Persistent data structures, sixième cours

From formal derivatives to navigation in a structure: contexts, zippers, fingers

Xavier Leroy

2023-04-13

Collège de France, chair of Software sciences
xavier.leroy@college-de-france.fr

Prologue:
structures annotated with a monoid

Binary search trees annotated by their sizes

The 2nd lecture mentioned the possibility of annotating the nodes of a BST by the sizes of the corresponding subtrees. This can be used

- to balance the trees (weight balancing);
- to determine quickly the size of a tree (time $\mathcal{O}(1)$ instead of $\mathcal{O}(n)$);
- to access tree elements by rank.

Determining the rank of an element

If the BST has size n and elements $x_0 < \cdots < x_{n-1}$, the rank of element x_i is the integer i.

```
type 'a tree = Leaf | Node of int * 'a tree * 'a * 'a tree

let size = function Leaf -> 0 | Node(s, _, _, _) -> s
let node l x r = Node(size l + 1 + size r, l, x, r)

let rec rank x t =
  match t with
  | Leaf -> raise Not_found
  | Node(l, y, r) ->
      if x < y then rank x l
      else if x = y then size l
      else size l + 1 + rank x r
```

Finding an element by its rank

The converse operation: given a rank i, find the element x_i.

```
let rec get i t =
  match t with
  | Leaf -> raise Out_of_bounds
  | Node(l, x, r) ->
      if i = size l then x
      else if i < size l then get i l
                         else get (i - size l - 1) r
```

Generalization to other annotations

An annotation of a binary tree $=$ a measure of its elements ranging over a monoid.

A monoid = a type μ equipped with a neutral element $\mathbf{0} : \mu$ and an associative operation $\oplus : \mu \to \mu \to \mu$.

$$x \oplus \mathbf{0} = \mathbf{0} \oplus x = x \qquad (x \oplus y) \oplus z = x \oplus (y \oplus z)$$

A measure: a function $\|\cdot\| : \alpha \to \mu$ that we extend into a function $\|\cdot\| : \alpha\ \texttt{tree} \to \mu$ by defining

$$\|\texttt{Leaf}\| = \mathbf{0}$$
$$\|\texttt{Node}(\ell, x, r)\| = \|\ell\| \oplus \|x\| \oplus \|r\|$$

Examples of monoids and measures

Measuring the size:

$$\mu = \texttt{int} \qquad \mathbf{0} = 0 \qquad x \oplus y = x + y \qquad \|x\| = 1$$

Measuring the sum of elements (for a tree of numbers):

$$\mu = \texttt{int} \qquad \mathbf{0} = 0 \qquad x \oplus y = x + y \qquad \|x\| = x$$

Measuring the variation interval of elements:

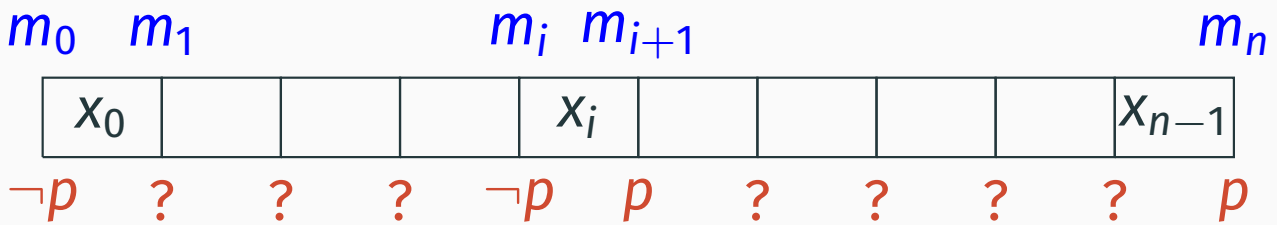
$$\mu = (\alpha \times \alpha)\ \texttt{option} \qquad \mathbf{0} = \texttt{None} \qquad \|x\| = \texttt{Some}(x, x)$$

$$\texttt{None} \oplus m = m \oplus \texttt{None} = m$$

$$\texttt{Some}(\ell_1, h_1) \oplus \texttt{Some}(\ell_2, h_2) = \texttt{Some}(\min(\ell_1, \ell_2), \max(h_1, h_2))$$

Scanning and splitting (Hinze et Paterson, 2006)

Scanning a sequence $t = x_0, \ldots, x_{n-1}$ from m_0:
compute the measures $m_1 = m_0 \oplus \|x_0\|, \ldots, m_{i+1} = m_i \oplus \|x_i\|$
until we reach an element x_i that brings a predicate $p : \mu \to \texttt{bool}$
from `false` to `true`.



Such an x_i always exists if $p(m_0) = \texttt{false}$ and
$p(m_0 \oplus \|t\|) = \texttt{true}$. It is not unique in general.

Splitting a sequence: scan, then return (ℓ, x_i, r)
where ℓ is the sequence of elements preceding x_i
and r the sequence of elements following x_i.

An OCaml implementation using functors

```
module type MONOID = sig
  type t
  val zero: t
  val add: t -> t -> t
end

module type ORDERED_MEASURED = sig
  type t
  val compare: t -> t -> int
  module M: MONOID
  val measure: t -> M.t
end

module BST(X: ORDERED_MEASURED) :
                    ORDERED_MEASURED with module M = X.M
= struct module M = X.M ... end
```

An OCaml implementation using functors

```
module M = X.M
type t = Leaf | Node of M.t * t * X.t * t
let measure t = match t with Leaf -> M.zero | Node(m, _, _, _) -> m
let node l x r =
  Node(M.add (measure l) (M.add (X.measure x) (measure r)), l, x, r)
let rec split p m t =
  match t with
  | Leaf -> raise Not_found
  | Node(_, l, x, r) ->
      let m1 = M.add m (measure l) in
      let m2 = M.add m1 (X.measure x) in
      if p m1 then
        let (l', x', r') = split p m l in
        (l', x', node r' x r)
      else if p m2 then (l, x, r)
      else
        let (l', x', r') = split p m2 r in
        (node l x l', x', r')
```

Application to finger trees: random access

Hinze and Paterson (2006) show how to annotate finger trees with measures and to implement a `split` operation that runs in time $\mathcal{O}(\log n)$.

