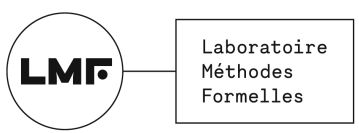
Parlons OCaml

Jean-Christophe Filliâtre

Algorithmique et programmation
Luminy, 5–9 mai 2025



cet après-midi

- initiation à OCaml avec le projet Euler 215

ce matin

- programmation fonctionnelle
- arbres de Braun et tableaux flexibles

c'est programmer sans données mutables

on construit de nouvelles valeurs, sans altérer les précédentes

- raisonnement plus simple

  grâce à la transparence référentielle

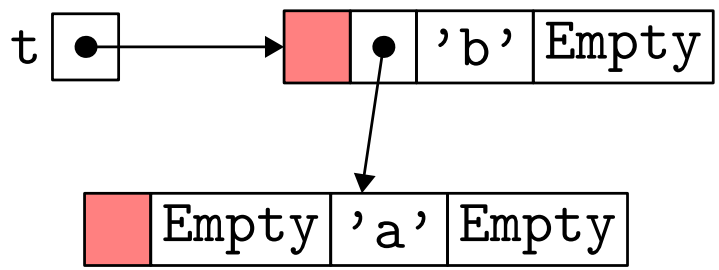
  ainsi, `add` x s dénotera toujours l'ensemble $\{x\} \cup s$

- code plus simple

des arbres binaires de recherche immuables

```
type bst =
  | Empty
  | Node of bst * elt * bst
```

```
let t =
  Node (Node (Empty, 'a', Empty),
        'b',
        Empty)
```



l'insertion d'un nouvel élément s'écrit naturellement

```
let rec add (x: elt) (t: bst) : bst =
  match t with
  | Empty ->
      Node (Empty, x, Empty)
  | Node (l, y, r) ->
      if x < y then Node (add x l, y,       r)
               else Node (     l, y, add x r)
```

filtrage

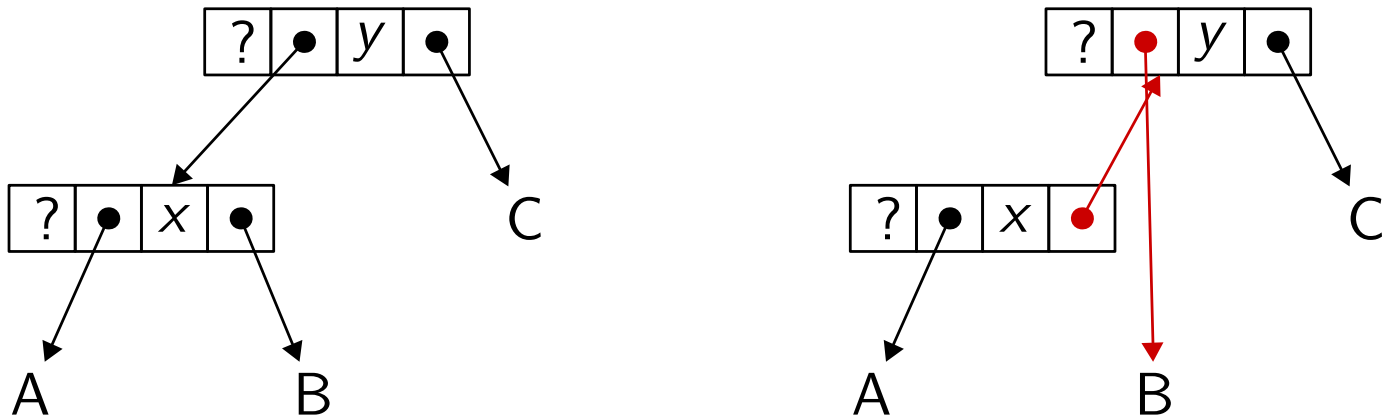
renvoie un arbre

reconstruit le nœud

- le code serait un peu plus pénible à écrire
  (exercice : le faire !)

