Structures de données persistantes, sixième cours

De la dérivation formelle à la navigation dans une structure: contextes, *zippers*, index

Xavier Leroy

2023-04-13

Collège de France, chaire de sciences du logiciel
xavier.leroy@college-de-france.fr

Prologue :
structures annotées par un monoïde

Arbres binaires de recherche annotés par leurs tailles

Au 2e cours nous avions mentionné la possibilité d'annoter les nœuds d'un arbre binaire de recherche par la taille du sous-arbre correspondant. Cela peut servir

- pour équilibrer les arbres ;
- pour déterminer rapidement la taille d'un arbre (en temps $\mathcal{O}(1)$ au lieu de $\mathcal{O}(n)$) ;
- pour accéder aux éléments de l'arbre par rang.

Déterminer le rang d'un élément

Si l'arbre est de taille n et ses éléments sont $x_0 < \cdots < x_{n-1}$, le rang de l'élément x_i est l'entier 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
```

Retrouver un élément par son rang

L'opération inverse : à partir de son rang i, retrouver l'élément 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
```

Extension à d'autres annotations

Une annotation d'un arbre binaire = une mesure de ses éléments à valeur dans un monoïde.

Un monoïde = un type μ équipé d'un élément neutre $\mathbf{0} : \mu$ et d'un opérateur associatif $\oplus : \mu \to \mu \to \mu$.

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

Une mesure : une fonction $\|\cdot\| : \alpha \to \mu$ que l'on étend de manière canonique en une fonction $\|\cdot\| : \alpha\ \texttt{tree} \to \mu$:

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

Exemples de monoïdes et de mesures

Mesurer la taille :

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

Mesurer la somme des éléments (pour un arbre de nombres) :

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

Mesurer l'intervalle de variations des valeurs :

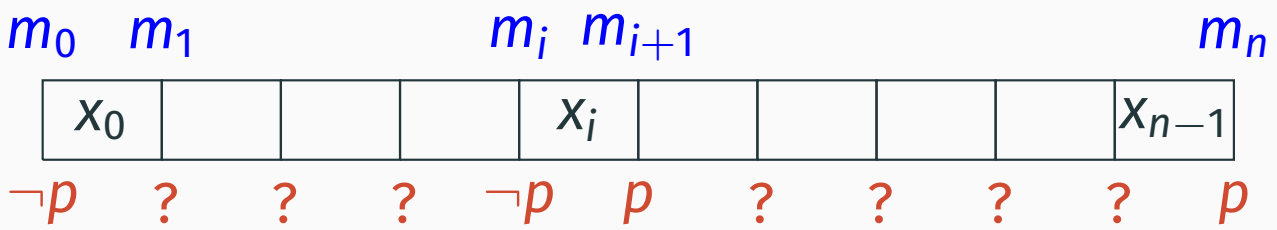
$$\mu = (\alpha \times \alpha)\ \texttt{option} \qquad \mathbf{0} = \texttt{None} \qquad \|x\| = \texttt{Some}(x, x)$$

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

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

Balayage (*scan*) et découpage (*split*) (Hinze et Paterson, 2006)

Balayer une séquence $t = x_0, \ldots, x_{n-1}$ à partir de m_0, c'est calculer les mesures $m_1 = m_0 \oplus \|x_0\|$, …, $m_{i+1} = m_i \oplus \|x_i\|$ jusqu'à trouver un élément x_i qui fait passer un prédicat $p : \mu \to \texttt{bool}$ de `false` à `true`.



Un tel x_i existe toujours si $p(m_0) = \texttt{false}$ et $p(m_0 \oplus \|t\|) = \texttt{true}$. Il n'est pas nécessairement unique.

Découper une séquence, c'est balayer puis renvoyer (ℓ, x_i, r) où ℓ sont les éléments qui précèdent x_i et r les éléments qui suivent x_i.

Une implémentation avec des foncteurs en OCaml

```
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
```

Une implémentation avec des foncteurs en OCaml

```
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 aux *finger trees* : accès direct

Hinze et Paterson (2006) montrent comment annoter les *finger trees* par des mesures et implémenter une opération `split` en temps $\mathcal{O}(\log n)$.

En utilisant le monoïde des tailles, on obtient l'accès direct au i^{e} élément de la séquence, en temps $\mathcal{O}(\log n)$.

```
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 aux *finger trees* : file de priorité

En utilisant le monoïde des intervalles, on obtient une file de priorité min-max, avec