Using the monoid of sizes, we get random access to the i-th element of the sequence, in time $\mathcal{O}(\log n)$.

```
let get i s =
  let (_, x, _) = split (fun sz -> sz > i) 0 t in x
let set i v s =
  let (l, _, r) = split (fun sz -> sz > i) 0 t in
  (concat l (cons v r))
let delete i s =
  let (l, _, r) = split (fun sz -> sz > i) 0 t in
  concat l r
```

Application to finger trees: priority queues

Using the monoid of intervals, we get a min-max priority queue:

- access to the smallest / largest element in time $\mathcal{O}(\log n)$;
- insertion in time $\mathcal{O}(1)$ amortized.

```
let extract_min s =
  match measure s with
  | None -> raise Empty
  | Some(min, max) ->
      let (l, _, r) =
        split (function None -> false | Some(m, _) -> m = min)
              None t
      in (min, concat l r)
```

Application to finger trees: ordered sequence

We can also used the last value monoid:

$$\mu = \alpha \ \texttt{option} \qquad \mathbf{0} = \texttt{None} \qquad \|x\| = \texttt{Some}(x)$$

$$\texttt{None} \oplus m = m \oplus \texttt{None} = m$$

$$\texttt{Some}(v_1) \oplus \texttt{Some}(v_2) = \texttt{Some}(v_2)$$

If we keep the sequence sorted in increasing order, these “last value” annotations support binary search like in a BST.

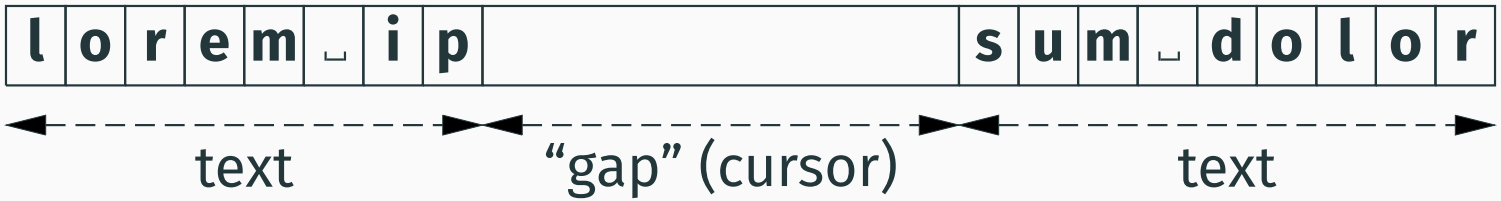
`split` supports implementing insertion, deletion, and search in time $\mathcal{O}(\log n)$.

`head`, `tail`, `last`, `take` give us access to the smallest / largest element in time $\mathcal{O}(1)$ amortized.

Navigation in a structure

Gap buffer

A data structure used by text editors (e.g. Emacs).



An array of characters, larger than the text being edited.

The text is stored in two contiguous fragments, one at the beginning of the array, the other at the end.

The gap (the free space in between) is the editing cursor.

Text editing operations

Loading the text

abcde

Text editing operations

Loading the text

abcde

Move back 3 characters

ab|cde

Text editing operations

Loading the text

abcde|

Move back 3 characters

ab|cde

Delete before cursor

a|cde

Text editing operations

Loading the text

abcde|

Move back 3 characters

ab|cde

Delete before cursor

a|cde

Insert “x”, then “y”

axy|cde

Text editing operations

Loading the text		abcde|
Move back 3 characters		ab|cde
Delete before cursor		a|cde
Insert “x”, then “y”		axy|cde
Delete after cursor		axy|de

Text editing operations

Loading the text		abcde|
Move back 3 characters		ab|cde
Delete before cursor		a|cde
Insert “x”, then “y”		axy|cde
Delete after cursor		axy|de
Saut au début du texte		|axyde

Cost of editing operations

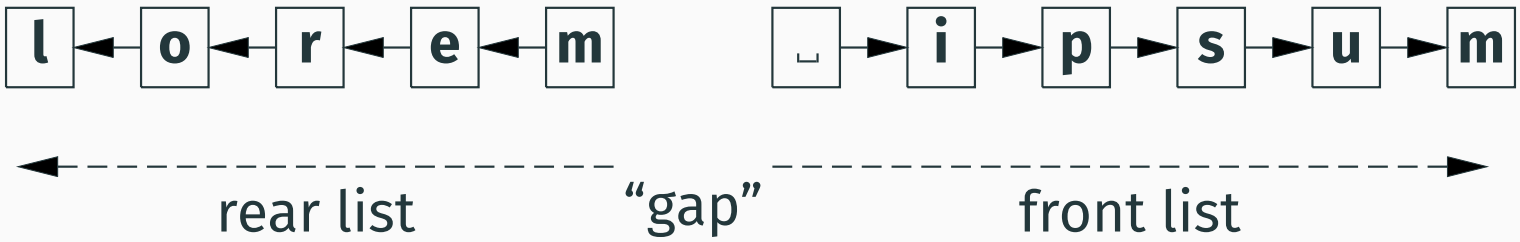
The cost of an operation is proportional to the distance between the cursor and the point where the operation takes place.

Insertion, deletion	$\mathcal{O}(1)$
Moving forward / backward d characters	$\mathcal{O}(d)$
Jump to beginning / end of text	$\mathcal{O}(n)$

(Plus: resizing the array when needed, which takes constant amortized time.)

A singly-linked list with a gap

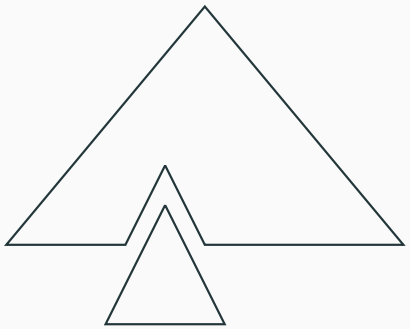
A “front” list and a “rear” list. The cursor sits between the two.

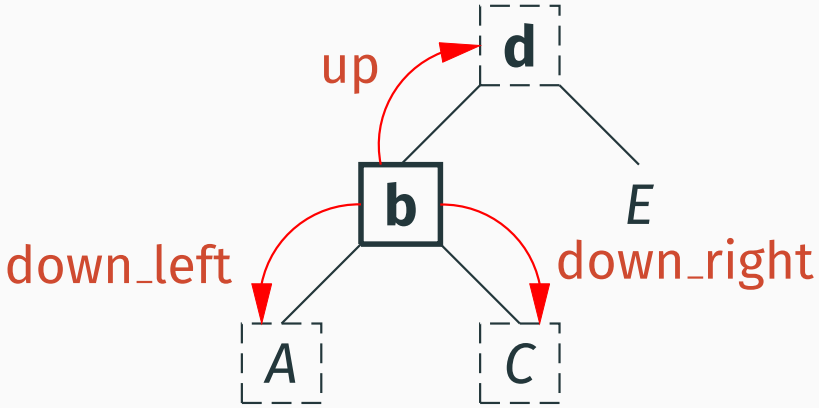


Navigation in a gap list

```
type 'a hlist = 'a list * 'a list

let insert x (r, f) = (x :: r, f)

let delete_after (r, f) =
  match f with [] -> raise Error | _ :: f' -> (r, f')
let delete_before (r, f) =
  match r with [] -> raise Error | _ :: r' -> (r', f)

let move_forward (r, f) =
  match f with [] -> (r, f) | x :: f' -> (r :: x, f')
let move_backward (r, f) =
  match r with [] -> (r, f) | x :: r' -> (r', x :: f)

let move_to_beginning (r, f) = ([], List.rev_append r f)
let move_to_end (r, f) = (List.rev_append f r, [])
```

Navigation in a tree





We'd like to move not just from leaf to leaf, but more generally from subtree to subtree.