- le risque d'erreur est important

avec des arbres AVL (équilibrés par la hauteur),
une rotation droite s'écrit simplement en deux affectations



mais on oublie facilement de mettre à jour les hauteurs stockées dans les nœuds !

alors qu'avec des AVL immuables

```
type height = int
type avl = Empty | Node of height * avl * elt * avl
```

on se donne un « constructeur intelligent »

```
let height = function Empty -> 0 | Node (h,_,_,_) -> h

let node (l: avl) (x: elt) (r: avl) : avl =
  Node (1 + max (height l) (height r), l, x, r)
```

$\mathcal{O}(1)$

et on écrit la rotation droite facilement, sans piège :

```
let rotate_right : avl -> avl = function
  | Empty | Node (_, Empty, _, _) ->
      assert false
  | Node (_, Node (_, ll, x, lr), y, r) ->
      node ll x (node lr y r)
```

- parfois un peu plus de temps

  exemple : ABR plutôt que table de hachage $\Rightarrow$ facteur log

- un peu plus d'espace

  exemple : l'insertion dans un arbre équilibré reste $\mathcal{O}(\log n)$ en temps mais elle alloue $\mathcal{O}(\log n)$ au lieu de $\mathcal{O}(1)$

  car on a copié le chemin de la racine au nouveau nœud

le caractère immuable permet le partage

ainsi, les arbres t et add x t partagent de nombreux sous-arbres

un tableau contenant tous les ensembles

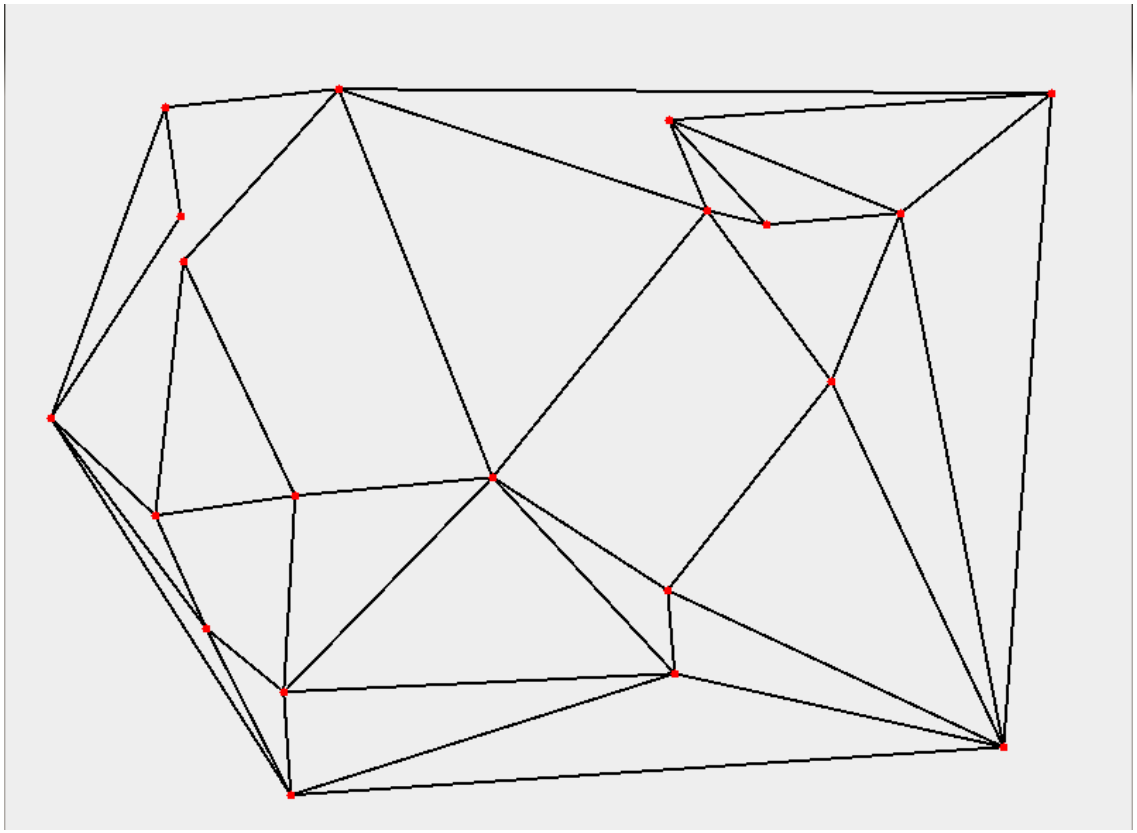
$$\{\}, \{1\}, \{1, 2, \}, \ldots, \{1, 2, \ldots, n-1\}$$

```
module S = Set.Make(Int)
let n = 1_000_000
let a = Array.make n S.empty
let () = for i = 1 to n-1 do a.(i) <- S.add i a.(i-1) done
```