- accès au plus petit / plus grand élément en $\mathcal{O}(\log n)$;
- insertion en $\mathcal{O}(1)$ amorti.

```
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 aux *finger trees* : séquence triée

On peut aussi utiliser le monoïde de la dernière valeur :

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

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

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

Si la séquence est gardée triée par ordre croissant, ces annotations de «dernières valeurs» permettent de faire des recherches dichotomiques comme sur un A.B.R.

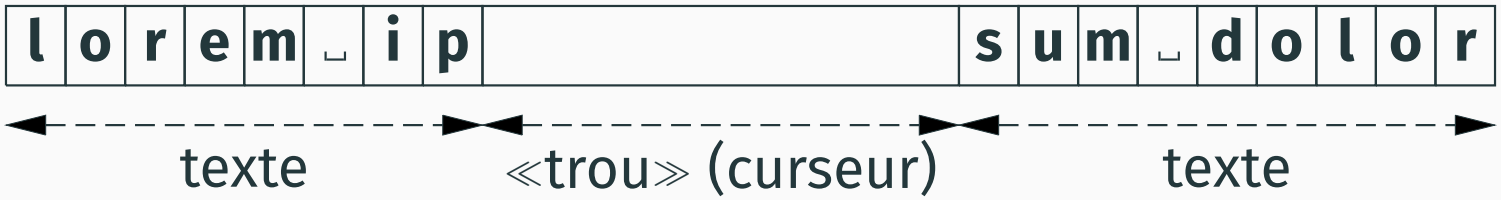
`split` nous permet d'implémenter l'insertion, la suppression, la recherche d'un élement, en temps $\mathcal{O}(\log n)$.

`head`, `tail`, `last`, `take` nous donnent accès au plus petit / plus grand élément en $\mathcal{O}(1)$ amorti.

Navigation dans une structure

Le tableau troué

Une structure de données utilisée par certains éditeurs de textes.



Un tableau de caractères, plus grand que le texte à éditer.

Le texte est stocké en 2 morceaux contigus, au début et en fin de tableau.

L'espace libre au milieu (le ≪trou≫) est le curseur d'édition.

Opérations d'édition du texte

Chargement du texte

abcde

Opérations d'édition du texte

Chargement du texte

abcde|

Reculer de 3 caractères

ab|cde

Opérations d'édition du texte

Chargement du texte

abcde|

Reculer de 3 caractères

ab|cde

Suppression arrière

a|cde

Opérations d'édition du texte

Chargement du texte		abcde|
Reculer de 3 caractères		ab|cde
Suppression arrière		a|cde
Insertion de «x» et «y»		axy|cde

Opérations d'édition du texte

Chargement du texte		abcde|
Reculer de 3 caractères		ab|cde
Suppression arrière		a|cde
Insertion de «x» et «y»		axy|cde
Suppression avant		axy|de

Opérations d'édition du texte

Chargement du texte		abcde|
Reculer de 3 caractères		ab|cde
Suppression arrière		a|cde
Insertion de «x» et «y»		axy|cde
Suppression avant		axy|de
Saut au début du texte		|axyde

Coût des opérations d'édition

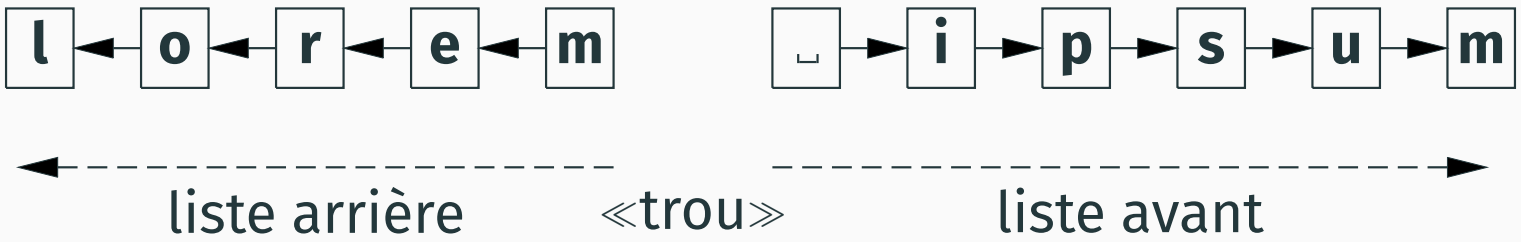
Le coût d'une opération d'édition est proportionnel à la distance entre le curseur courant et le point de l'opération.

Insertion, suppression	$\mathcal{O}(1)$
Avancer / reculer de d caractères	$\mathcal{O}(d)$
Sauter au début / à la fin du texte	$\mathcal{O}(n)$

(Plus : redimensionnement du tableau, coût amorti constant.)

Une liste (simplement chaînée) trouée

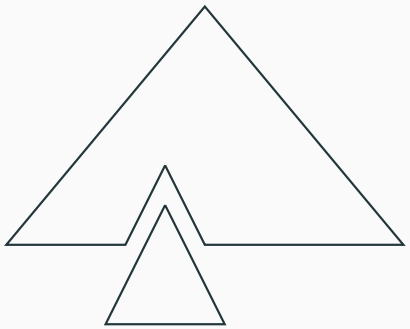
Une liste «arrière» et une liste «avant», avec le point d'insertion entre les deux listes.

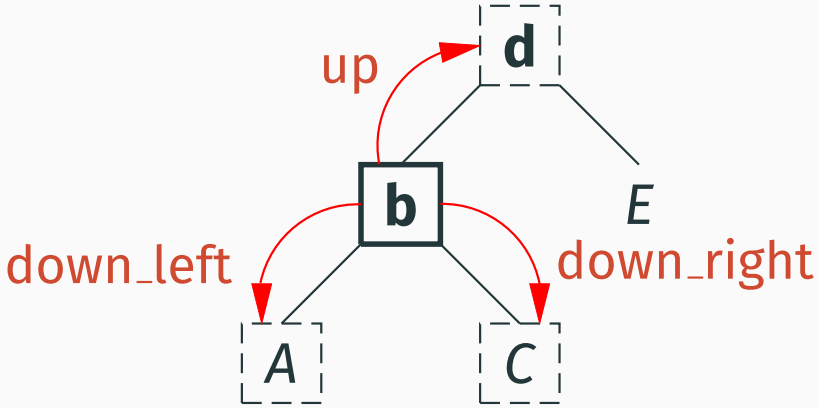


Navigation dans une liste trouée

```
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 dans un arbre





On voudrait se déplacer non seulement de feuille en feuille, mais plus généralement de sous-arbre en sous-arbre.

D'où une représentation par une paire

un sous-arbre (le curseur)
+ un arbre troué (le reste de l'arbre)

Arbre troué = contexte ?

En sémantique opérationnelle et dans d'autres domaines où on manipule des algèbres de termes, on a la notion de contexte C comme étant ≪un terme avec un trou≫ noté $[\,]$.