Hence a representation as a pair of

a subtree (the cursor)
+ a tree with a "hole" (the remainder of the tree).

A tree with a hole = a context?

In operational semantics and other areas that use term algebras, we use contexts C as "a term with a hole" (written $[\,]$).

For example, in the case of binary trees

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

here is the type of contexts ($=$ trees with a hole):

```
type 'a context =
  | Hole
  | Node_left of 'a context * 'a * 'a tree
  | Node_right of 'a tree * 'a * 'a context
```

Operations over contexts

The main operation, written $C[t]$, rebuilds a term by filling the hole [] in C with the term t.

```
let rec fill_hole c t =
  match c with
  | Hole -> t
  | Node_left(c', x, r) -> Node(fill_hole c' t, x, r)
  | Node_right(l, x, c') -> Node(l, x, fill_hole c' t)
```

Navigation using contexts

Navigating with pairs (context, subtree) is possible, but moves take non-constant time!

```
let rec down_left (c, t) =
  match (c, t) with
  | (Hole, Node(l, x, r)) -> (Node_left(Hole, x, r), l)
  | (Hole, Leaf) -> raise Error
  | (Node_left(c, x, r), t) ->
      let (c', t') = down_left (c, t) in
      (Node_left(c', x, r), t')
  | (Node_right(l, x, c), t) ->
      let (c', t') = down_left (c, t) in
      (Node_right(l, x, c'), t')
```

Navigation using contexts

We would have hit the same inefficiency in the example of gap lists if we represented the rear list “in the wrong direction”:

✔ 

✘ 

Wanted: a “reversed” tree context, with the hole at the top…

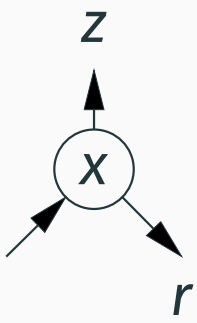
Zippers (G. Huet, 1997)

A reversed representation of contexts, where the first constructor of the zipper is next to the hole, and the last constructor (`Top`) denotes the root of the tree.

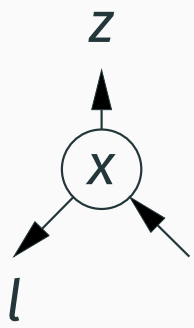
For binary trees:

```
type 'a zipper =
  | Top
  | Left_of of 'a * 'a tree * 'a zipper
  | Right_of of 'a tree * 'a * 'a zipper
```

`Left_of`(x, r, z):



`Right_of`(l, x, z):

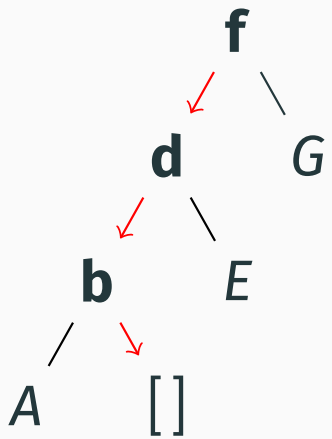


Zipper vs. context

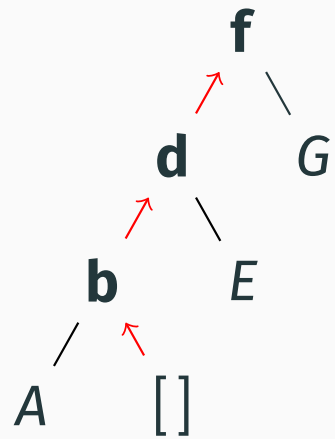
In a context, constructors go from top to bottom.

In a zipper, constructors go from bottom to top.

Context:



Zipper:



Context:

$$\texttt{Node\_left}(\texttt{Node\_left}(\texttt{Node\_right}(A, \mathbf{b}, \texttt{Hole}), \mathbf{d}, E), \mathbf{f}, G)$$

Zipper:

$$\texttt{Right\_of}(A, \mathbf{b}, \texttt{Left\_of}(\mathbf{d}, E, \texttt{Left\_of}(\mathbf{f}, G, \texttt{Top})))$$

Operations over zippers

The main operation, app *z t*, rebuilds a term by wrapping the zipper *z* around the term *t*.

```
let rec app z t =
  match z with
  | Top -> t
  | Left_of(x, r, z') -> app z' (Node(t, x, r))
  | Right_of(l, x, z') -> app z' (Node(l, x, t))
```

(Note: tail recursive!)

Navigation with zippers

All three basic moves take constant time!

```
let down_left (t, z) =
  match t with
  | Leaf -> raise Error
  | Node(l, x, r) -> (l, Left_of(x, r, z))

let down_right (t, z) =
  match t with
  | Leaf -> raise Error
  | Node(l, x, r) -> (r, Right_of(l, x, z))

let up (t, z) =
  match z with
  | Top -> raise Error
  | Left_of(x, r, z') -> (Node(t, x, r), z')
  | Right_of(l, x, z') -> (Node(l, x, t), z')
```

Binary search with zippers

We can view binary search in a BST as returning the place where the desired value x is or should be.

```
let rec search x (t, z) =
  match t with
  | Leaf -> (t, z)
  | Node(l, y, r) ->
      if y = x then (t, z) else
      if y < x then search x (l, Left_of(y, r, z))
               else search x (r, Right_of(l, y, z))
```

This way, we can move from any point to point x :

```
let move_to x (t,z) = search x (app z t, Top)
```

Time: $\mathcal{O}(\log n)$. (Can be improved, see later.)

