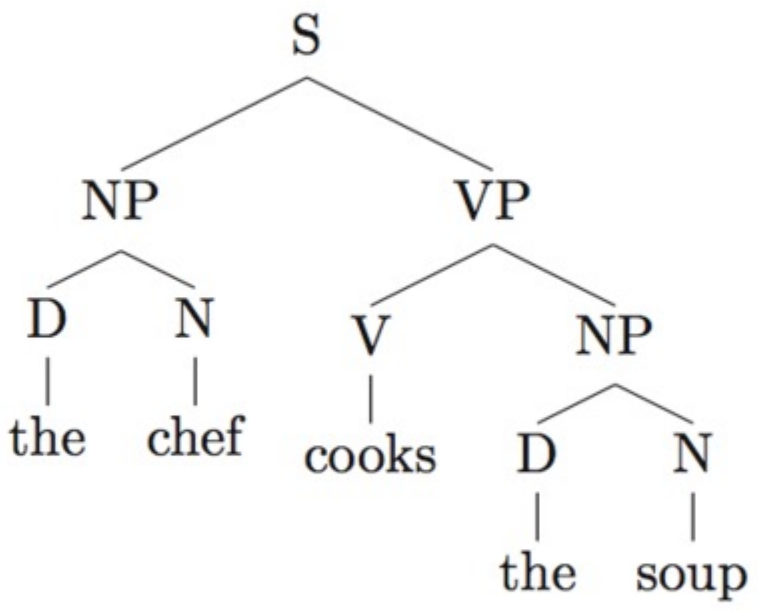
Domain Name Hierarchy



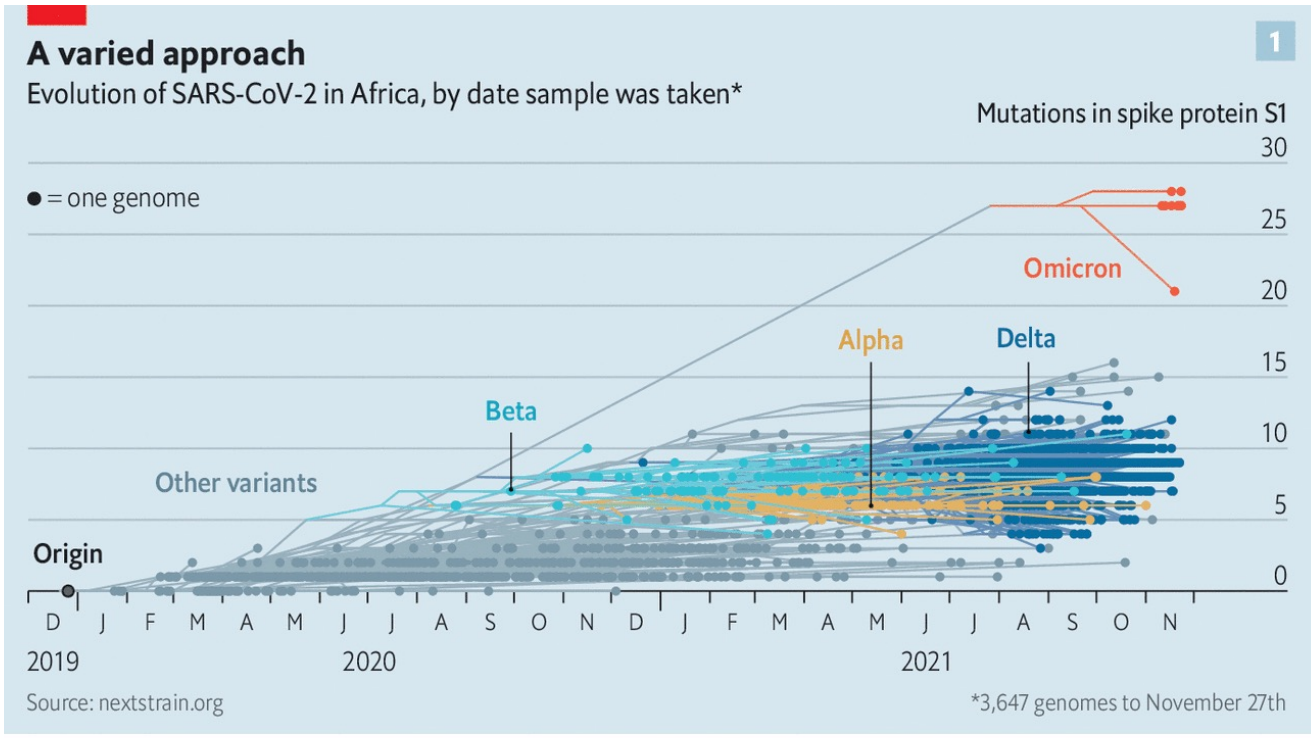
Game trees



Natural-Language Parse Trees



COVID evolutionary tree

