15-150 Fall 2023

Lecture 8

Stephen Brookes

"I had a pretty good day. For a little while, my computer and I were both functional at the same time."


WARNING
SPOILER
ALERT

the plan

- ***searching*** and ***sorting*** trees

  under various assumptions
  (arbitrary, sorted, balanced)

  - correctness

  - work and span

tree basics

datatype 'a tree = Empty | Node **of** 'a tree * 'a * 'a tree

- size T = number of nodes
- depth T = length of longest path
    - A *full binary tree* of depth d has size $2^d - 1$
    - depth T is O(log (size T)) for a balanced tree,
      depth T is O(size T) otherwise

conventions

- I prefer T for trees, t for types (and tea to drink)
- I often use capitalized names for datatype constructors like Node, Empty, SOME, NONE
  - Not required by ML, but be consistent…

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree;

fun size empty = 0
|    size (Node(A, _, B)) = 1 + size A + size B
```

What happens?

inorder traversal

```
fun inord Empty = [ ]
|    inord (Node(T1, x, T2)) = (inord T1) @ x :: (inord T2)
```

- inord T = inorder traversal list for T
- length(inord T) = size T

balanced trees

- Empty is *balanced*
- Node(A, x, B) is *balanced* iff

  |size(A) - size(B)| ≤ 1
  and A, B are *balanced*

A **structurally inductive** definition

- If T is balanced, every node of T is balanced
- If T is balanced, each child has about half the data

building a balanced tree

- To turn a list into a balanced tree…

```
fun list2tree [ ] = Empty
|     list2tree L =
   let
     val n = length L
     val (A, x::B) = takedrop (n div 2, L)
   in
     Node(list2tree A, x, list2tree B)
   end
```

takedrop(2, [1,2,3,4,5]) = ([1,2], [3,4,5])

list2tree [1,2,3,4,5] = ???

imprecision

- MANY trees can have the same *in-order* list
- We don't always **need** to specify which one!

```
list2tree : 'a list -> 'a tree
  ENSURES list2tree L = a balanced tree T such that inord T = L
```

sorted trees

Empty is *sorted*

Node(A, x, B) is *sorted* iff

every integer in A is $\leq$ x,
every integer in B is $\geq$ x,
and A and B are *sorted*

Theorem

T is a *sorted tree*
iff
inord T is a *sorted list*

motivation

Sorted data may be easier to deal with…

- That's why dictionaries are in lexicographic order!

Bantis zōbrie issa se ossȳngnoti lēdys
The night is dark and full of terrors.
Muña Zaldrizoti
Mother of dragons
Skorverdon dekuroti Dōros hen kesīr ilza?
How much farther is the Wall?
Valar dohaeris
All men must serve
Valar morghulis
All men must die

what we will do

Let's first look at functions for ***searching*** trees

- unsorted, sorted
- unbalanced, balanced

- We'll contrast the work and span.

searching

an unsorted tree

mem : int * int tree -> bool

```
fun mem (x, Empty) = false
|     mem (x, Node(A, y, B)) =
      (x = y) orelse mem (x,A) orelse mem (x, B)
                              (* not designed for parallel evaluation *)
```

ENSURES mem (x,T) = true iff x is in T

$W_{mem}(x,T)$ is O(size T)

$S_{mem}(x,T)$ is also O(size T)

searching

an unsorted tree

mem : int * int tree -> bool

```
fun mem (x, Empty) = false
|     mem (x, Node(A, y, B)) =
      (x = y) orelse
        let
          val (a, b) = (mem (x, A), mem (x, B))
        in                    (* designed for parallel evaluation *)
          a orelse b
        end
```

$W_{mem}(x, T)$ is $O(\text{size } T)$

$S_{mem}(x, T)$ is $O(\text{depth } T)$ ... let's see why

searching

an unsorted tree

```
fun mem (x, Empty) = false
|     mem (x, Node(A, y, B)) =
      (x = y) orelse
        let
          val (a, b) = (mem (x, A), mem (x, B))
        in
          a orelse b
        end
```

Let $\mathbf{S_{mem}(d)}$ be the span for mem(x,T) when T has depth d

(worst-case: when T is "list-like")

$\mathbf{S_{mem}(0) = 1}$

$\mathbf{S_{mem}(d) = 1 + S_{mem}(d\text{-}1)}$ **(why?)**

Hence $S_{mem}(d)$ is $O(d)$

searching a *sorted* tree

