each of (Int, String)

2. **Sum types** (**one-of**): a value of `T` contains a value of `T1` *or* a value of `T2` **[done]**

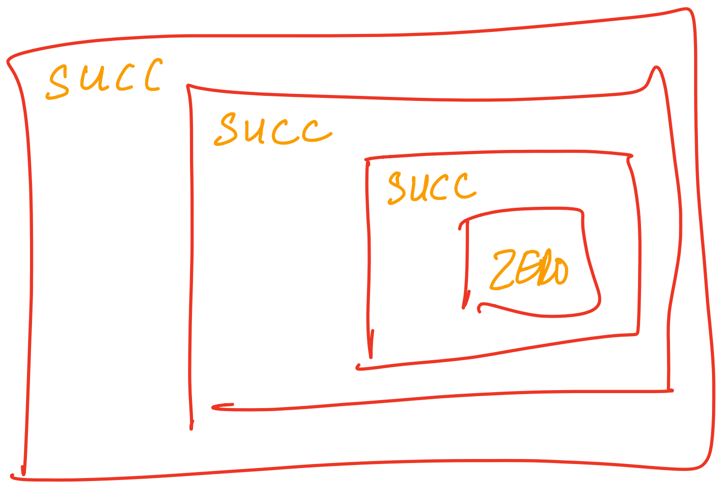
    - Union (*sum*) of two sets: $v(T) = v(T1) \cup v(T2)$

3. **Recursive types**: a value of `T` contains a *sub-value* of the same type `T`

# *Recursive types*

Let's define **natural numbers** from scratch:

```
data Nat = ???
```



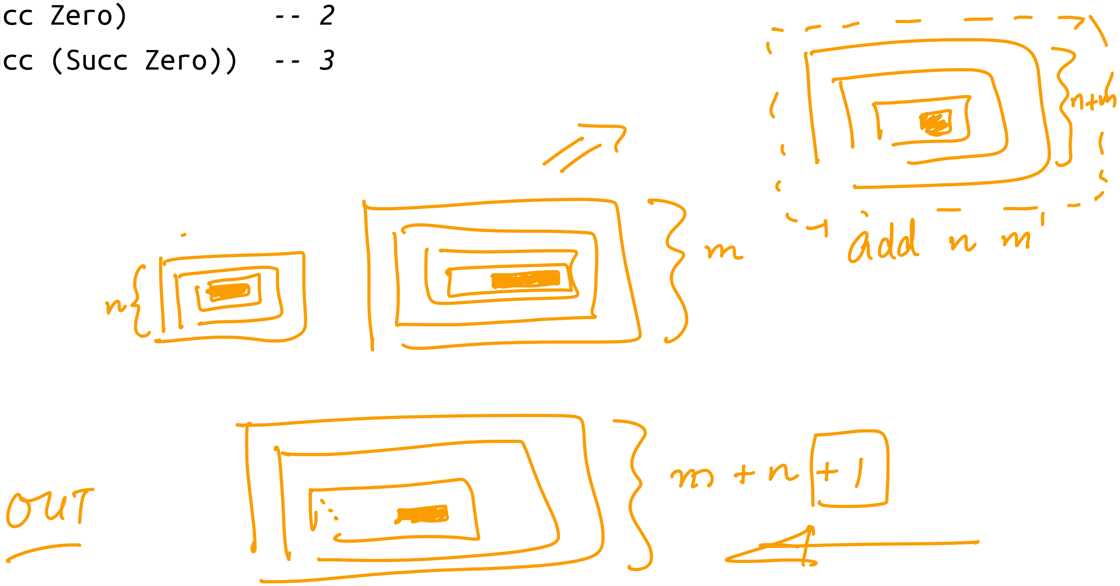
```haskell
data Nat = Zero | Succ Nat
```

A `Nat` value is:

- either an *empty* box labeled `Zero`
- or a box labeled `Succ` with another `Nat` in it!