Par exemple, pour les arbres binaires

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

voici le type des contextes (= arbres avec un trou noté `Hole`) :

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

Opérations sur les contextes

L'opération principale, notée $C[t]$, reconstruit un terme en remplaçant dans C le trou [ ] par le terme 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)
```

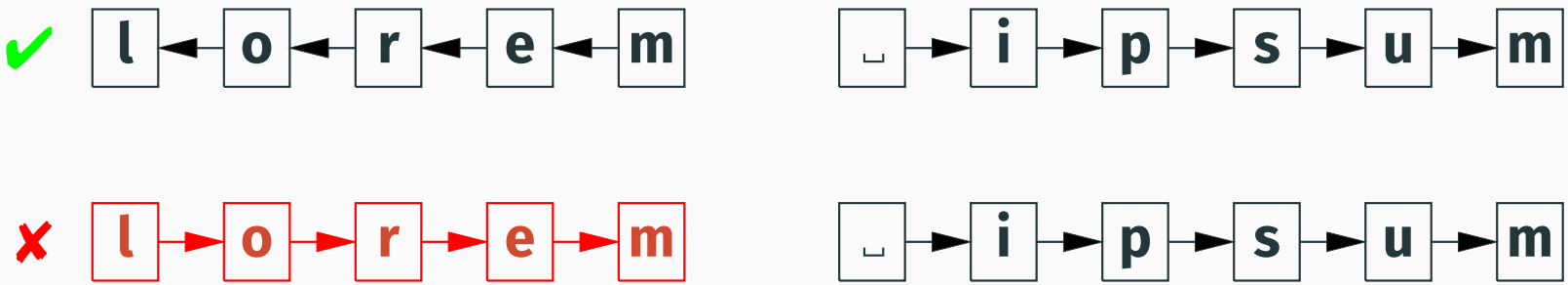
Navigation avec des contextes

La navigation sur des paires (contexte, sous-arbre) est possible mais les déplacements ne sont pas en temps constant!

```
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 avec des contextes

On aurait eu la même inefficacité dans l'exemple des listes trouées si on avait représenté la liste ≪arrière≫ dans le mauvais sens :



Wanted : un contexte d'arbre ≪renversé≫, avec ≪le trou≫ en haut.

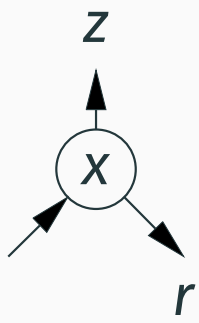
Les zippers (G. Huet, 1997)

Une représentation inversée des contextes, où le premier constructeur du zipper est au contact du «trou», et le dernier dénote la racine de l'arbre (`Top`).

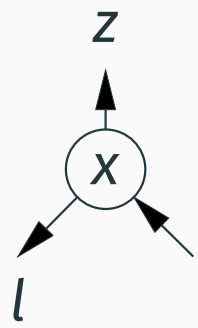
Exemple pour les arbres binaires :

```
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) :



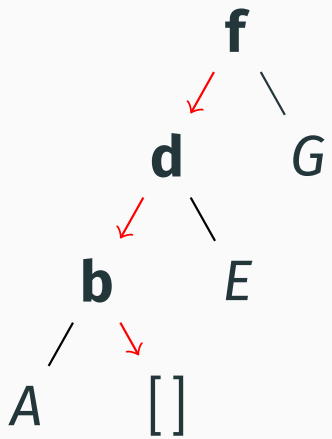
`Right_of`(l, x, z) :



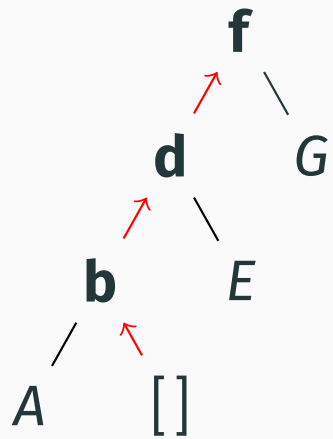
Zipper vs. contexte

Les constructeurs du contexte vont «vers le bas»,
ceux du zipper «vers le haut».

Contexte :



Zipper :



Contexte :

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

Zipper :

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

Opérations sur les zippers

L'opération principale, `app` *z t*, reconstruit un terme en plaçant le terme *t* dans le zipper *z*.

```
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))
```

Navigation avec des zippers

Les 3 déplacements élémentaires sont en temps constant !

```
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')
```

Recherche dichotomique avec des zippers

On peut présenter la recherche dichotomique dans un A.B.R. comme renvoyant l'endroit où se trouve / où devrait se trouver la valeur x recherchée.