```
fun mem (x, Empty) = false
|     mem (x, Node(A, y, B)) =
              case Int.compare(x, y) of
                    LESS       => mem(x, A)
                  | EQUAL      => true
                  | GREATER => mem (x, B)
```

REQUIRES T is sorted
ENSURES mem (x, T) = true iff x is in T

$W_{mem}(x, T)$ is O(depth T)

$S_{mem}(x, T)$ is O(depth T)

check these

search summary

	work	span	worst-case (n items)
unsorted tree	O(size)	O(depth)	**work O(n)** **span O(n)**
sorted tree	O(depth)	O(depth)	**work O(n)** **span O(n)**
balanced unsorted tree	O(size)	O(depth)	**work O(n)** **span O(log n)**
balanced sorted tree	O(depth)	O(depth)	**work O(log n)** **span O(log n)**

- For a balanced tree T we know that **depth T** is **O(log(size T))**

motivation

- Trees may be *better* than lists…

  … and balanced trees may be *even better*.

- And *sorted* trees may enable *even faster* code.

Let's develop a function
that sorts a tree (of integers)

sorting a tree

- If the tree is Empty, do nothing
- Otherwise

  (recursively) *sort* the two children, then

  merge the sorted children, then

  insert the root value

We'll design helpers to *insert* and *merge*

merge will also need a helper
to *split* a tree in two

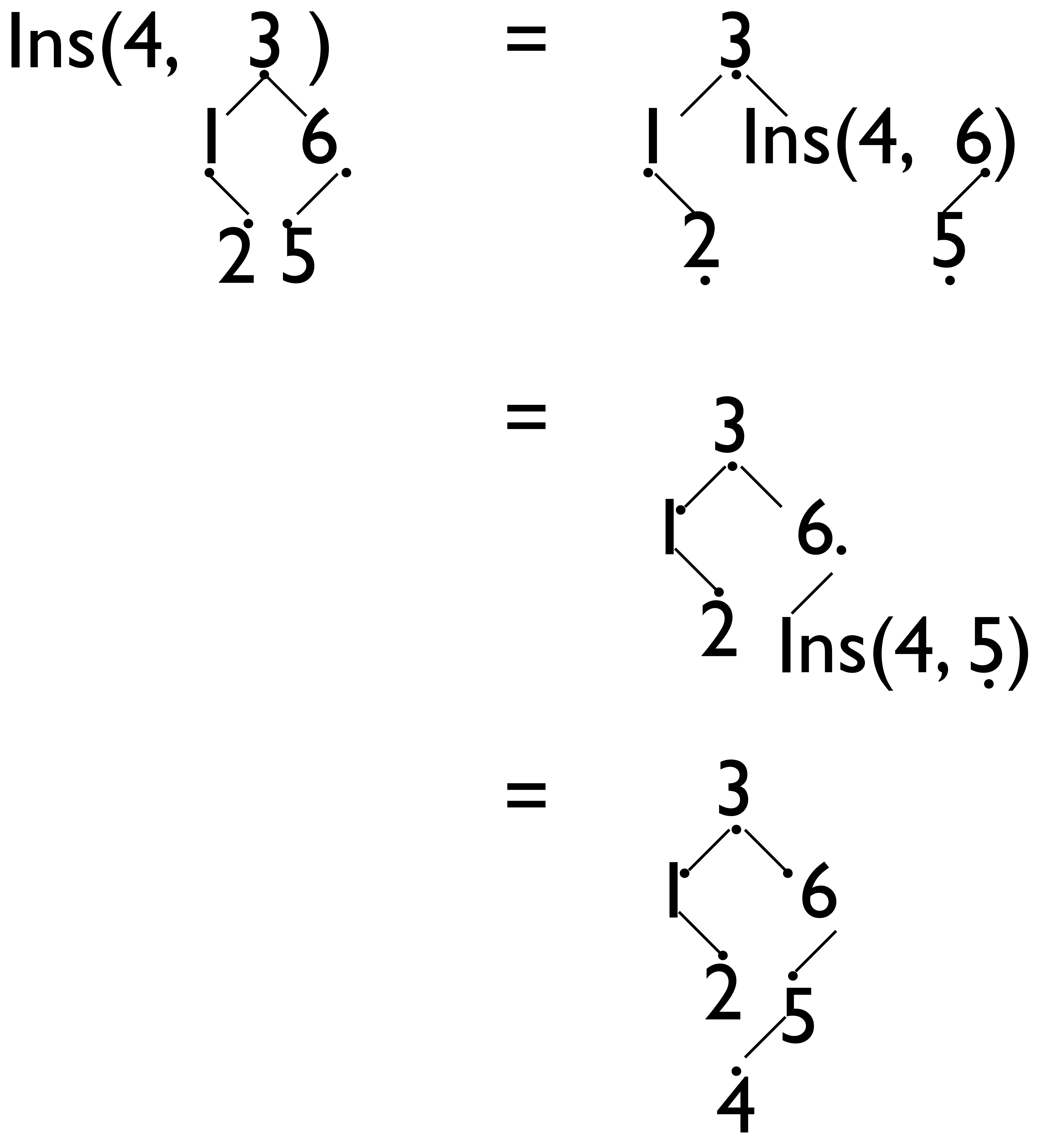
inserting in a tree

Ins : int * int tree -> int tree
REQUIRES T is a sorted tree
ENSURES Ins(x,T) = a sorted tree
consisting of x and T