Application: a functional presentation of splay trees

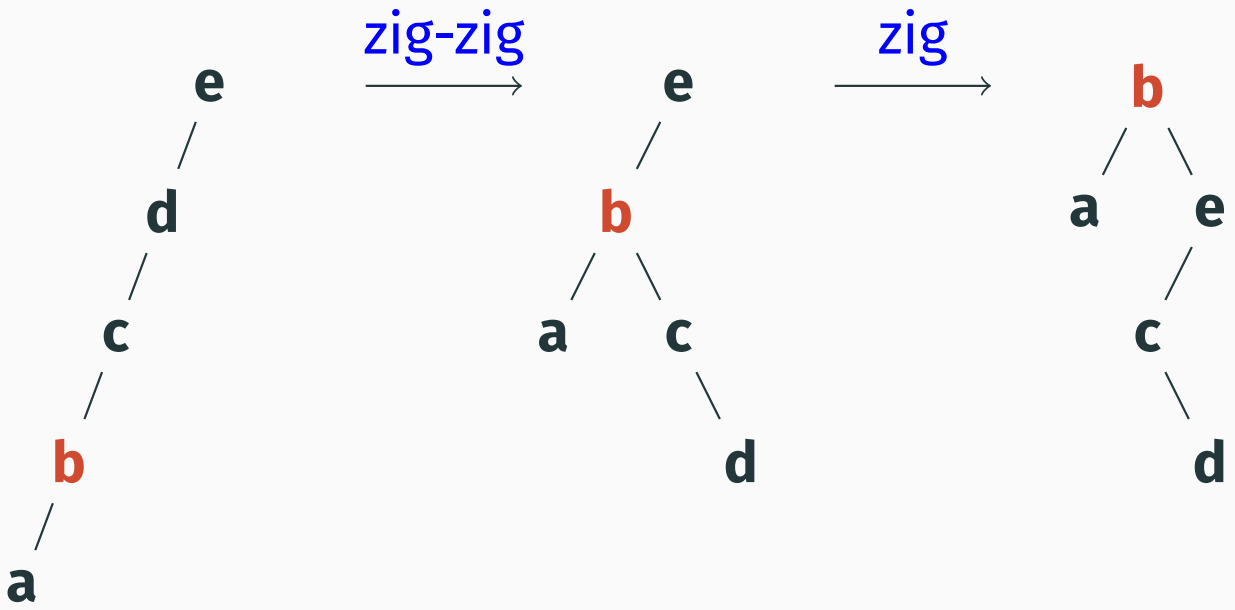
Splay trees (Sleator and Tarjan 1985):

- Binary search trees without explicit balancing.
- At each operation (search, insertion, deletion), the x element involved is moved to the top of the tree using well-chosen rotations.
- These rotations progressively reduce tree imbalance, leading to operations that run in time $\mathcal{O}(\log n)$ amortized and $\mathcal{O}(n)$ worst-case.

Example of search in a splay tree

We search for **b** in the heavily left-leaning tree **abcde**.

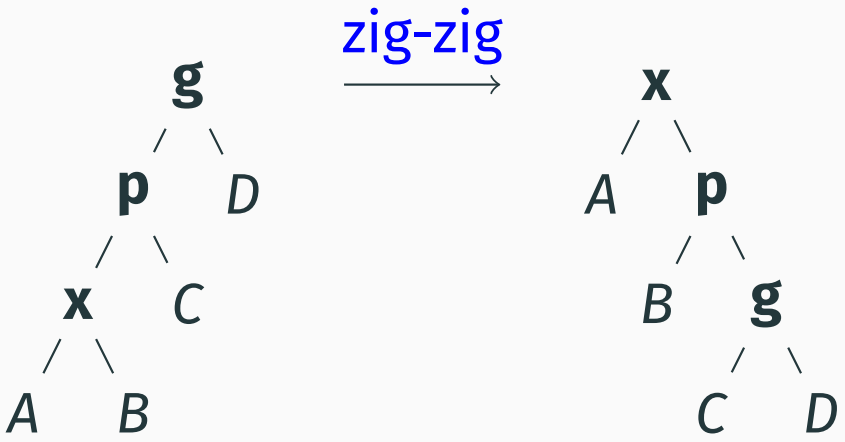
We move **b** to the top by two rotations, *zig-zig* then *zig*.

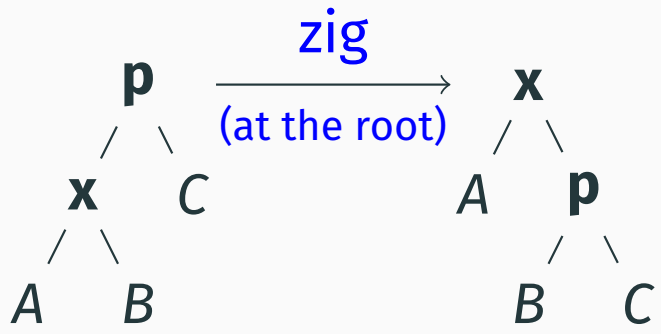


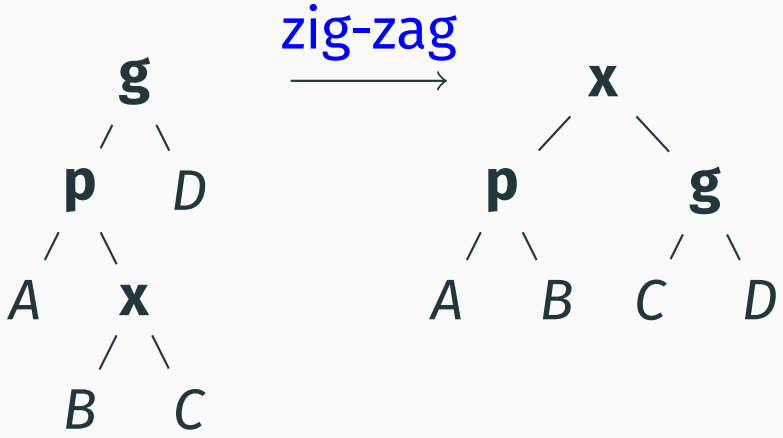
Rotations for splay trees

To “hoist” node **x** above its parent **p** and its grandparent **g**.

(Omitted: 3 symmetrical rotations.)







Splaying using a zipper

`splay l x r z` returns a BST equivalent to `app z (Node(l, x, r))` but having `x` at the top.

```
let rec splay l x r z =
  match z with
  | Top -> Node(l, x, r)
  | Left_of(p, c, Top) -> (* final zig  *)
      Node(l, x, Node(r, p, c))
  | Right_of(c, p, Top) -> (* final zig  *)
      Node(Node(c, p, l), x, r)
  | Left_of(p, c, Left_of(q, d, z)) -> (* zig-zig *)
      splay l x (Node(r, p, Node(c, q, d))) z
  | Right_of(c, p, Right_of(d, q, z)) -> (* zig-zig *)
      splay (Node(Node(d, q, c), p, l)) x r z
  | Right_of(a, p, (Left_of(q, d, z))) -> (* zig-zag *)
      splay (Node(p, a, l)) x (Node(r, q, d)) z
  | Left_of(p, a, Right_of(d, q, z)) -> (* zig-zag *)
      splay (Node(d, q, l)) x (Node(p, r, a)) z
```

Insertion in a splay tree

Insertion = search

+ creating a new node (if needed)

+ splaying.

```
let add x t =
  match search x (t, Top) with
  | (Leaf, z') -> splay Leaf x Leaf z'     (* not found *)
  | (Node(l, _, r), z') -> splay l x r z' (* found *)
```

(For a purely-functional presentation of splay trees without zippers, see Nipkow et al, *FAV!*, chap. 21.)