```
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))
```

On peut ainsi se déplacer depuis n'importe quel point au point x :

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

Temps : $\mathcal{O}(\log n)$. (On peut faire mieux, voir plus loin.)

Application : une présentation fonctionnelle des arbres splay

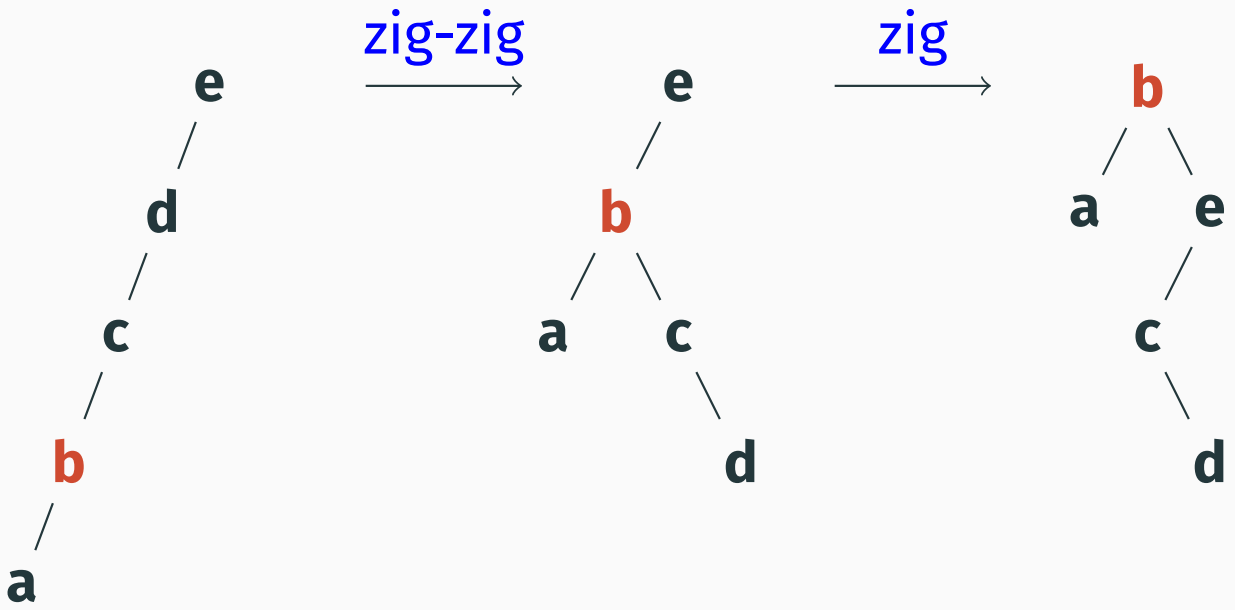
Les arbres splay (*splay trees*, Sleator et Tarjan 1985) :

- Des arbres binaires de recherche, sans critère explicite d'équilibrage.
- À chaque opération (recherche, insertion), l'élément x concerné est ramené au sommet de l'arbre, par des rotations bien choisies.
- Ces rotations réduisent progressivement les déséquilibres dans l'arbre, obtenant des opérations en temps $\mathcal{O}(\log n)$ amorti et $\mathcal{O}(n)$ dans le pire cas.

Exemple de recherche dans un arbre splay

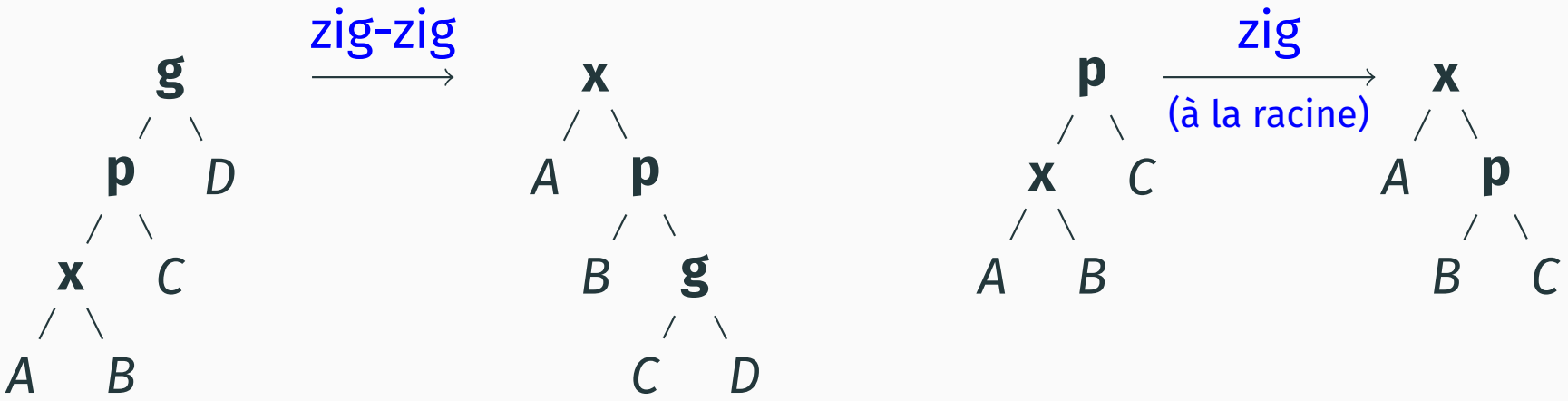
On recherche **b** dans l'arbre très déséquilibré à gauche **abcde**.

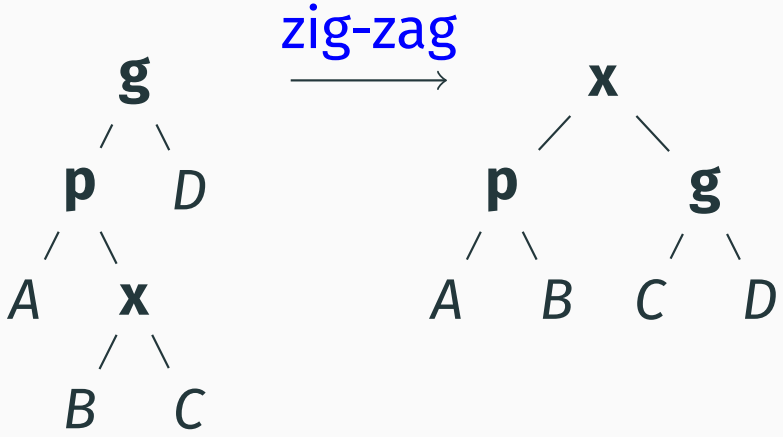
On remonte **b** au sommet par deux rotations, *zig-zig* et *zig*.



Les rotations des arbres splay

Pour ≪remonter≫ le nœud **x** au-dessus de son parent **p** et de son grand-parent **g**. (3 autres rotations symétriques omises.)





Le *splaying* vu comme une application de zipper

`splay l x r z` produit un A.B.R. équivalent à `app z (Node(l, x, r))` mais avec `x` au sommet.

```
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
```

L'insertion dans un arbre splay

Insertion = recherche

+ création d'un nœud si nécessaire

+ *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 *)
```

(Pour une présentation fonctionnelle pure mais sans zippers des arbres splay, voir Nipkow et al, *FAV!*, chap. 21.)