```
fun Ins (x, Empty) = Node(Empty, x, Empty)
|     Ins (x, Node(T1, y, T2)) =
    if x > y then Node(T1, y, Ins(x, T2))
              else  Node(Ins(x, T1), y, T2)
```

(contrast with list insertion)

example



merging trees

Merge : int tree * int tree -> int tree

REQUIRES T_1 and T_2 are sorted trees

ENSURES Merge(T_1,T_2) = a sorted tree consisting of T_1 and T_2

Merge (Node(L_1,x,R_1), T_2) = ???

We could *split* T_2 into two subtrees (L_2, R_2), then do Node(Merge(L_1 ,L_2), x, Merge(R_1 ,R_2))

But we need to *stay sorted* and *not lose data*…

… so *split* should use x and build (L_2, R_2) so that $L_2 \leq x \leq R_2$ …

splitting a tree

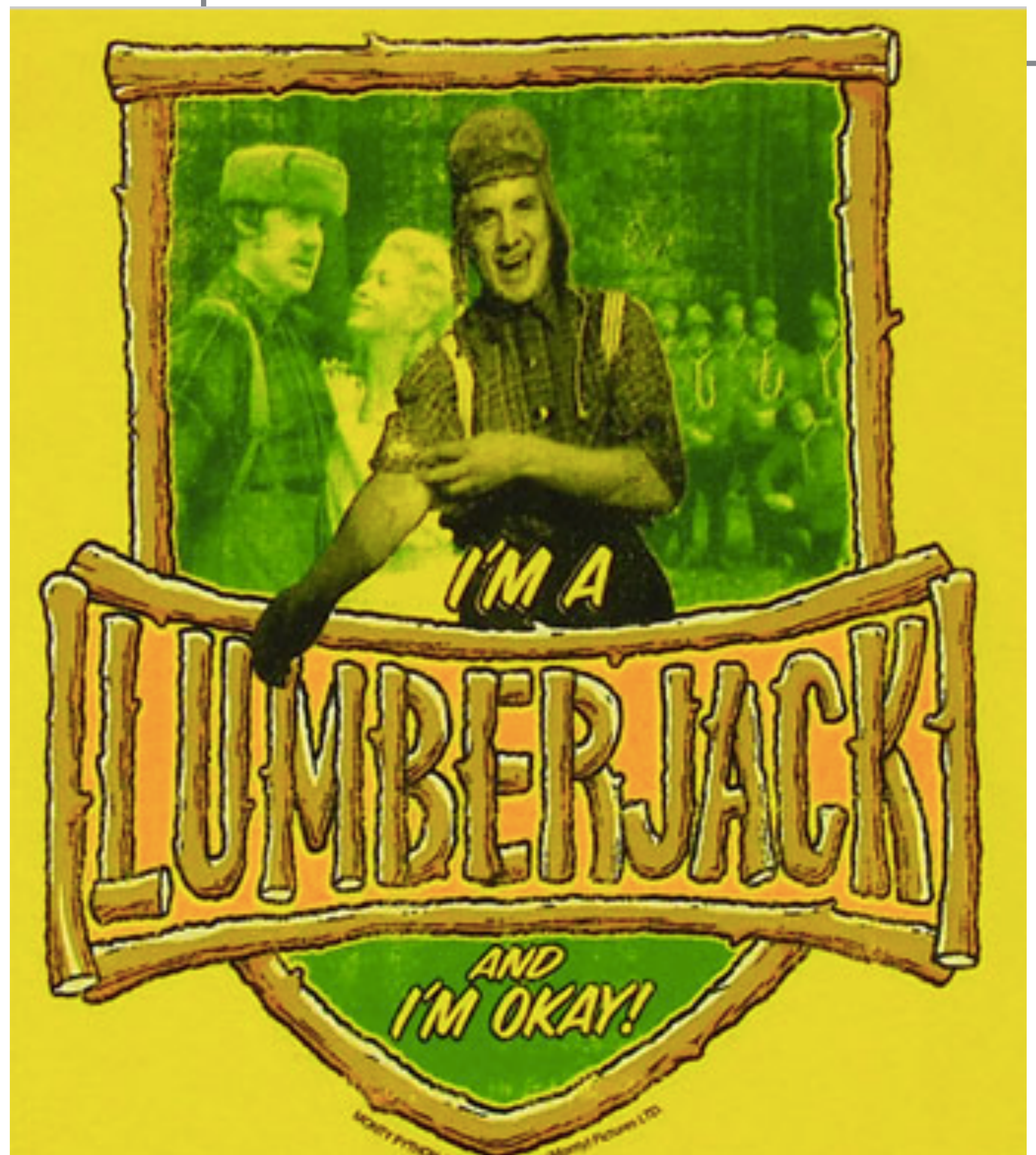
SplitAt : int * int tree -> int tree * int tree

REQUIRES T is a sorted tree

ENSURES SplitAt(x, T) =

a pair of sorted trees (U_1, U_2) such that

$U_1 \leq x \leq U_2$ and U_1, U_2 is a perm of T



Not completely precise, but that's OKAY!

Plan

Define SplitAt(x, T) using ***structural recursion***

- SplitAt(x, Node(T_1, y, T_2)) should
  - *compare* x and y
  - call SplitAt(x, -) on T_1 or T_2
  - build the result

SplitAt

SplitAt : int * int tree -> int tree * int tree

REQUIRES T is a sorted tree

ENSURES SplitAt(x,T) = a pair of sorted trees (U_1, U_2)

such that $U_1 \leq x \leq U_2$ and U_1, U_2 is a perm of T