Connections with formal derivatives

Contexts ≈ zippers ≈ lists of deltas

For a regular algebraic type, the type of zippers and the type of contexts are isomorphic to a list of deltas:

```
type 'a delta =
  | Left of 'a * 'a tree | Right of 'a tree * 'a

type 'a context =
  | Hole                                              (* = [] *)
  | Node_left of 'a context * 'a * 'a tree  (* = Left (x, r) :: c *)
  | Node_right of 'a tree * 'a * 'a context (* = Right(l,x) :: c *)

type 'a zipper =
  | Top                                             (* = [] *)
  | Left_of of 'a * 'a tree * 'a zipper   (* = Left (x, r) :: z *)
  | Right_of of 'a tree * 'a * 'a zipper  (* = Right (l, x) :: z *)
```

Applying deltas in the right order

```
let app d t =
  match d with Left(l, x) -> Node(l, x, t)
             | Right(x, r) -> Node(t, x, r)

let rec fill_hole c t =
  match c with [] -> t | d :: c -> app d (fill_hole c t)
let rec app_zipper z t =
  match z with [] -> t | d :: z -> app_zipper z (app d t)
```

For a context:

first delta = top of the tree; empty list = the hole

filling a context $C[t]$ = a right fold of `app`.

For a zipper:

first delta = neighbor of the hole; empty list = the top

applying a zipper = a left fold of `app`.

Constructing the type of deltas from the data type

```
type 'a tree =
  | Leaf
  | Node of
      'a tree * 'a * 'a tree
```

```
type 'a delta =

  | Left of 'a * 'a tree
  | Right of 'a tree * 'a
```

A constructor without any recursive occurrence of the type disappears.

A constructor with n arguments including k recursive arguments becomes k constructors with $n - 1$ arguments, obtained by removing one of the k recursive occurrences.

```
'a tree * 'a * 'a tree  ⇒  ~~'a tree *~~ 'a * 'a tree
                           'a tree * 'a ~~* 'a tree~~
```

An algebra of types

Types:	$\tau ::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2}$	`empty`, `unit`, `bool`
	$\mid t \mid \alpha \mid \beta$	type variables
	$\mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2$	sum and product

Isomorphisms: $+$ and $\times$ are commutative and associative; moreover,

$$\mathbf{0} + \tau \equiv \tau \qquad \mathbf{0} \times \tau \equiv \mathbf{0} \qquad \mathbf{1} \times \tau \equiv \tau \qquad \mathbf{2} \times \tau \equiv \tau + \tau$$

Fixed points of types

Types:	$\tau ::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2}$	`empty`, `unit`, `bool`
	$\mid t \mid \alpha \mid \beta$	type variables
	$\mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2$	sum and product

A regular algebraic datatype `type` $t = \tau$ is a fixed point $\mu t.\tau$.

Examples:

$\mu t.\ \mathbf{1} + t$	Peano numbers (`Zero`/`Succ`)
$\mu t.\ \mathbf{1} + \alpha \times t$	lists of α (`Nil`/`Cons`)
$\mu t.\ \alpha + t \times t$	binary trees with α at leaves
$\mu t.\ \mathbf{1} + t \times \alpha \times t$	binary trees with α at nodes
$\mu t.\ \alpha + t \times t + t \times t \times t$	2-3 trees with α at leaves

The type of deltas

(C. McBride, *The derivative of a regular type is its type of one-hole contexts* 2001)

The type of deltas for the regular algebraic datatype $\mu t.\tau$ is

$$(\partial_t \tau)\,[t \leftarrow \mu t.\tau]$$

that is, the formal derivative of τ with respect to the variable t taken at point $\mu t.\tau$.

$$\partial_t \mathbf{0} = \partial_t \mathbf{1} = \partial_t \mathbf{2} = \mathbf{0}$$

$$\partial_t t = \mathbf{1}$$

$$\partial_t \alpha = \mathbf{0} \quad \text{if } \alpha \neq t$$

$$\partial_t (\tau_1 + \tau_2) = \partial_t \tau_1 + \partial_t \tau_2$$

$$\partial_t (\tau_1 \times \tau_2) = \partial_t \tau_1 \times \tau_2 + \tau_1 \times \partial_t \tau_2$$

Examples of type derivatives

$$\partial_t \left(\mathbf{1} + t\right) = \mathbf{0} + \mathbf{1} \equiv \mathbf{1}$$

The type of deltas for Peano numbers is `unit`.

$$\partial_t \left(\mathbf{1} + \alpha \times t\right) = \mathbf{0} + \mathbf{0} \times t + \alpha \times \mathbf{1} \equiv \alpha$$

The type of deltas for lists of α is α.

$$\partial_t \left(\alpha + t \times t\right) = \mathbf{0} + \mathbf{1} \times t + t \times \mathbf{1} \equiv t + t \equiv \mathbf{2} \times t$$

The type of deltas for binary trees with α at leaves is “a tree or a tree”, or, equivalently, “a Boolean and a tree”.

$$\partial_t \left(\mathbf{1} + t \times \alpha \times t\right) \equiv \alpha \times t + t \times \alpha \equiv \mathbf{2} \times \alpha \times t$$

The type of deltas for binary trees with α at nodes is “an α and a tree, or a tree and an α”.

Further reading

Extension to non-regular data types and to GADTs:

- C. McBride. *The derivative of a regular type is its type of one-hole contexts*. 2001.

A formalization of the connection between formal derivatives and contexts:

- M. Abbot, T. Altenkirch, N. Ghani et C. McBride.
  ∂ for data: differentiating data structures.
  Fundamenta Informaticae, 2005.