Liens avec la dérivation formelle

Contextes ≈ zippers ≈ listes de deltas

Pour un type algébrique régulier, le type des zippers et le type des contextes sont isomorphes à une liste de 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 *)
```

Appliquer les deltas dans le bon ordre

```
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)
```

Pour un contexte :

- premier delta = sommet de l'arbre ; liste vide = le trou
- remplissage d'un contexte $C[t]$ = un *fold right* de `app`.

Pour un zipper :

- premier delta = voisinage du trou ; liste vide = le sommet
- application d'un zipper = un *fold left* de `app`.

Construire le type des deltas à partir du type de données

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

```
type 'a delta =

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

Un constructeur sans occurrence récursive du type disparaît.

Un constructeur à n arguments dont k ont le type défini récursivement devient k constructeurs à $n - 1$ arguments, obtenus en supprimant une des k occurrences récursives du type.

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

Une algèbre de types

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

Isomorphismes : $+$ et $\times$ sont commutatifs et associatifs ; de plus,

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

Points fixes de types

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

Un type algébrique régulier `type` $t = \tau$ est un point fixe $\mu t.\tau$.

Exemples :

$\mu t.\ \mathbf{1} + t$	entiers de Peano (`Zero`/`Succ`)
$\mu t.\ \mathbf{1} + \alpha \times t$	listes de α (`Nil`/`Cons`)
$\mu t.\ \alpha + t \times t$	arbres binaires avec des α aux feuilles
$\mu t.\ \mathbf{1} + t \times \alpha \times t$	arbres binaires avec des α aux nœuds
$\mu t.\ \alpha + t \times t + t \times t \times t$	arbres 2-3 avec des α aux feuilles

Le type des deltas

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

Le type des deltas pour le type algébrique régulier $\mu t.\tau$ est

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

c.à.d. la dérivée formelle du type τ par rapport à la variable t prise au point $\mu t.\tau$.

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

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

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

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

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

Exemples de dérivées de types

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

Le type des deltas pour les entiers de Peano est `unit`.

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

Le type des deltas pour les listes de α est α.

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

Le type des deltas pour les arbres binaires avec des α aux feuilles est ≪un arbre ou un arbre≫, c.à.d. ≪un booléen et un arbre≫.

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

Le type des deltas pour les arbres binaires avec des α aux nœuds est ≪un α et un arbre, ou un arbre et un α≫.

Pour aller plus loin

Extension aux types non réguliers et aux GADT :

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

Une formalisation du lien entre dérivation formelle et contextes :

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

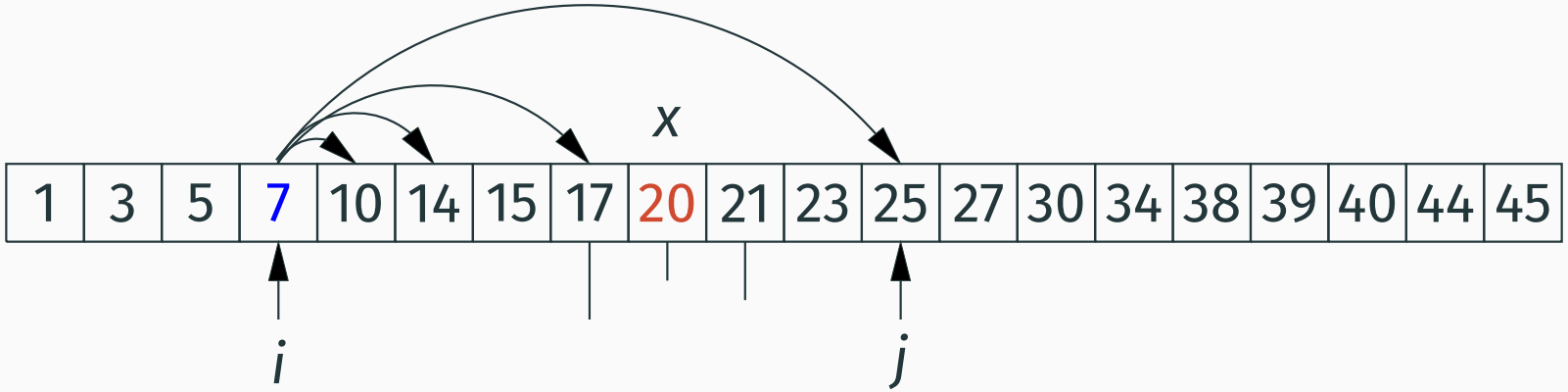
Les index

Les index (*fingers*) (Guibas, McCreight, Plass, Roberts, 1977)

Un index = un ≪pointeur≫ sur un élément x d'une structure de données, qui permet d'accélérer les opérations portant sur les éléments proches de x.

Typiquement, si une opération ordinaire est en temps $\mathcal{O}(f(n))$,
où n est la taille de la structure,
l'opération avec index est en temps $\mathcal{O}(f(d))$,
où d est la distance par rapport à l'index.

Exemple : index dans un tableau trié



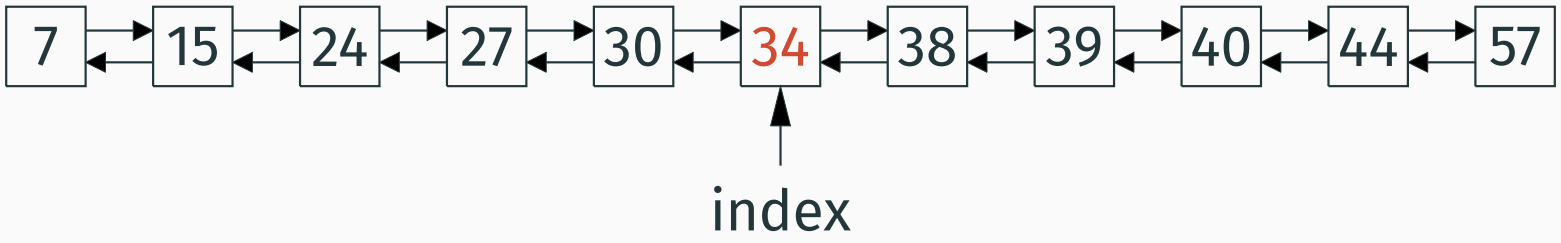
Un index sur l'élément x_i = son indice i.