```
fun SplitAt(x, Empty) = (Empty, Empty)

|   SplitAt(x, Node(T1, y, T2)) =
    if y>x then
       let val (L1, R1) = SplitAt(x, T1) in (L1, Node(R1, y, T2)) end
          else
       let val (L2, R2) = SplitAt(x, T2) in (Node(T1, y, L2), R2) end
```

Merge

Merge : int tree * int tree -> int tree

REQUIRES T_1 and T_2 are sorted trees

ENSURES Merge(T_1, T_2) = a sorted tree consisting of T_1 and T_2

```
fun Merge (Empty, T2) = T2

  |  Merge (Node(L1, x, R1), T2) =
     let
       val (L2, R2) = SplitAt(x, T2)
     in
       Node(Merge(L1, L2), x, Merge(R1, R2))
     end
```

(as we promised!)

example

Standard ML of New Jersey

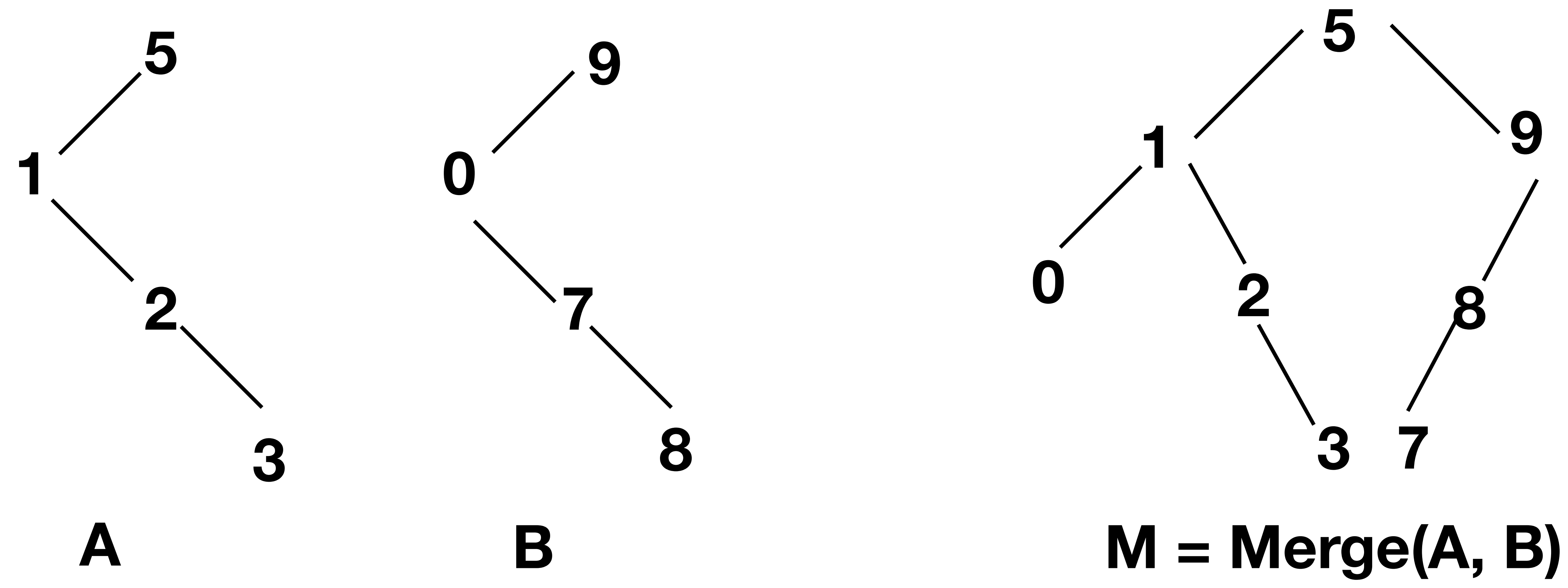
```
val A = Node (Node (Empty,1,Node (Empty,2,Node (Empty,3,Empty))),5,Empty)

val B = Node (Node (Empty,0,Node (Empty,7,Node (Empty,8,Empty))),9,Empty)

val M = Merge(A, B);

val M =
 Node
   (Node (Node (Empty,0,Empty),1,Node (Empty,2,Node (Empty,3,Empty))),5,
    Node (Node (Node (Empty,7,Empty),8,Empty),9,Empty)) : int tree

- inord M;
val it = [0,1,2,3,5,7,8,9] : int list
```



comments

- There's more than one way to split a tree,
  and many ways to merge two sorted trees into one.

- It's not always easy to see what results you'll get.

  - IT DOESN'T MATTER!

  - You just need to know that the results will
    satisfy the SPECIFICATION

**A sorted tree containing all
the items of T1 and T2**

Msort

Msort : int tree -> int tree

ENSURES Msort T = a sorted permutation of T

```
fun Msort Empty = Empty
 |  Msort (Node(T1, x, T2)) =
       Ins (x, Merge(Msort T1, Msort T2))
```

Correct?

- **Q**: How to *prove* that Msort is correct?
  A: Use structural induction.
- First prove that the *helper functions* Merge, SplitAt, Ins are correct.
  Again use structural induction.

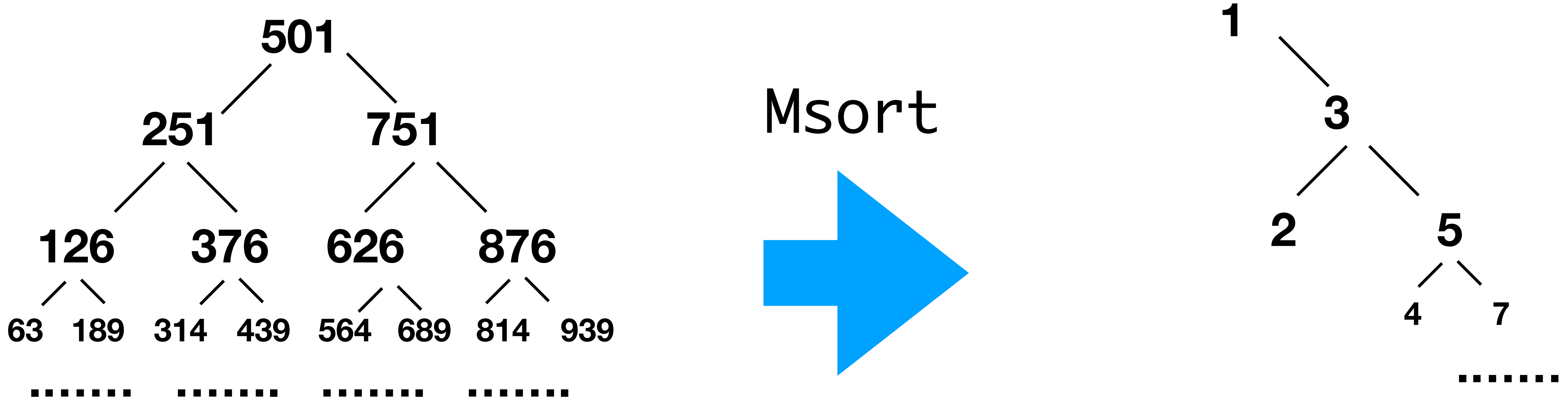
  See lecture notes
- The helper *specs* were chosen to permit an easy correctness proof for Msort.

demo