Some `Nat` values:

```haskell
Zero                      -- 0
Succ Zero                 -- 1
Succ (Succ Zero)          -- 2
Succ (Succ (Succ Zero))   -- 3
...
```



zero - m = zero
m - zero = m

# Functions on recursive types

**Recursive code mirrors recursive data**

## 1. Recursive type as a parameter

```
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor
```

**Step 1:** add a pattern per constructor

```
toInt :: Nat -> Int
toInt Zero     = ... -- base case
toInt (Succ n) = ... -- inductive case
                     -- (recursive call goes here)
```

**Step 2:** fill in base case:

```
toInt :: Nat -> Int
toInt Zero     = 0   -- base case
toInt (Succ n) = ... -- inductive case
                     -- (recursive call goes here)
```

**Step 2:** fill in inductive case using a recursive call:

```
toInt :: Nat -> Int
toInt Zero     = 0            -- base case
toInt (Succ n) = 1 + toInt n -- inductive case
```

# *QUIZ*

What does this evaluate to?

```
let foo i = if i <= 0 then Zero else Succ (foo (i - 1))
in foo 2
```

**A.** Syntax error

**B.** Type error

**C.** `2`

**D.** `Succ Zero`

**E.** `Succ (Succ Zero)`

## *2. Recursive type as a result*

```haskell
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor
```

```haskell
fromInt :: Int -> Nat
fromInt n
  | n <= 0    = Zero                     -- base case
  | otherwise = Succ (fromInt (n - 1)) -- inductive case
                                         -- (recursive call goes here)
```

## *3. Putting the two together*

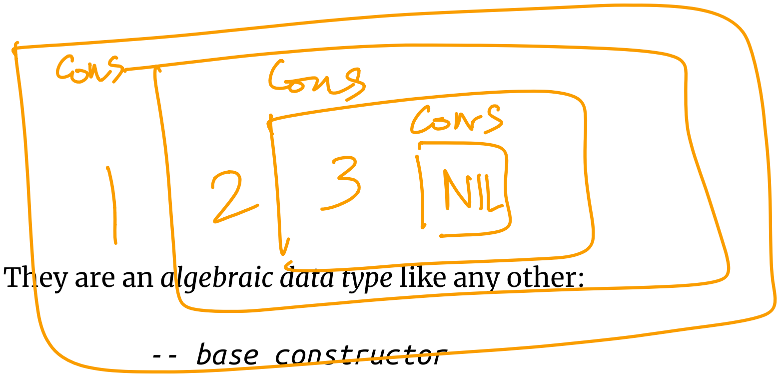
```haskell
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor


add :: Nat -> Nat -> Nat
add n m = ???

sub :: Nat -> Nat -> Nat
sub n m = ???
```

```haskell
data Nat = Zero     -- base constructor
         | Succ Nat -- inductive constructor
```

```haskell
add :: Nat -> Nat -> Nat
add Zero     m = m               -- base case
add (Succ n) m = Succ (add n m) -- inductive case

sub :: Nat -> Nat -> Nat
sub n        Zero     = n       -- base case 1
sub Zero     _        = Zero    -- base case 2
sub (Succ n) (Succ m) = sub n m -- inductive case
```

Lessons learned:

- **Recursive code mirrors recursive data**
- With **multiple** arguments of a recursive type, which one should I recurse on?
- The name of the game is to pick the right **inductive strategy**!




# *Lists*

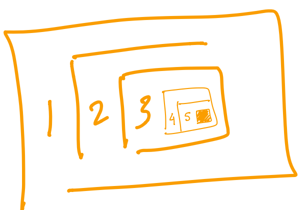
Lists aren't built-in! They are an *algebraic data type* like any other:

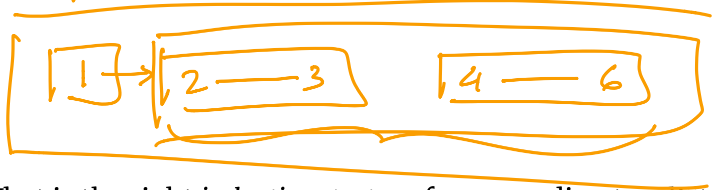
```
data List = Nil              -- base constructor
          | Cons Int List -- inductive constructor
```

- List `[1, 2, 3]` is *represented* as `Cons 1 (Cons 2 (Cons 3 Nil))`
- Built-in list constructors `[]` and `(:)` are just fancy syntax for `Nil` and `Cons`