Pour rechercher un élément $x > x_i$ à proximité de x_i :

- On prend $j = i + 1, i + 2, i + 4, i + 8, \ldots$ jusqu'à ce que $x \leq x_j$.
- On fait une recherche dichotomique usuelle entre i et j.

Temps : $\mathcal{O}(\log(j - i))$, c.à.d. $\mathcal{O}(\log d)$ car $j - i \leq 2d$.

Exemple : index dans une liste doublement chaînée triée



À partir d'un pointeur sur l'élément x_i, on peut

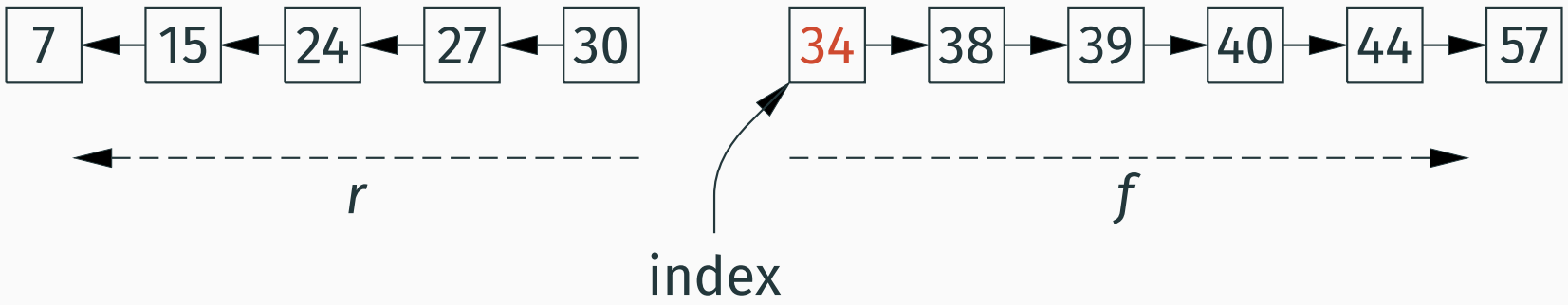
- supprimer, insérer avant/après cet élément en temps $\mathcal{O}(1)$;
- rechercher l'élément x_j en temps $\mathcal{O}(d)$, où $d = |j - i|$.

On peut descendre à $\mathcal{O}(\log d)$ en remplaçant la liste par une *skip list* ou par un arbre équilibré (un B-tree dans Guibas et al 1977).

Les zippers comme index purement fonctionnels

Si on a un seul index par structure de donnée, il peut souvent se représenter comme une paire (zipper, sous-terme).

Exemple : un index dans une liste triée = une liste trouée (r, f) avec r triée par ordre décroissant et f par ordre croissant.



Pour rechercher un élément x au voisinage du trou, on cherche dans f si $f \neq [\,]$ et $x \geq \mathtt{hd}(f)$, ou dans r si $r \neq [\,]$ et $x \leq \mathtt{hd}(r)$. Temps $\mathcal{O}(d)$.

Un index dans un arbre binaire de recherche

Un index sur un sous-arbre quelconque d'un A.B.R. peut aussi se représenter comme une paire (t, z) où t est un sous-arbre et z est un zipper d'arbre.

Pour rechercher un élément x au voisinage, il faut un critère rapide (temps constant) pour savoir s'il faut chercher dans t ou bien ≪remonter≫ dans z pour élargir la recherche.

Solution : annoter chaque arbre par l'intervalle des valeurs qu'il contient, comme vu en début de cours.

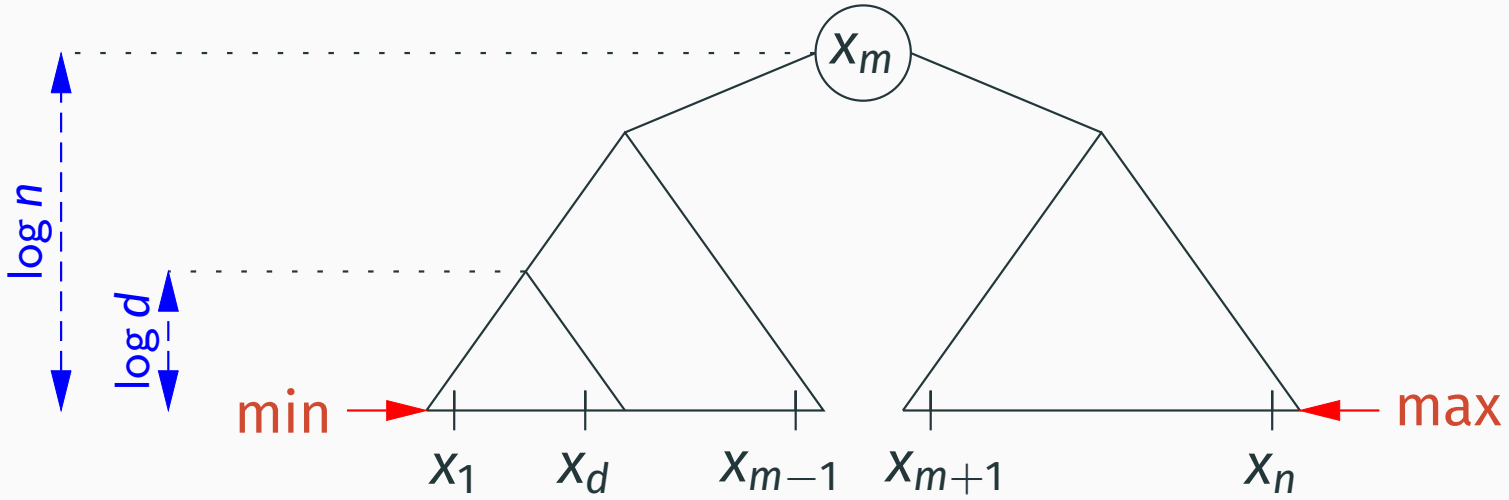
Si x est dans l'intervalle de t, il faut chercher dans t (et nulle part ailleurs). Sinon, il faut élargir la recherche.