```
- val T = list2tree (upto 1 1000);
val T =
  Node
    (Node
       (Node
          (Node (Node (#,#,#),63,Node (#,#,#)),126,
           Node (Node (#,#,#),189,Node (#,#,#))),251,
        Node
          (Node (Node (#,#,#),314,Node (#,#,#)),376,
           Node (Node (#,#,#),439,Node (#,#,#)))),501,
     Node
       (Node
          (Node (Node (#,#,#),564,Node (#,#,#)),626,
           Node (Node (#,#,#),689,Node (#,#,#))),751,
        Node
          (Node (Node (#,#,#),814,Node (#,#,#)),876,
           Node (Node (#,#,#),939,Node (#,#,#))))) : int tree
```

```
- Msort T;
val it =
  Node
    (Empty,1,
     Node
       (Node (Empty,2,Empty),3,
        Node (Node (Empty,4,Empty),5,Node (Node (#,#,#),7,Node (#,#,#)))))
  : int tree

- inord it;
val it = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,...] : int list
```



taking stock

- We’ve defined a function Msort : int tree -> int tree
- It satisfies the *sorting* spec
- It runs pretty fast even on a tree of size 100000
- But it doesn’t always return a ***balanced*** sorted tree

efficiency?

- We know how to msort a ***list*** of n items in **O(n log n)** time
- A ***balanced tree*** of n items has size **n**, depth **O(log n)**
- **The million dollar question**:
  how efficient is Msort T
  when T is a balanced tree of size n?

The Span is…?

What's the span of **Msort T ?**

when T is balanced, depth d

Span of Ins

```
fun Ins (x, Empty) = Node(Empty, x, Empty)
|    Ins (x, Node(T1, y, T2)) =
   if x > y then Node(T1, y, Ins(x, T2))
            else  Node(Ins(x, T1), y, T2)
```

(no way to parallelize!)

For a balanced tree of depth d>0,

$$S_{Ins}(d) = 1 + S_{Ins}(d-1)$$

$S_{Ins}(d)$ is $O(d)$

Span of SplitAt

(similarly)

For a balanced tree of depth d>0,

$$S_{SplitAt}(d) = 1 + S_{SplitAt}(d-1)$$

$S_{SplitAt}(d)$ is $O(d)$

Span of Merge

independent

```
fun Merge (Empty, T2) = T2
  | Merge (Node(l1, x, r1), T2) =
      let val (l2, r2) = SplitAt(x, T2) in Node(Merge(l1, l2), x, Merge(r1, r2)) end
```

For balanced trees of depth d>0,

assuming the trees got by splitting have depth ≤ d-1, we get

$$S_{Merge}(d) = S_{SplitAt}(d) + \max(S_{Merge}(d-1), S_{Merge}(d-1))$$

$$= S_{SplitAt}(d) + S_{Merge}(d-1)$$

$$= O(d) + S_{Merge}(d-1)$$

$S_{Merge}(d)$ is $O(d^2)$

Span of Msort