ce sont des AVL ←

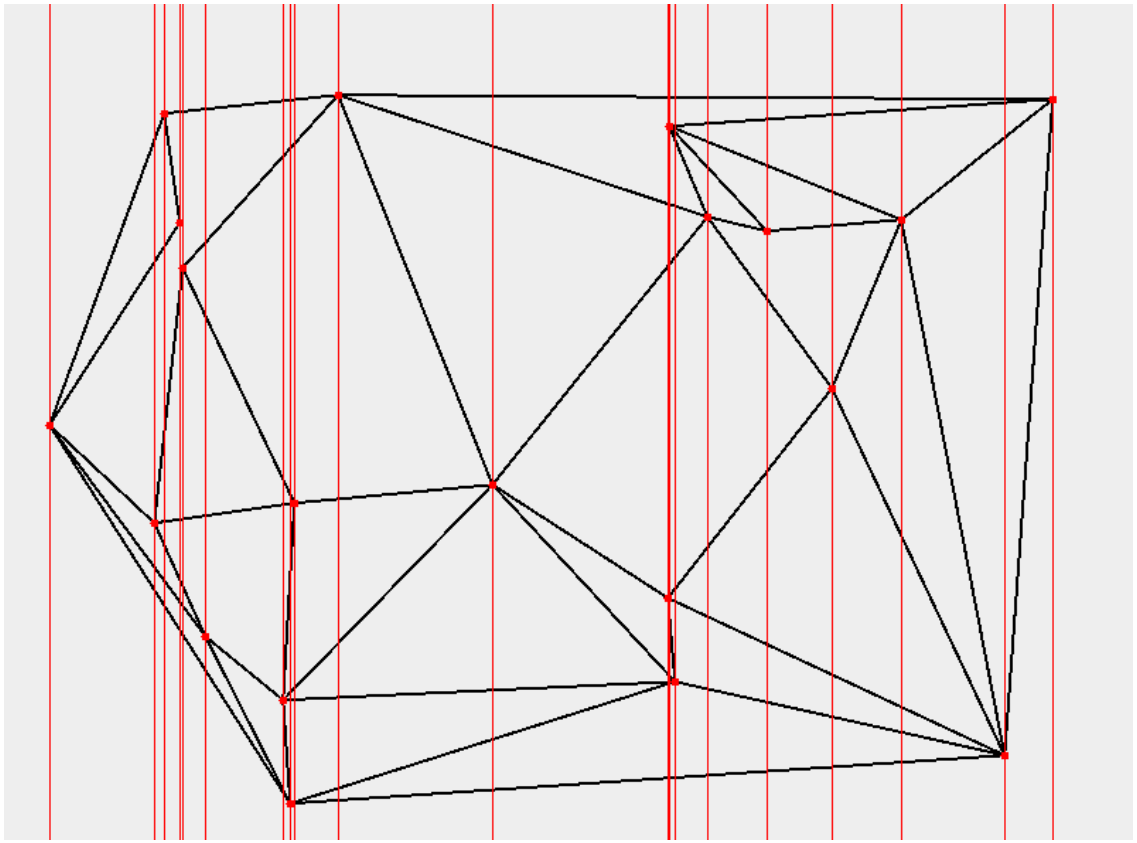
fait remarquable,
sa construction a demandé un temps et un espace $n \log n$

le problème de la localisation du point :