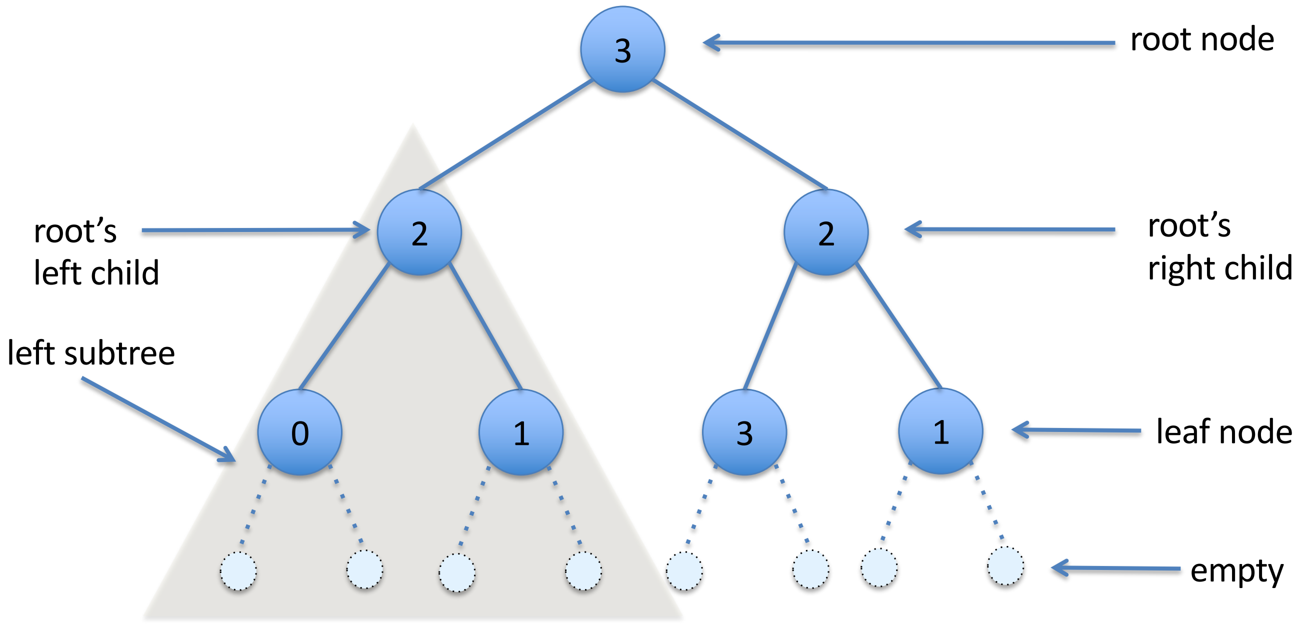


The Economist

Binary Trees

A particular form of tree-structured data

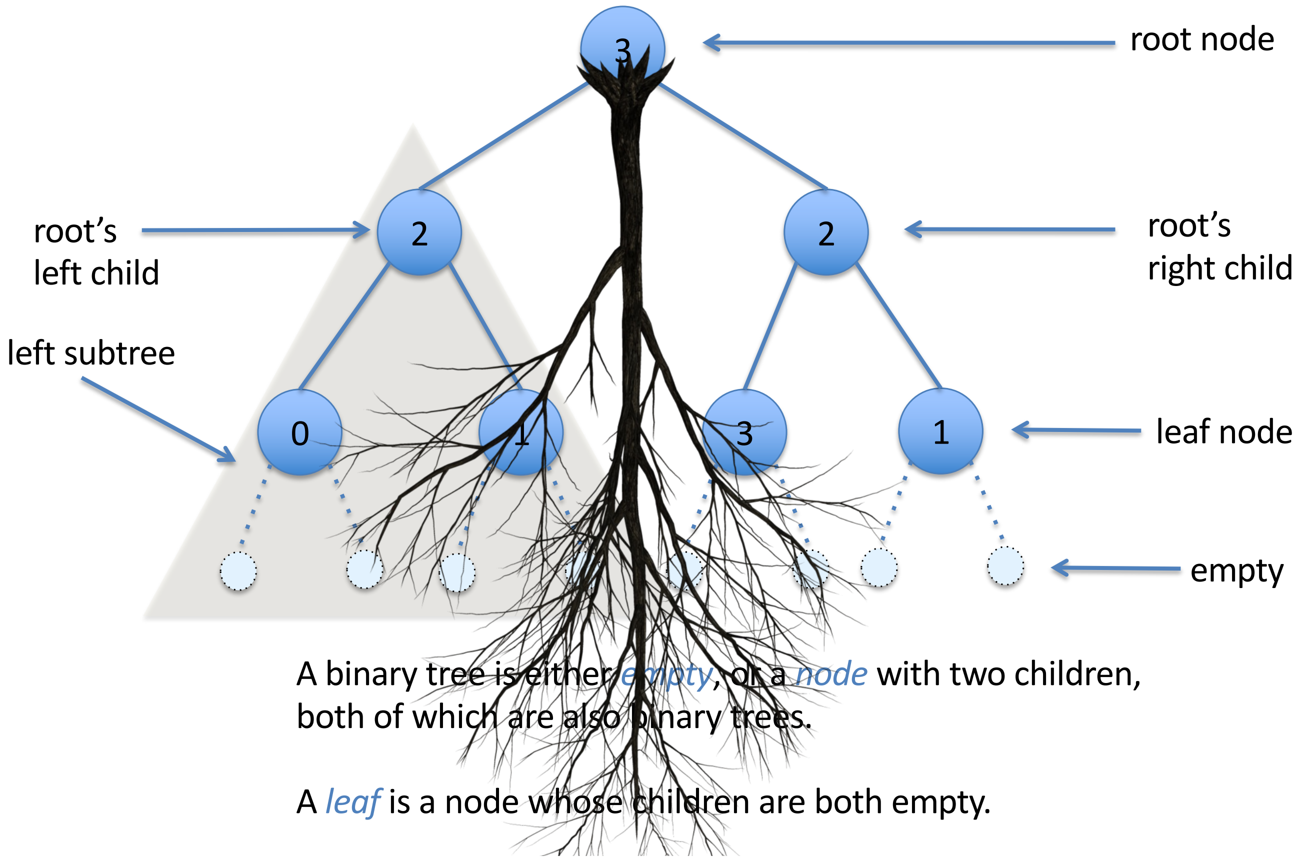
Binary Trees



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