```
fun Msort Empty = Empty
|   Msort (Node(T1, x, T2)) =
        Ins (x, Merge(Msort T1, Msort T2))
```

independent

For a balanced tree of depth $d > 0$

$$S_{Msort}(d) = \max(S_{Msort}(d-1), S_{Msort}(d-1)) + S_{Merge}(d) + S_{Ins}(2d)$$

$$= S_{Msort}(d-1) + O(d^2)$$

REALLY???

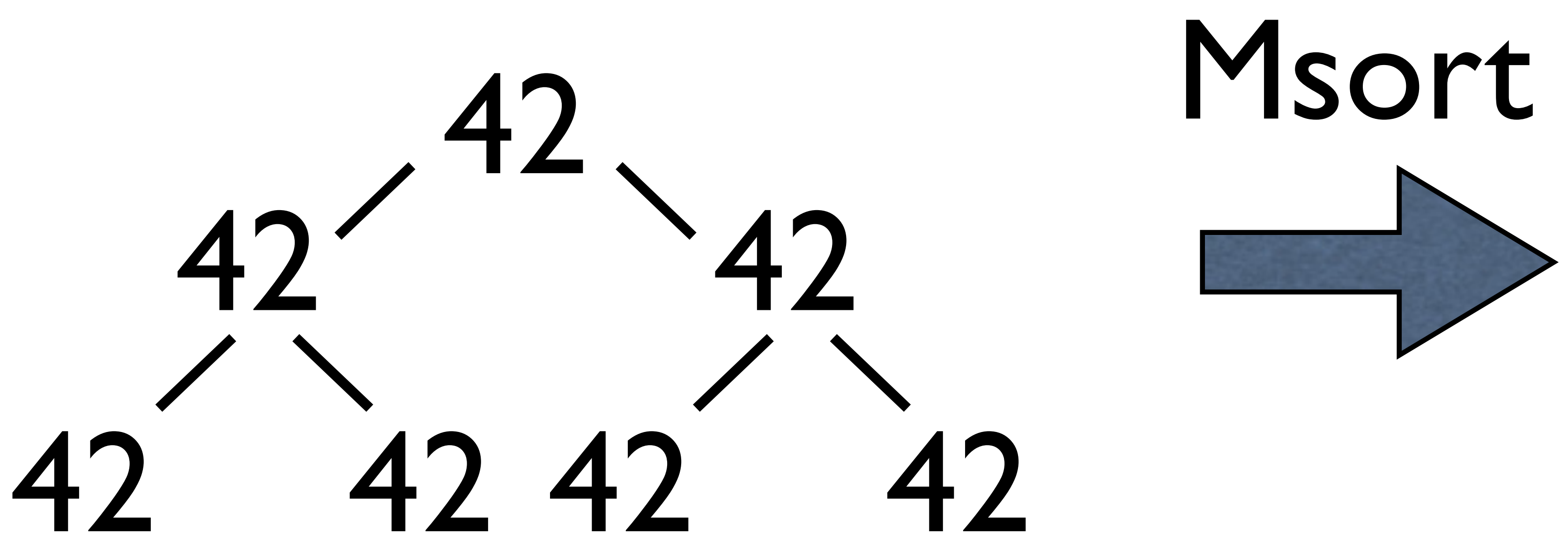
$S_{Msort}(d)$ is $O(d^3)$

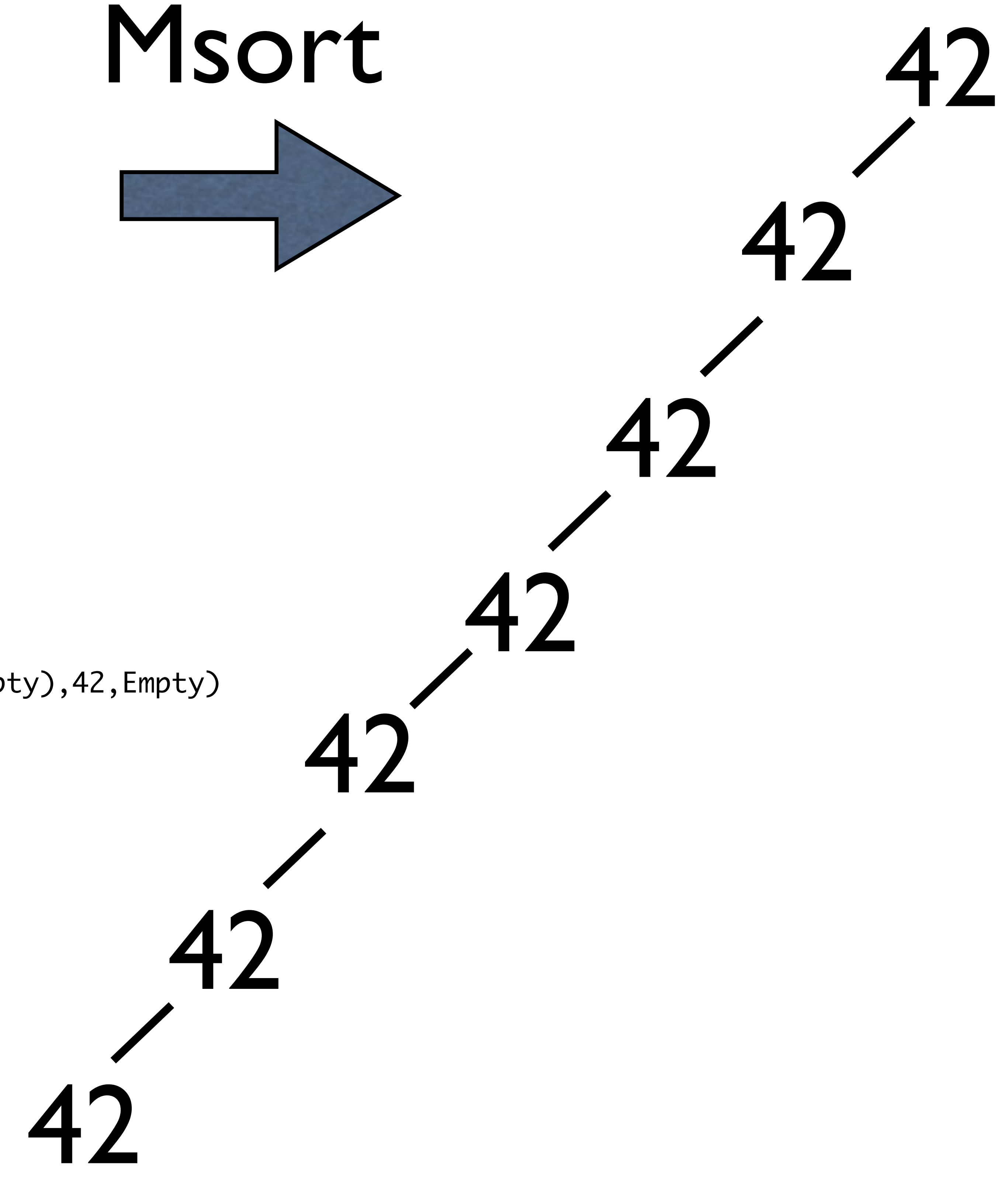
oops

- We assumed that splitting, merging, inserting with ***balanced trees*** produces ***balanced trees***
- That's **NOT** true!

losing balance

Msort can produce badly *unbalanced* trees