Functions on lists follow the same general strategy:

```
length :: List -> Int
length Nil         = 0                  -- base case
length (Cons _ xs) = 1 + length xs   -- inductive case
```





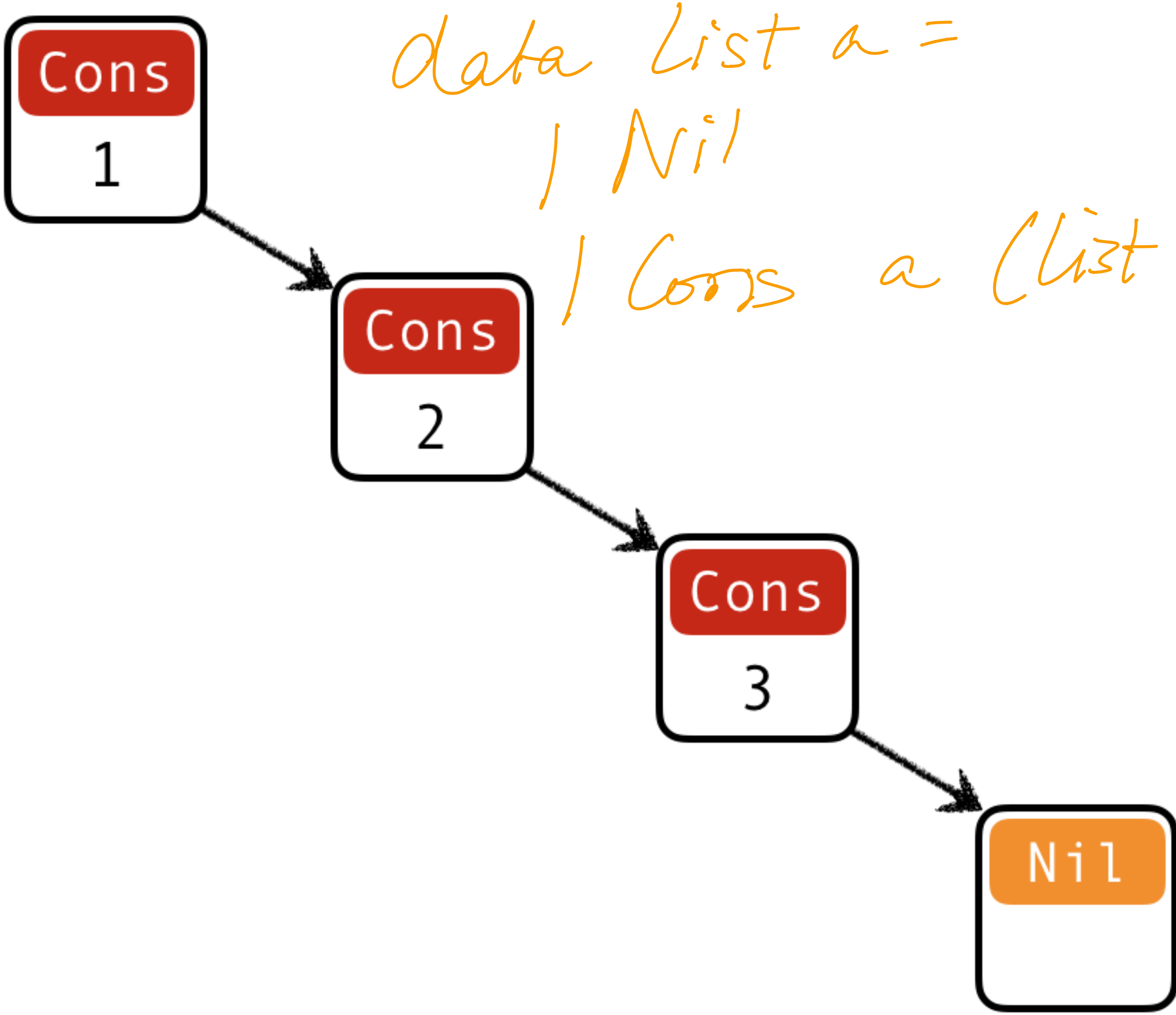

What is the right *inductive strategy* for appending two lists?

```
append :: List -> List -> List
append xs ys = ??
```