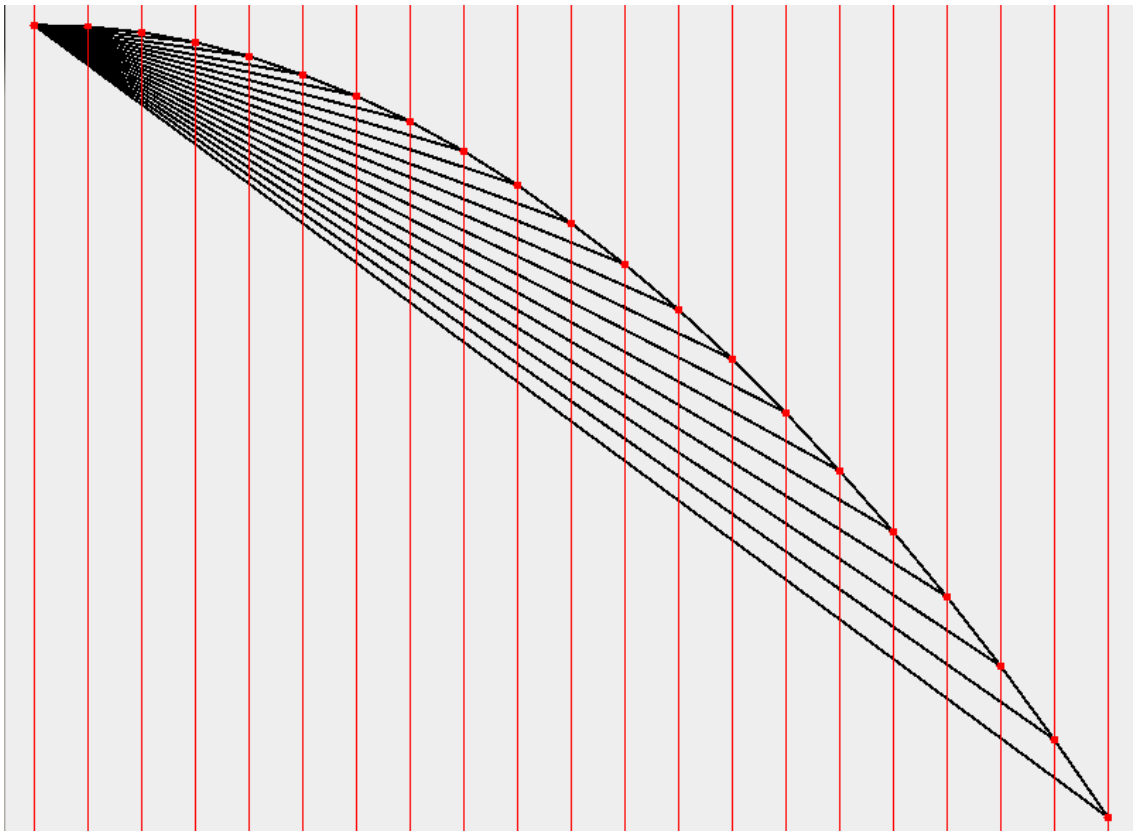
étant donné un point, dans quel polygone se trouve-t-il ?

le problème de la localisation du point :