Recherche à partir d'un index dans un A.B.R.

```
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)
```

Généralement plus efficace que `search x (app z t, Top)`
mais pas en $\mathcal{O}(\log d)$...

Recherche à partir d'un index dans un A.B.R.



Cas favorable : si (t, z) pointe sur le plus petit élément x_1 (ou le plus grand). Supposant l'arbre équilibré, le sous-arbre contenant x_1 et x_d contient $\mathcal{O}(d)$ éléments et est de hauteur $\mathcal{O}(\log d)$.

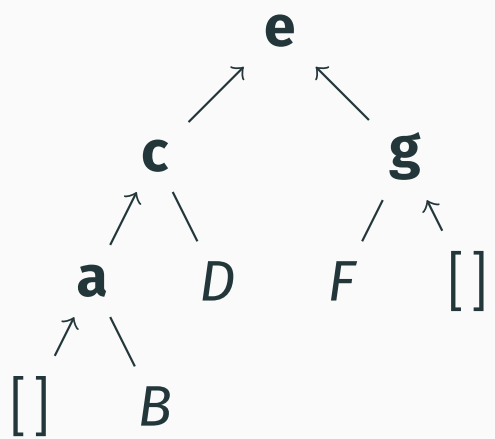
Cas défavorable : l'index pointe sur x_{m-1} (le plus grand élément du sous-arbre gauche) et on cherche x_{m+1}. La distance d est 2 mais on va quand même remonter au sommet.

Deux index dans une structure

En général, les zippers ne permettent pas d'avoir plusieurs index dans une structure.

Exception : deux index dans un A.B.R., un sur le plus petit élément et l'autre sur le plus grand élément.

On représente la branche la plus à gauche et la branche la plus à droite par des zippers, pointant de bas en haut.



Index min et index max dans un A.B.R.

```
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)
```

Recherche à partir d'un index min-max

Recherche plus facile que depuis un index quelconque : pas besoin d'intervalles, et temps garanti $\mathcal{O}(\log d)$ où d est la plus petite distance au min ou au max.

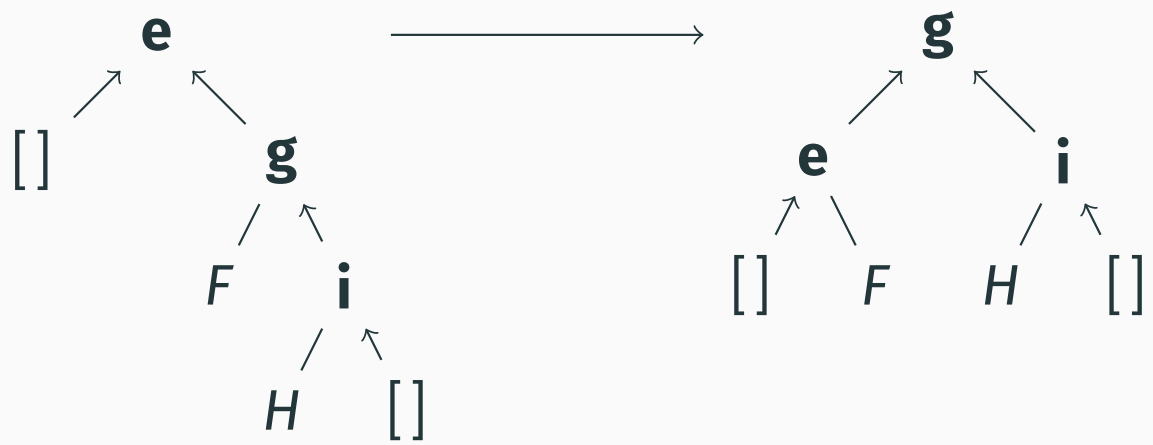
```
let rec mem_left v = function
  | [] -> false
  | [(x, r)] -> v = x || mem x r
  | (x1, r1) :: ((x2, _) :: _) as lz ->
      if v < x1 then false else
      if v = x1 then true else
      if v < x2 then mem v r1 else mem_left v lz

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)
```

Rééquilibrage de l'A.B.R. à index min-max

Le rééquilibrage local au zipper gauche ou au zipper droit est assez simple ; chaque rotation se fait en temps constant.

Les rotations impliquant le sommet de l'arbre sont plus coûteuses (rotation en temps $\log n$) car il faut accéder aux derniers éléments des zippers.

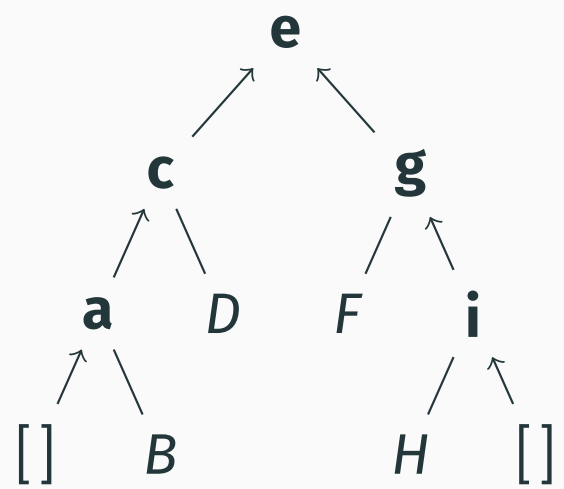
Exemple : rééquilibrage lorsque le zipper gauche est vide.