# *Trees*

Lists are *unary trees* with elements stored in the nodes:

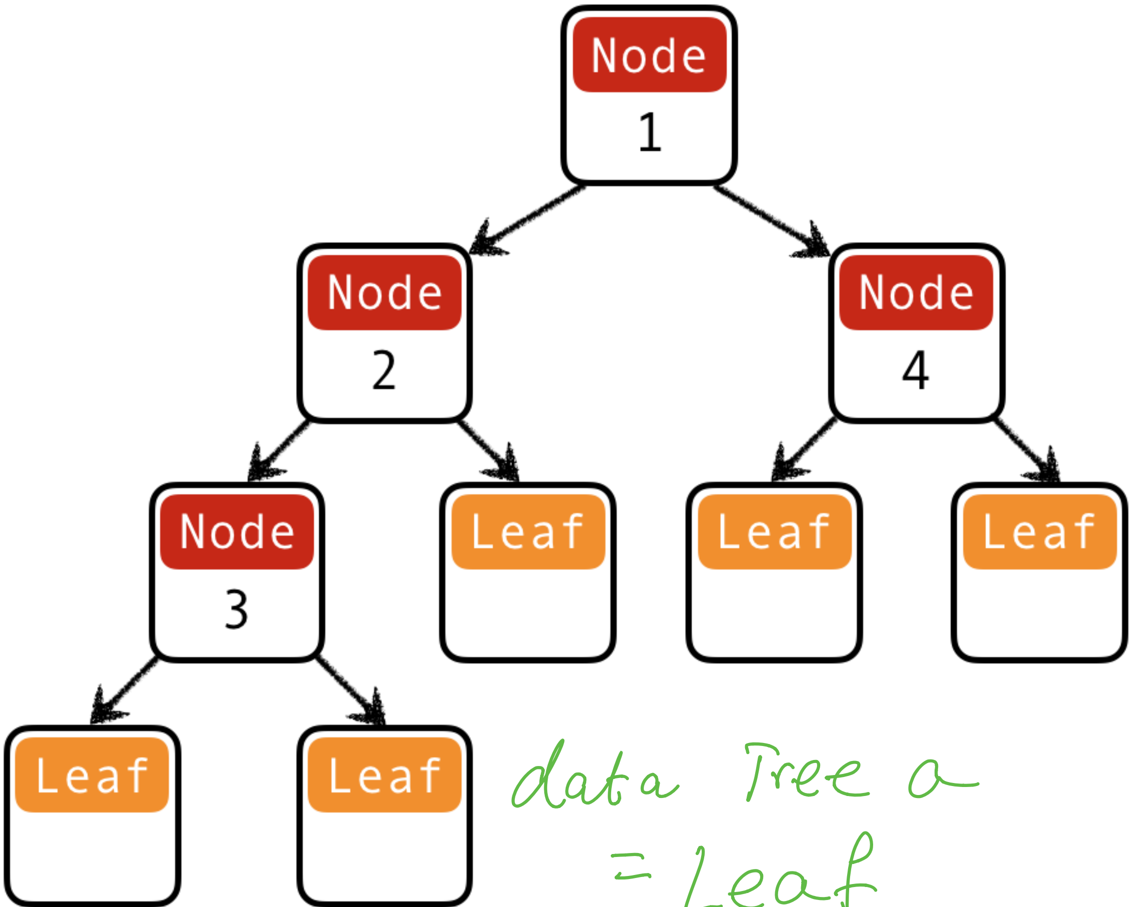
```
data List a =
  | Nil
  | Cons a (List a)
```



Lists are unary trees

```
data List = Nil | Cons Int List
```

How do we represent *binary trees* with elements stored in the nodes?