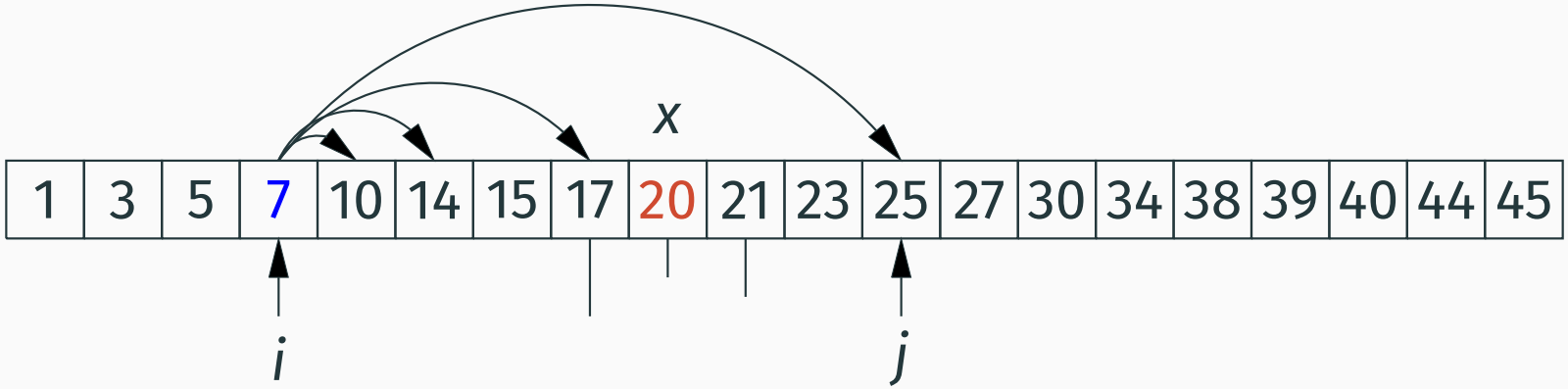
Fingers

Fingers (Guibas, McCreight, Plass, Roberts, 1977)

A finger $=$ a "pointer" to an element x of a data structure that supports faster operations over elements near x.

Typically, if a regular operation takes time $\mathcal{O}(f(n))$,
where n is the size of the structure,
the finger-based operation takes time $\mathcal{O}(f(d))$,
where d is the distance from the finger.

Example: a finger in a sorted array



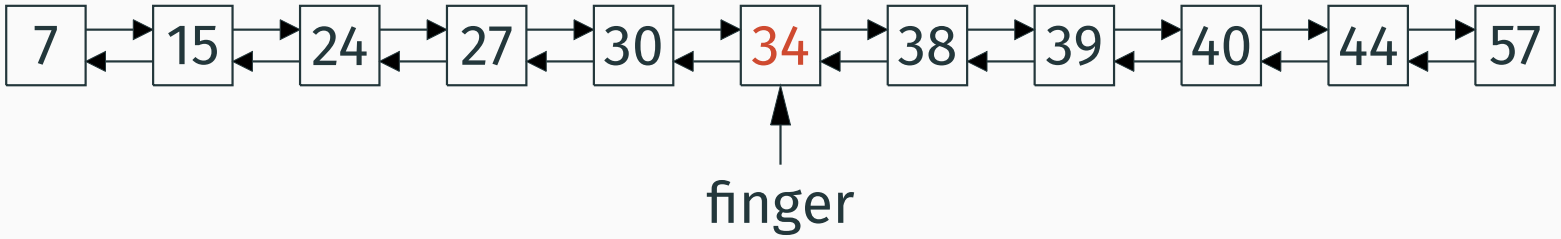
A finger on élément $x_i =$ its position i.

Searching for an element $x > x_i$ near x_i:

- Try $j = i + 1, i + 2, i + 4, i + 8, \ldots$ until $x \leq x_j$.
- Do a binary search between i and j.

Time: $\mathcal{O}(\log(j - i))$, that is, $\mathcal{O}(\log d)$ since $j - i \leq 2d$.

Example: a finger in a doubly-linked list



From a pointer to element x_i, we can

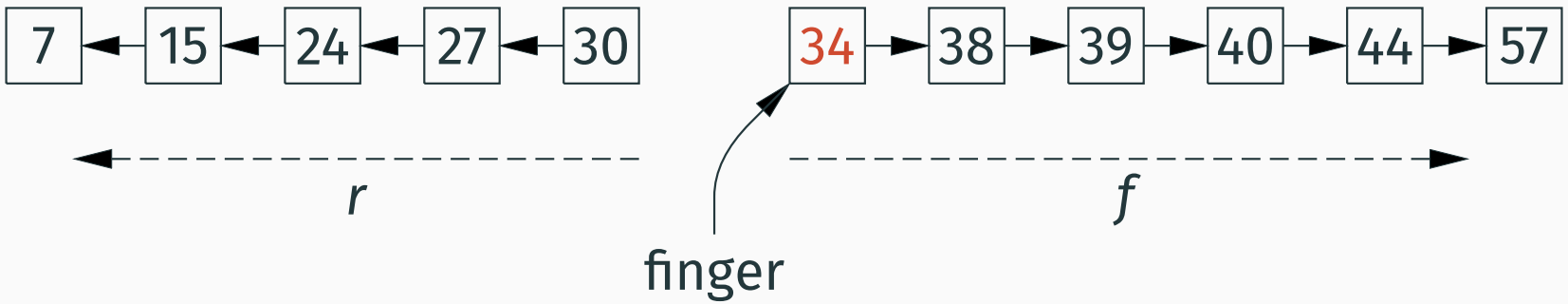
- delete or insert before or after this element in $\mathcal{O}(1)$ time;
- search for element x_j in time $\mathcal{O}(d)$ where $d = |j - i|$.

We can reduce search time to $\mathcal{O}(\log d)$ by replacing the list with a skip list or by a balanced tree (a B-tree in Guibas et al 1977).

Zippers as purely-functional fingers

If we have only one finger per data structure, it can often be represented as a (zipper, subtree) pair.

Example: a finger in a sorted list $=$ a gap list (r, f)
with r sorted in increasing order and f in decreasing order.



To search for an element x near the finger, we search in f if $f \neq [\,]$ and $x \geq \mathtt{hd}(f)$, or in r if $r \neq [\,]$ and $x \leq \mathtt{hd}(r)$.
Time $\mathcal{O}(d)$.

A finger in a binary search tree

A finger to a subtree of a BST can also be represented as a pair (t, z) where t is a subtree and z a tree zipper.

To search x near the finger, we need a fast (constant-time) test to determine whether we should search in t or "move up" z to widen the search.

Solution: annotate each subtree with the variation interval of values it contains, as seen at the beginning of the lecture.