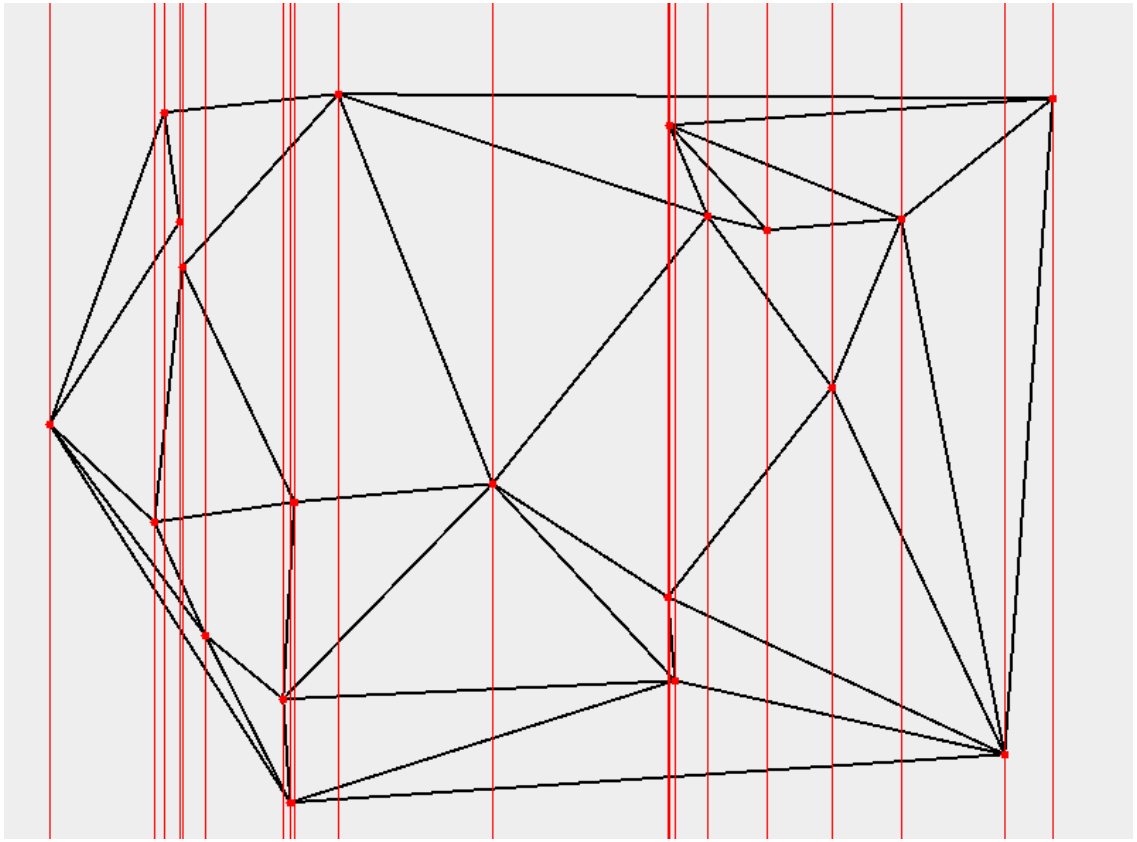
étant donné un point, dans quel polygone se trouve-t-il ?

le problème de la localisation du point :



potentiellement un espace quadratique

le problème de la localisation du point :



bande = AVL immuable $\Rightarrow$ temps et espace $\mathcal{O}(n \log n)$

le partage de structures immuables est aussi exploité dans

- la structure de corde
- les BDD / les IDD / HashLife
  (partage maximal + mémoïsation)

2

arbres de Braun et tableaux flexibles

des tableaux immuables

```
make: int -> 'a -> 'a t
get : 'a t -> int -> 'a
set : 'a t -> int -> 'a -> 'a t
```