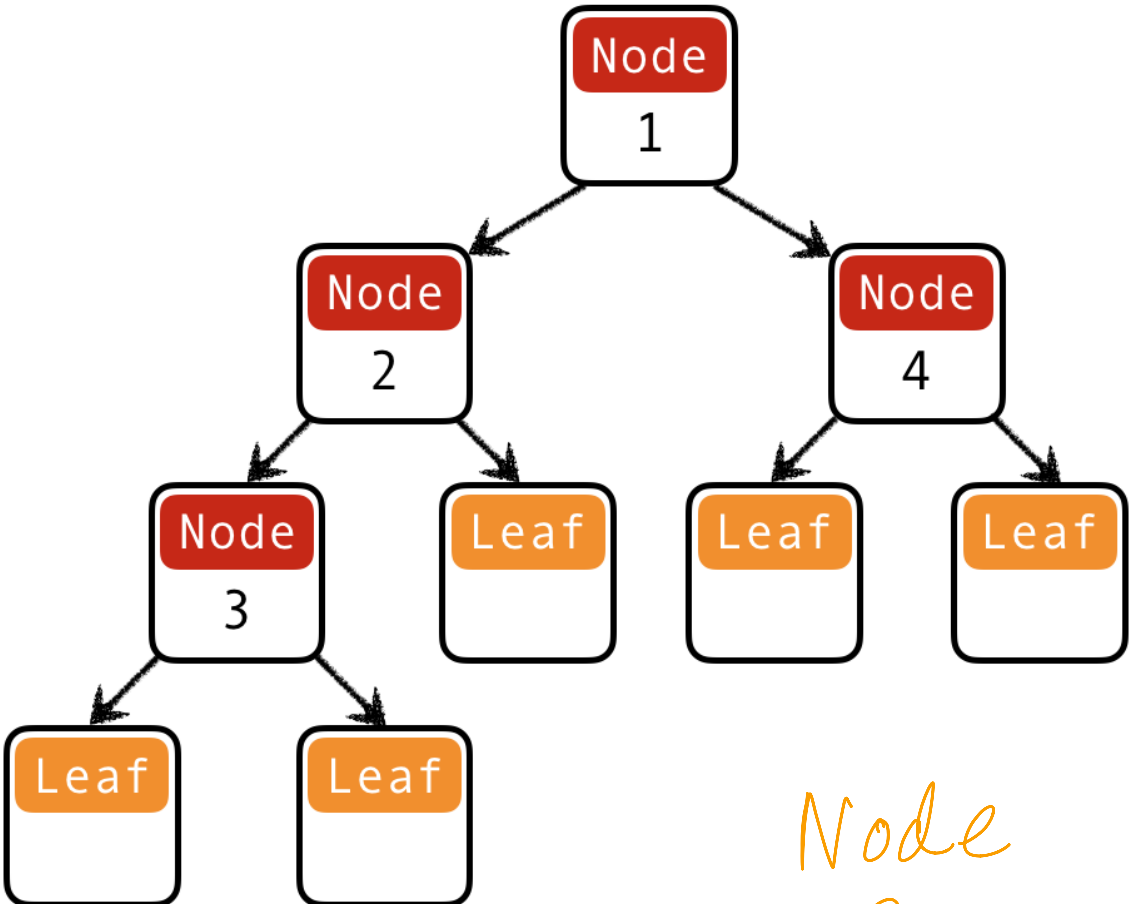


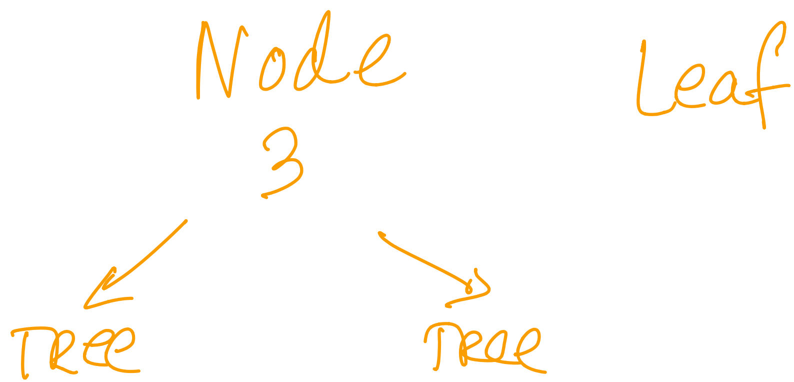

Binary trees with data at nodes

```
data Tree a
  = Leaf
  | Node a (Tree a) (Tree a)
```

# *QUIZ: Binary trees I*

What is a Haskell datatype for *binary trees* with elements stored in the nodes?