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

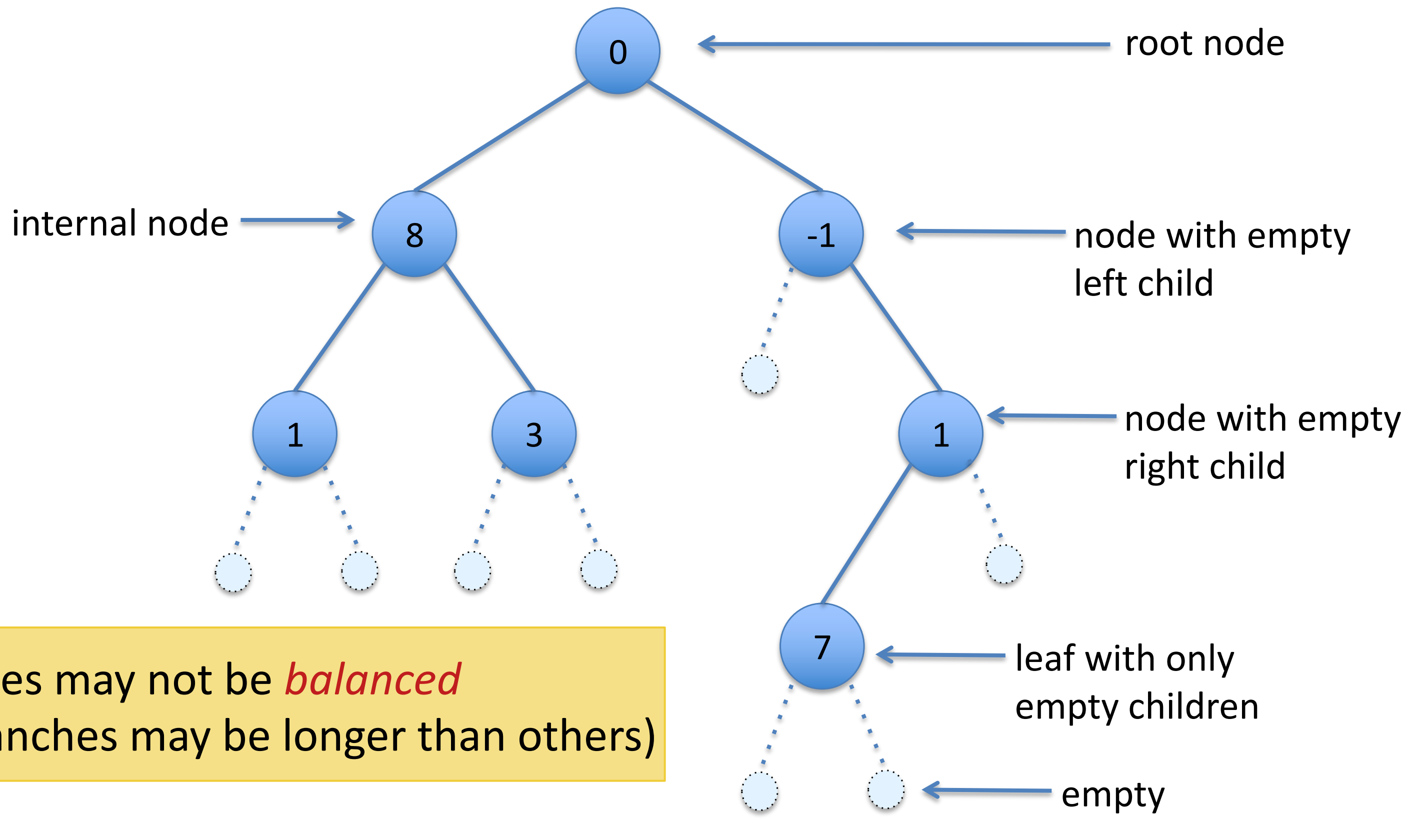
Trees are drawn upside-down



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

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

Another Binary Tree



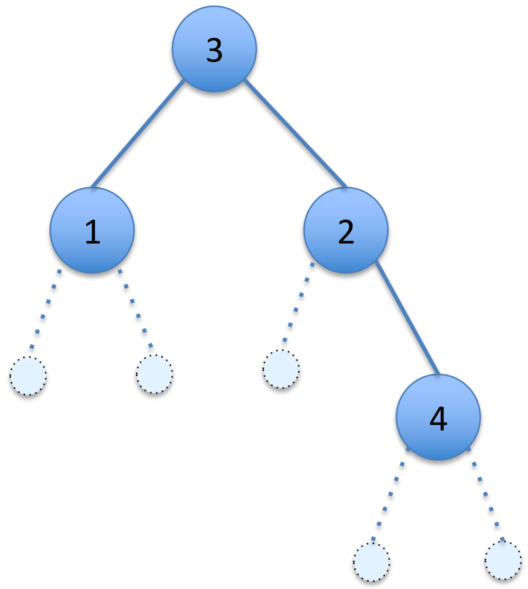
Binary trees may not be *balanced*
(some branches may be longer than others)