← *n fois la même valeur*

offrant l'ajout et le retrait de chaque côté

```
cons: 'a -> 'a t -> 'a t
tail: 'a t -> 'a t
snoc: 'a t -> 'a -> 'a t
liat: 'a t -> 'a t
```

et toutes ces opérations en temps $\mathcal{O}(\log n)$

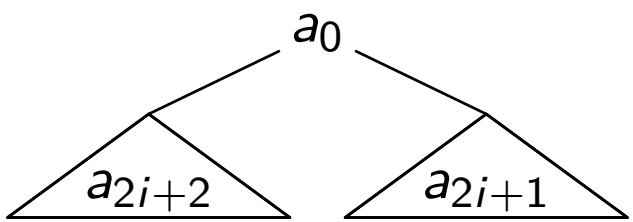
un tableau flexible est un arbre binaire

```
type 'a t = Empty | Node of 'a t * 'a * 'a t
```

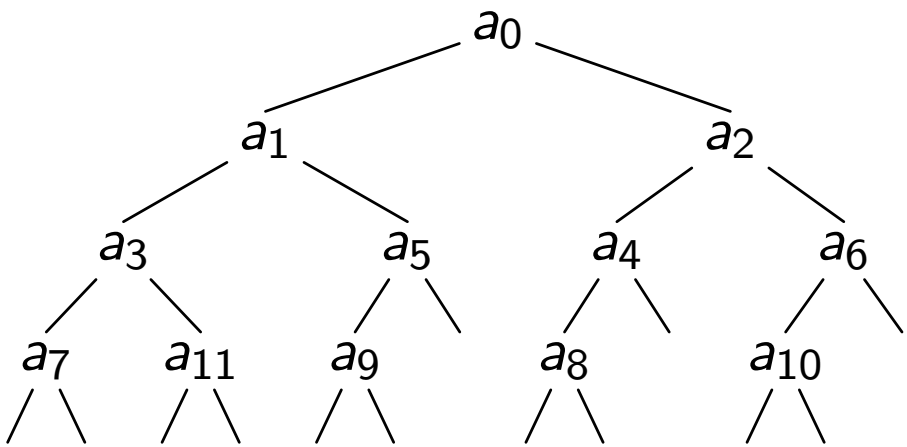
et pour représenter le tableau

$$a_0, a_1, \ldots, a_{n-1}$$

on utilise, récursivement, le schéma suivant :



avec $n = 12$ éléments :



Définition : un arbre de Braun est un arbre binaire dans lequel, pour tout nœud `Node(`ℓ`, x, r)`, on a

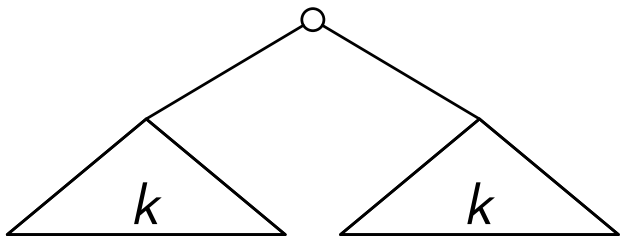
$$|r| \leq |\ell| \leq |r| + 1$$

où $|t|$ désigne la taille de l'arbre t

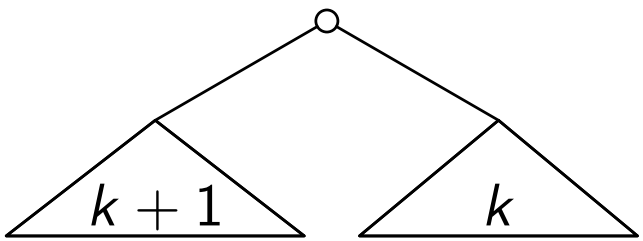
(exercice : montrer que la hauteur est logarithmique)

la forme d'un arbre de Braun est imposée par sa taille n

- si $n = 0$, alors c'est l'arbre vide
- si $n = 2k + 1$, alors



- si $n = 2k + 2$, alors



$\Rightarrow$ unicité de représentation d'un tableau flexible

$\Rightarrow$ on peut utiliser l'égalité structurelle polymorphe d'OCaml (=)