Binary trees with data at nodes

```
(A) data Tree = Leaf | Node Int Tree
```

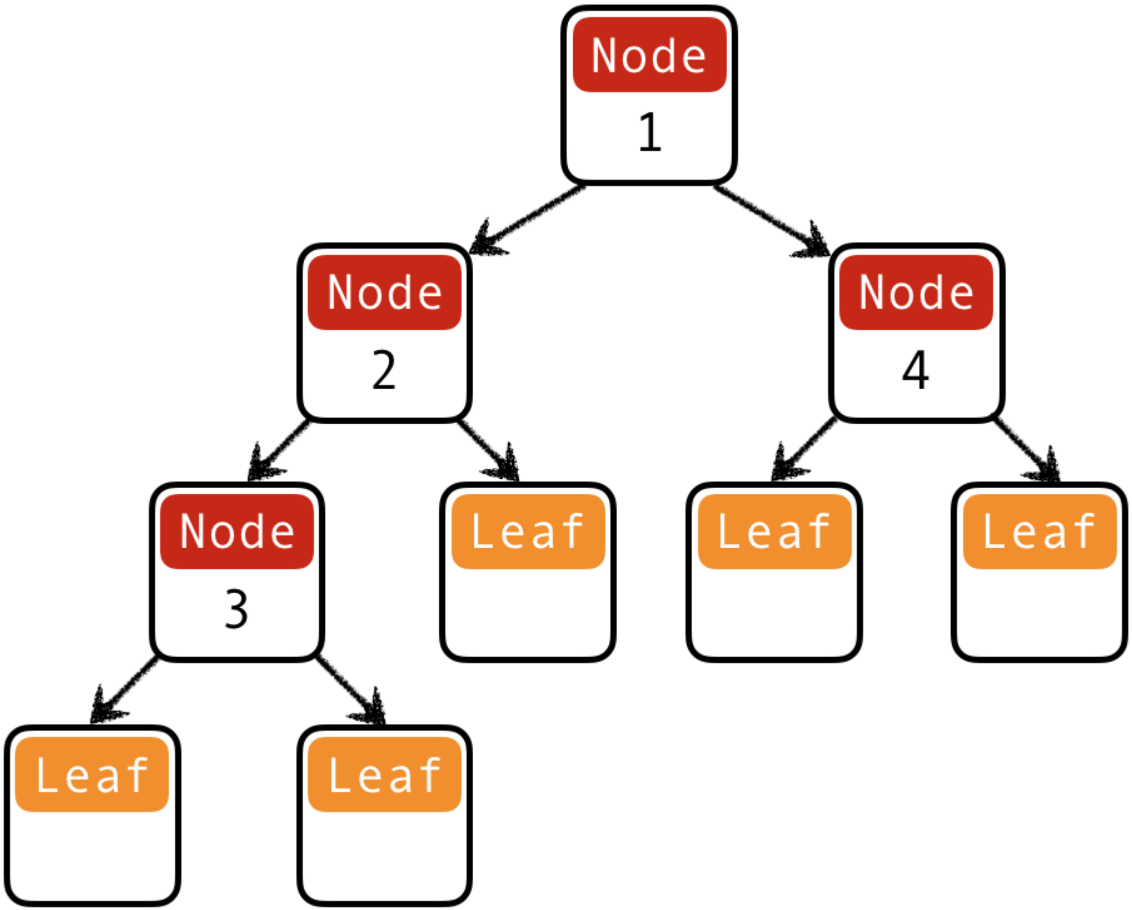
X

(B) **data** Tree = Leaf | Node Tree Tree

(C) **data** Tree = Leaf | Node Int Tree Tree

(D) **data** Tree = Leaf Int | Node Tree Tree

(E) **data** Tree = Leaf Int | Node Int Tree Tree




Binary trees with data at nodes