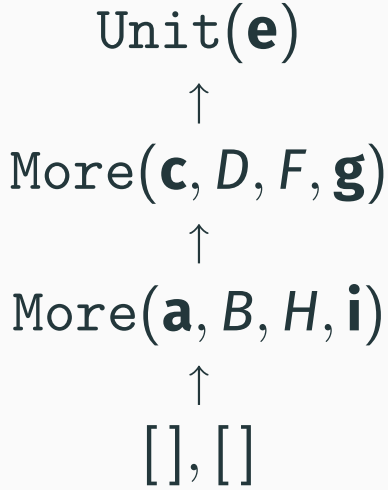


Index min-max sur un arbre parfait

Si l'arbre binaire est parfait, toutes les branches ont la même longueur, y compris le zipper gauche et le zipper droit.

On peut donc fusionner les deux zippers en un seul !

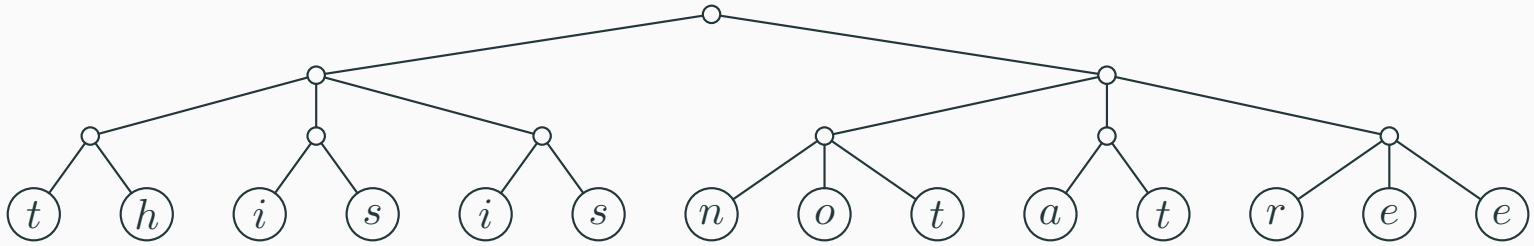




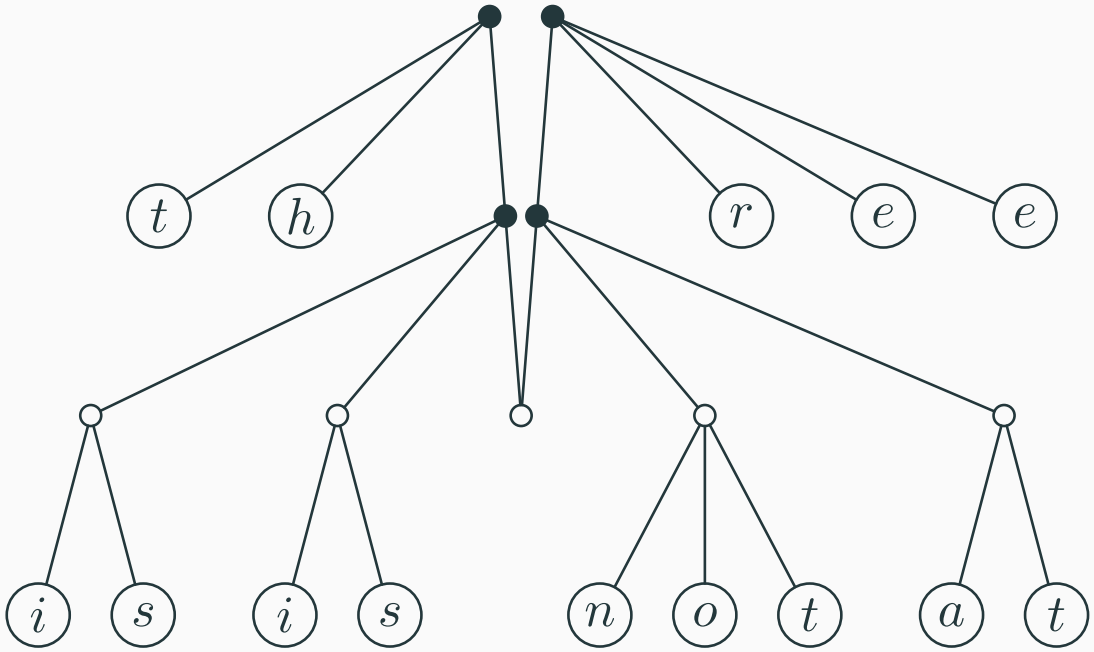

```
type 'a minmax =
  | Empty | Unit of 'a
  | More of 'a * 'a tree * 'a minmax * 'a tree * 'a
```

Les *finger trees* : des index min-max sur un arbre 2-3

Un arbre 2-3 avec valeurs aux feuilles : (Hinze et Paterson 2006)



Le même tenu par les nœuds le plus à gauche et le plus à droite :



Les *finger trees* : des index min-max sur un arbre 2-3

Un type non régulier des arbres 2-3 avec valeurs aux feuilles :

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

Avec des index min et max à base de zippers :

```
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
```

Point d’étape

Point d'étape

Un peu d'algèbre nous permet d'ajouter de nouveaux traits à nos structures de données :

- Annotation par des mesures à valeurs dans des monoïdes
  $\rightarrow$ accès par rang, par valeur min ou max, …
- Dérivation formelle pour construire les zippers
  $\rightarrow$ navigation, index, …

Bibliographie

Bibliographie

L'article original sur les zippers :

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

L'article original sur les *finger trees*, qui introduit également l'idée des annotations par monoïde :

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