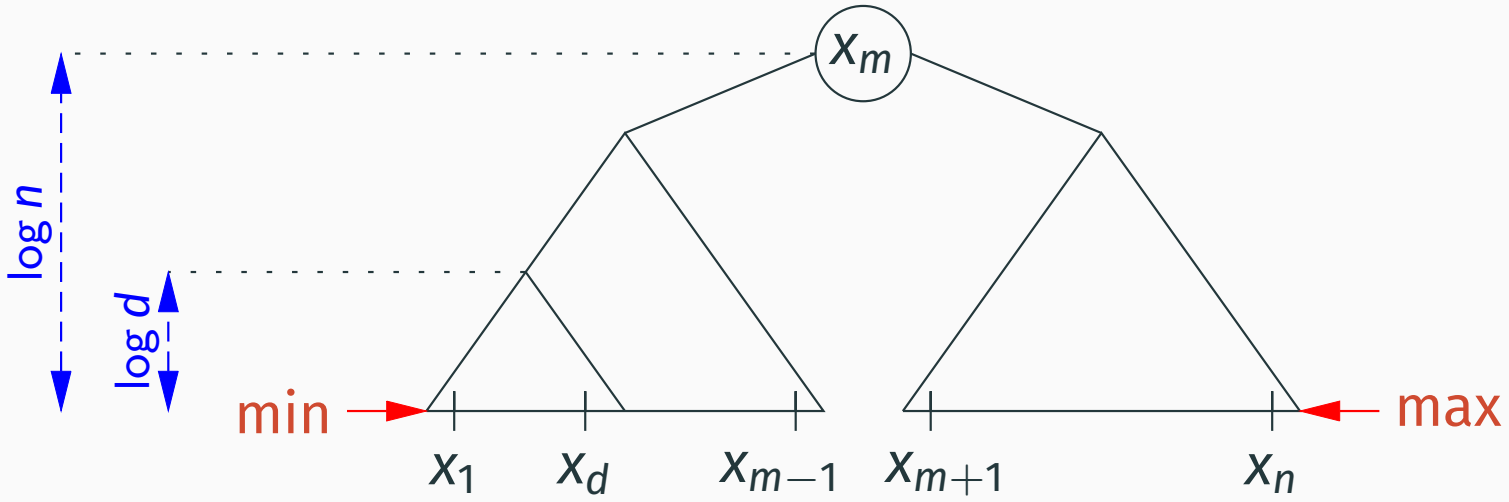
If x is in the interval of t, we must search in t (and nowhere else). Otherwise, we must widen the search.

Searching from a finger in a BST

```
let rec finger_search x (t, z) =
  if in_interval x (measure t)
  then search x (t, z)
  else
    match z with
    | Top -> search x (t, z) (* or: raise Not_found *)
    | Left_of(y, r, z) -> finger_search x (node t y r, z)
    | Right_of(l, y, z) -> finger_search x (node l y t, z)
```

Generally more efficient than `search x (app z t, Top)`
but not in $\mathcal{O}(\log d)$ worst-case time…

Searching from a finger in a BST



Best case: if (t, z) points to the smallest element x_1 (or the largest). Assuming the tree is balanced, the subtree containing both x_1 and x_d contains $\mathcal{O}(d)$ elements and has height $\mathcal{O}(\log d)$.

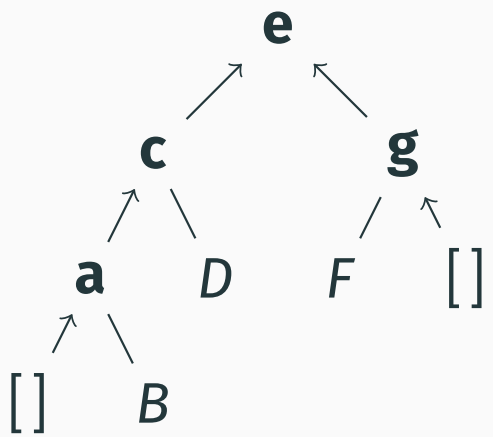
Worst case: the finger points to x_{m-1} (the largest element of the left subtree) and we search for x_{m+1}. The distance d is 2, but we must move up all the way to the root, in time $\mathcal{O}(\log n)$.

Two fingers in a structure

In general, zippers do not support multiple fingers in a structure.

An exception: two fingers in a BST, one to the smallest element, the other to the largest element.

We represent the leftmost branch and the rightmost branch by zippers, going from bottom to top.



Min finger and max finger in a BST

```
type 'a left_zipper = ('a * 'a tree) list
type 'a right_zipper = ('a tree * 'a) list

type 'a min_max =
  | Empty
  | Topnode of 'a left_zipper * 'a * 'a right_zipper

let rebuild (mm: 'a min_max) : 'a tree =
  match mm with
  | Empty -> Leaf
  | Topnode(lz, x, rz) ->
      Node(List.fold_left (fun l (x, r) -> Node(l, x, r)) Leaf lz,
           x,
           List.fold_left (fun r (l, x) -> Node(l, x, r)) Leaf rz)
```

Searching from a min finger or a max finger

Searching is easier than from an arbitrary finger: no need for variation intervals, and worst-case time $\mathcal{O}(\log d)$ where d is the distance to the min or to the max, whichever is smaller.

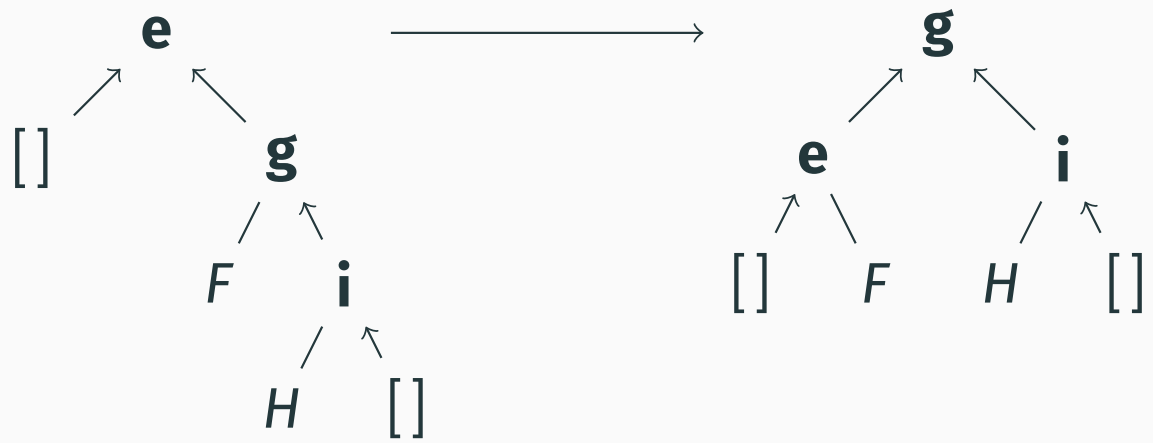
```
let rec mem_left v = function
  | [] -> false
  | [(x, r)] -> v = x || mem x r
  | (x1, r1) :: ((x2, _) :: _) as lz ->
      if v < x1 then false else
      if v = x1 then true else
      if v < x2 then mem v r1 else mem_left v lz

let rec mem_right v = function ...

let mem_min_max v = function
  | Leaf -> false
  | Topnode(lz, x, rz) ->
      v = x || (if x < v then mem_left v lz else mem_right v rz)
```

Rebalancing a BST with min and max fingers

Local rebalancing in the left zipper or the right zipper is relatively simple. Each rotation takes constant time.

Rotations that involve the top of the tree are more expensive (time $\log n$) because we need access to the last elements of the zippers.