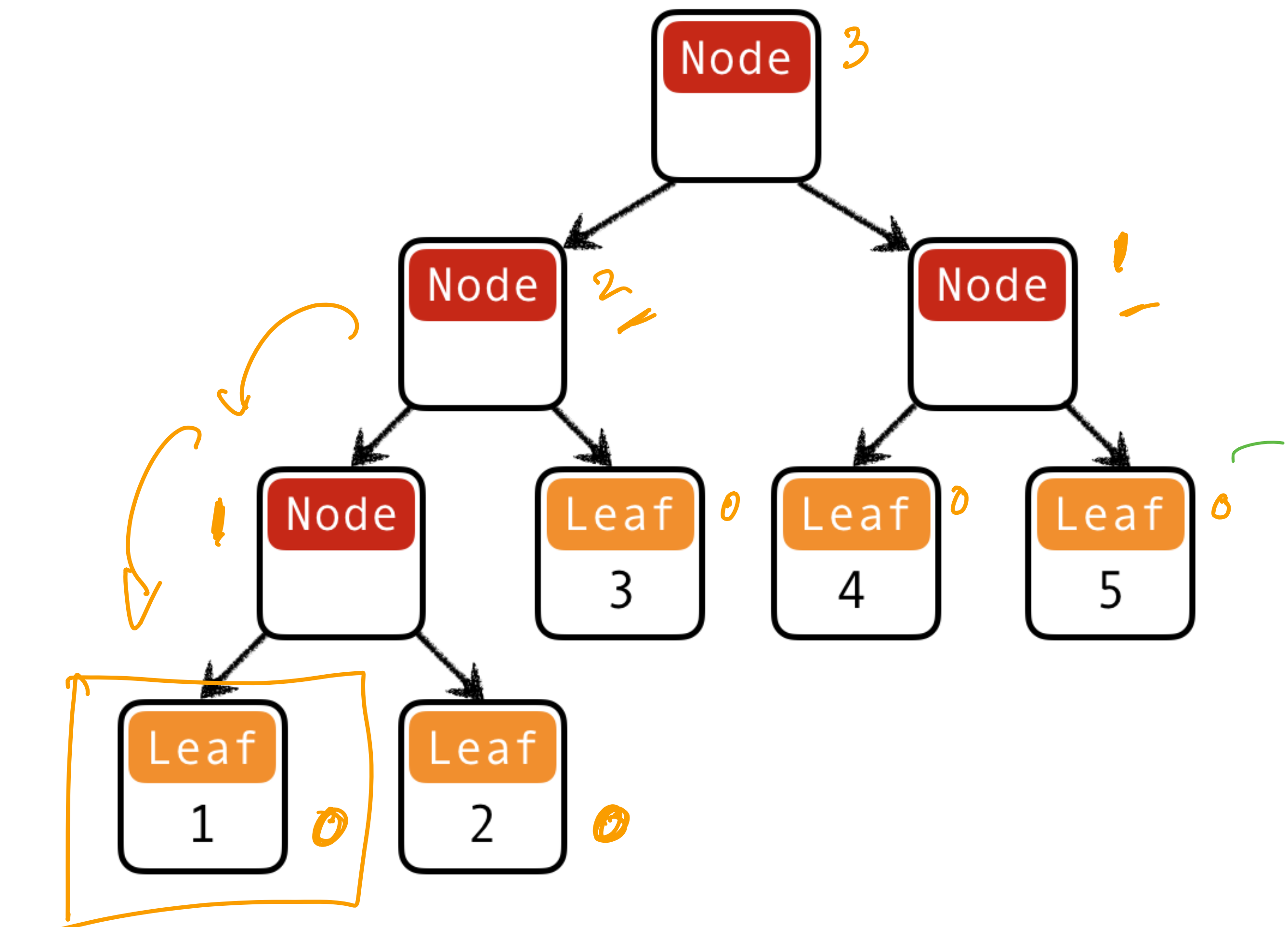
```haskell
data Tree = Leaf | Node Int Tree Tree

t1234 = Node 1
          (Node 2 (Node 3 Leaf Leaf) Leaf)
          (Node 4 Leaf Leaf)
```

# *Functions on trees*

```haskell
depth :: Tree -> Int
depth t = ??
```

# *QUIZ: Binary trees II*

What is a Haskell datatype for *binary trees* with elements stored in the leaves?