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)))
```

=



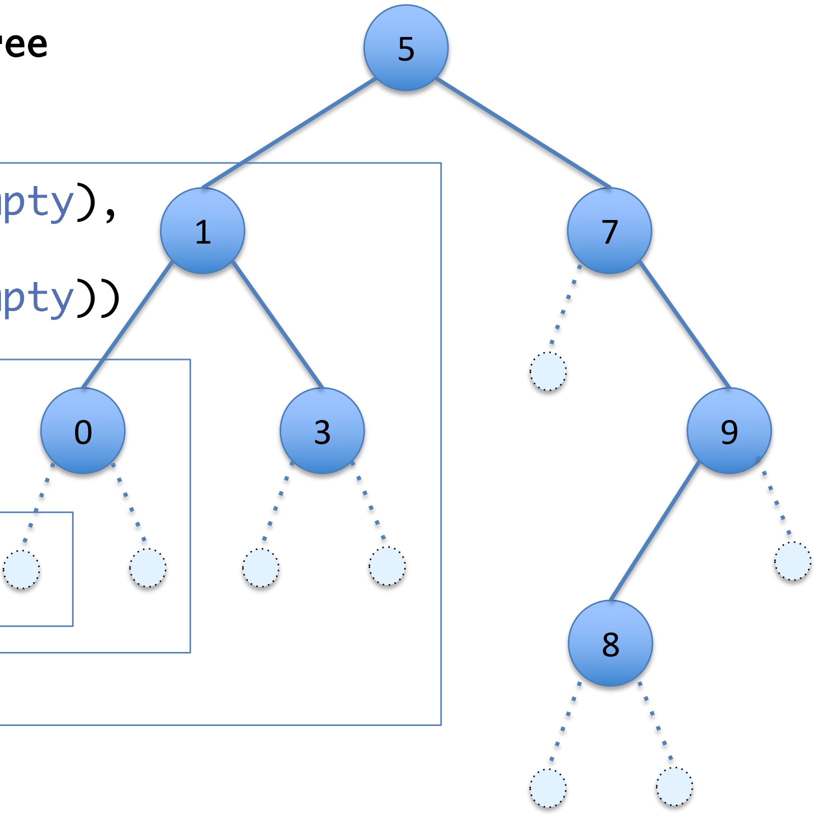
Representing trees

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

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

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

```
Empty
```



Working 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 …) … x … (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 ) -> … hd … (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…

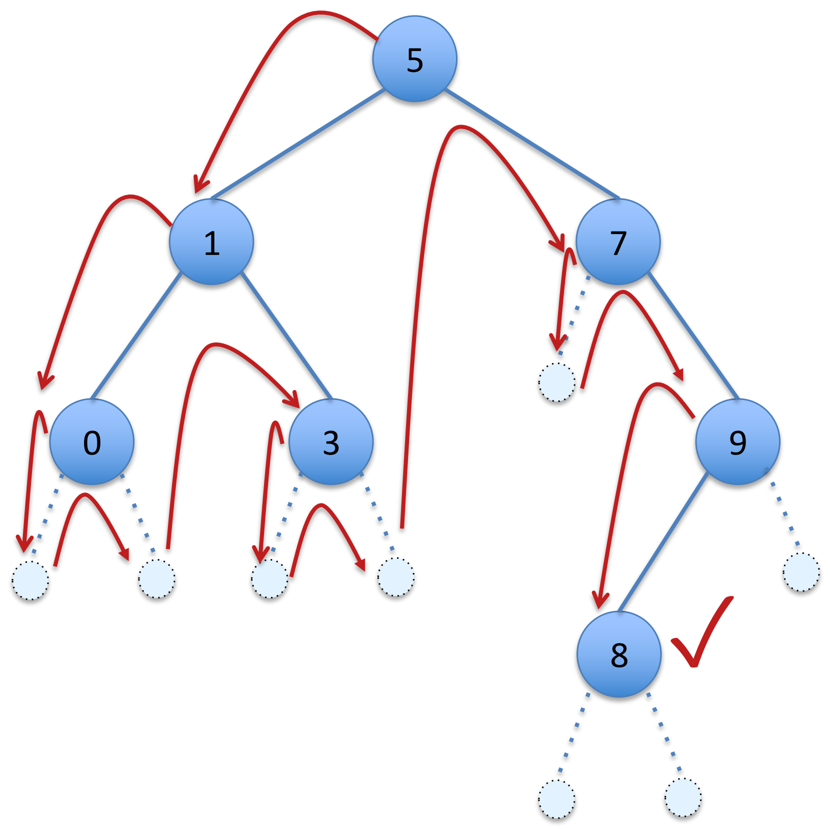
Searching for Data in a Tree

```
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
```

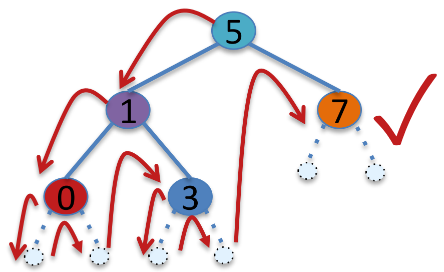
- This function searches through the tree, looking for n
- In the worst case, it might have to traverse the *entire tree*

The || operator is Boolean "or"

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
```

Skipping some steps…

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

Skipping some steps…

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