```
- Msort (Full(42, 3));
val it =
  Node (Node (Node (Node (Node (#,#,#),42,Empty),42,Empty),42,Empty),42,Empty)
   : int tree

- layers it;
val it = [[42],[42],[42],[42],[42],[42],[42]]
   : int list list
```

results

- Using Msort may produce a *poorly balanced* tree
- Its worst-case **work** is no better than that of msort on lists
- In “average” cases the tree-based method may be faster
- But we can make no promises :-)

towards a solution

- *Merge*, *Ins* don't preserve balance!
- We ***could*** use a tree *balancing* function...

```
fun Msort Empty = Empty
 |  Msort (Node(t1, x, t2)) =
       balance(Ins (x, Merge(Msort t1, Msort t2)))
```

- Or new versions of *Ins* and *Merge* that actually *preserve balance*

But perfect balance is hard to achieve...
and there are other solutions...

balanced vs **sorted**

- **Msort** produces a sorted tree
- Maintaining balance (along with sortedness) is a lot of extra work!
- Later we will see how to build *nearly-balanced* sorted trees…
- …with the same asymptotic behavior as *perfectly-balanced* sorted trees

lesson

- Datatypes allow us to design our own *types*
- *Structural induction* allows us to *define* functions, identify sets of values with special properties, and *reason about* program behavior
- Work and span recurrences are good for estimating how efficient our code is, asymptotically
- But be careful to do proofs and analysis *accurately*!
    - Be aware of any assumptions you make