Binary trees with data at leaves

(A) **data** `Tree = Leaf | Node Int Tree`

(B) `data Tree = Leaf | Node Tree Tree`

(C) `data Tree = Leaf | Node Int Tree Tree`

left right

(D) `data Tree = Leaf Int | Node Tree Tree`

(E) `data Tree = Leaf Int | Node Int Tree Tree`

```
data Tree = Leaf Int | Node Tree Tree

t12345 = Node
          (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3))
          (Node (Leaf 4) (Leaf 5))
```

# *Example: Calculator*

I want to implement an arithmetic calculator to evaluate expressions like:

- 2.3
- `4.0 + 2.9`
- `3.78 - 5.92`
- `(4.0 + 2.9) * (3.78 - 5.92)`

Enum 2.3

EPlus (ENum 4.0) (ENum 2.9)

What is a Haskell datatype to *represent* these expressions?

```
data Expr = ???
```

```
= ENum Double
| EPlus Expr Expr
| EMinus Expr Expr
| EMul Expr Expr
```

```haskell
data Expr = Num Float
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
```

How do we write a function to *evaluate* an expression?

```haskell
eval :: Expr -> Float
eval e = ???
```

# *Recursion is...*

Building solutions for *big problems* from solutions for *sub-problems*

- **Base case:** what is the *simplest version* of this problem and how do I solve it?
- **Inductive strategy:** how do I *break down* this problem into sub-problems?
- **Inductive case:** how do I solve the problem *given* the solutions for subproblems?

# *Why use Recursion?*

1. Often far simpler and cleaner than loops
    - But not always...
2. Structure often forced by recursive data
3. Forces you to factor code into reusable units (recursive functions)

# *Why **not** use Recursion?*

1. Slow

2. Can cause stack overflow

# *Example: factorial*

```
fac :: Int -> Int
fac n
  | n <= 1    = 1
  | otherwise = n * fac (n - 1)
```

Lets see how `fac 4` is evaluated:

```
<fac 4>
  ==> <4 * <fac 3>>                -- recursively call `fact 3`
  ==> <4 * <3 * <fac 2>>>          --   recursively call `fact 2`
  ==> <4 * <3 * <2 * <fac 1>>>>    --     recursively call `fact 1`
  ==> <4 * <3 * <2 * 1>>>          --     multiply 2 to result
  ==> <4 * <3 * 2>>                --   multiply 3 to result
  ==> <4 * 6>                      -- multiply 4 to result
  ==> 24
```

Each *function call* `<>` allocates a frame on the *call stack*

- expensive
- the stack has a finite size

Can we do recursion without allocating stack frames?

# *Tail Recursion*

Recursive call is the *top-most* sub-expression in the function body

- i.e. no computations allowed on recursively returned value
- i.e. value returned by the recursive call == value returned by function

# *QUIZ: Is this function tail recursive?*

```
fac :: Int -> Int
fac n
  | n <= 1    = 1
  | otherwise = n * fac (n - 1)
```

**A.** Yes

**B.** No

# *Tail recursive factorial*

Let's write a tail-recursive factorial!

```
facTR :: Int -> Int
facTR n = ...
```

Lets see how `facTR` is evaluated:

```
<facTR 4>
  ==>    <<loop 1  4>> -- call loop 1 4
  ==>   <<<loop 4  3>>> -- rec call loop 4 3
  ==>  <<<<loop 12 2>>>> -- rec call loop 12 2
  ==> <<<<<loop 24 1>>>>> -- rec call loop 24 1
  ==> 24                  -- return result 24!
```

Each recursive call **directly** returns the result

- without further computation
- no need to remember what to do next!
- no need to store the "empty" stack frames!

# *Why care about Tail Recursion?*

Because the *compiler* can transform it into a *fast loop*

```
facTR n = loop 1 n
  where
    loop acc n
      | n <= 1    = acc
      | otherwise = loop (acc * n) (n - 1)
```

```
function facTR(n){
  var acc = 1;
  while (true) {
    if (n <= 1) { return acc ; }
    else        { acc = acc * n; n = n - 1; }
  }
}
```

- Tail recursive calls can be optimized as a **loop**
    - no stack frames needed!

- Part of the language specification of most functional languages
    - compiler **guarantees** to optimize tail calls

That's all folks!

---