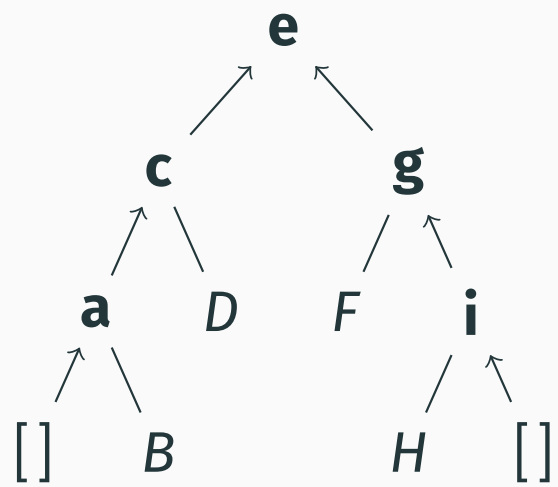
Example: rebalancing when the left zipper is empty.

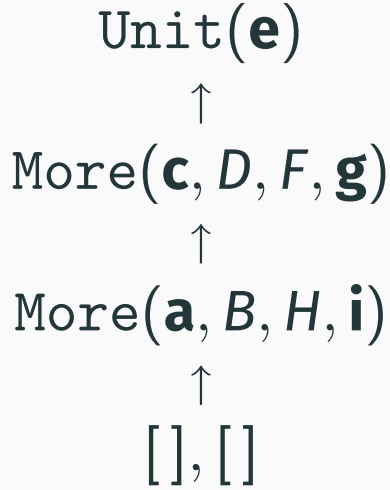


Min and max fingers on a perfect tree

If the binary tree is perfect, all branches have the same lengths, including the left zipper and the right zipper.

We can, therefore, fuse the two zippers in one!

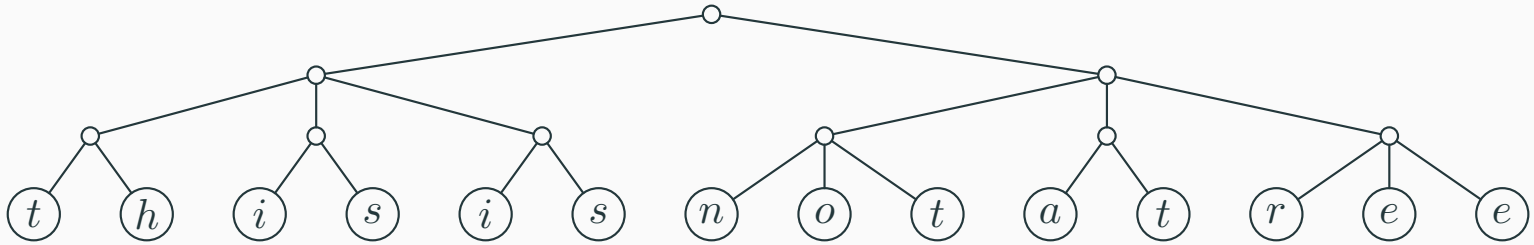




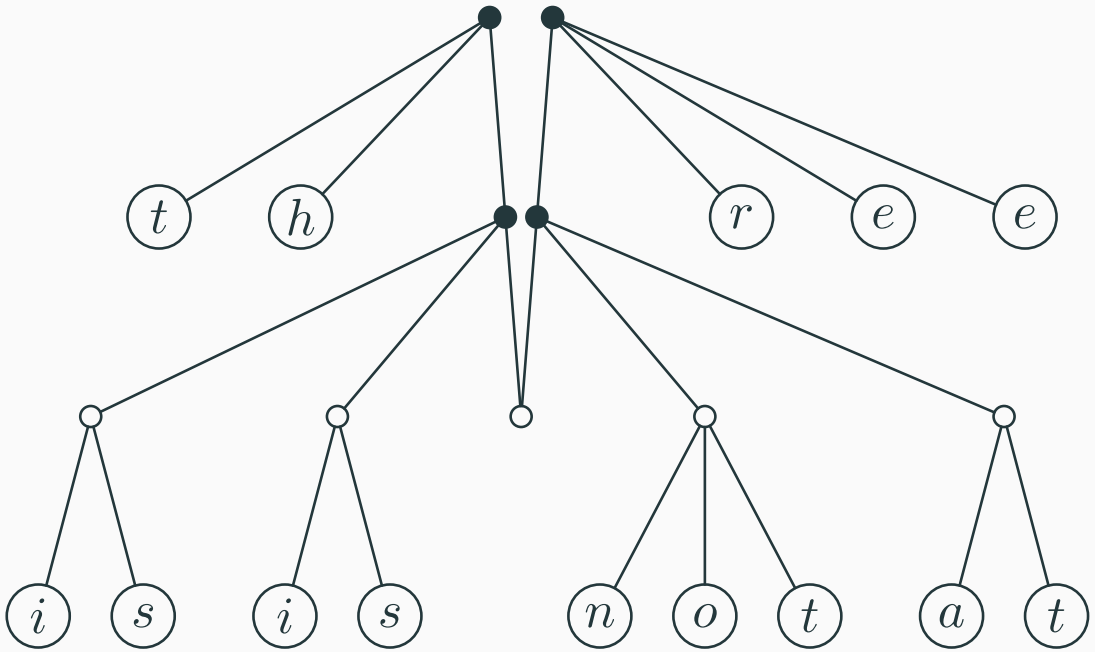

```
type 'a minmax =
  | Empty | Unit of 'a
  | More of 'a * 'a tree * 'a minmax * 'a tree * 'a
```

Finger trees: min and max fingers on a 2-3 tree

A 2-3 tree with values at leaves: (Hinze and Paterson, 2006)



The same, held by the leftmost and the rightmost nodes:



Finger trees: min and max fingers on a 2-3 tree

A non-regular type of 2-3 trees with values at leaves:

```
type 'a node = Pair of 'a * 'a | Triple of 'a * 'a * 'a
type 'a tree23 = Leaf of 'a | Node of 'a node tree23
```

Adding min and max fingers represented as a shared zipper:

```
type 'a digit =
  | One of 'a | Two of 'a * 'a | Three of 'a * 'a * 'a
type 'a seq =
  | Nil
  | Unit of 'a
  | More of 'a digit * 'a node seq * 'a digit
```

Summary

Summary

With the help of a bit of algebra, we can add new features to our data structures:

- Annotation by measures ranging over monoids
  $\rightarrow$ access by rank, by min or max value, …
- Formal derivatives to define zippers
  $\rightarrow$ navigation, fingers, …

References

References

The original article on zippers:

- Gérard P. Huet, *The Zipper*, J. Funct. Program. 7(5), 1997.

The original article on finger trees, which also introduces annotations using monoids:

- Ralf Hinze et Ross Paterson, *Finger trees: a simple general-purpose data structure*, J. Funct. Program. 16(2), 2006.