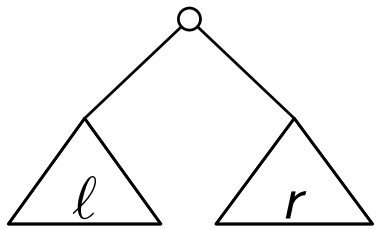
détaillons les trois opérations

1. `size: 'a t -> int`
2. `make: int -> 'a -> 'a t`
3. `iter: (int -> 'a -> unit) -> 'a t -> unit`

bien entendu, on peut calculer la taille de l'arbre en temps linéaire

cependant, la structure très rigide d'un arbre de Braun nous permet de faire beaucoup mieux

pour un arbre de Braun



on peut

1. calculer $k = |r|$
2. calculer $\delta = |\ell| - |r| \in \{0, 1\}$
3. renvoyer $1 + 2k + \delta$

```
let rec size = function
  | Empty ->
      0
  | Node (l, _, r) ->
      let k = size r in 1 + 2*k + diff k l
```

renvoie $|\texttt{l}| - \texttt{k}$

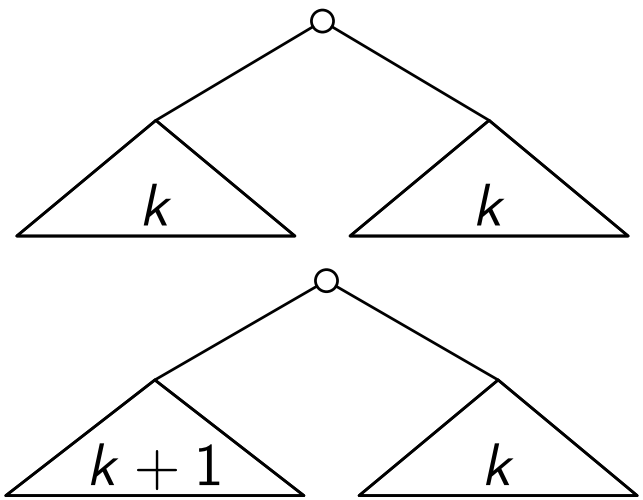
`diff` n ℓ renvoie $|\ell| - n$ sachant que $n \leq |\ell| \leq n + 1$

cas de base : $0 \leq |\ell| \leq 1$

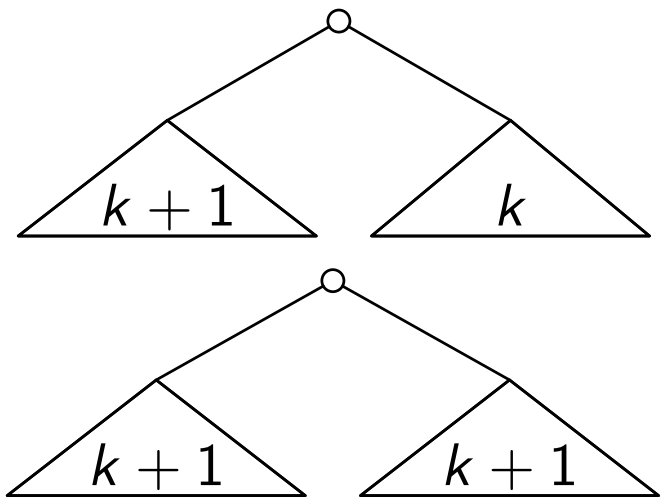
```
let rec diff n = function
  | Empty ->
      0
  | Node (Empty, _, Empty) ->
      1 - n
```

sinon, on se laisse guider par la parité de n

$n = 2k + 1$



$n = 2k + 2$




```
  | Node (l, _, r) ->
      diff ((n-1) / 2) (if n mod 2 = 1 then l else r)
```

`diff` est clairement en $\mathcal{O}(\log n)$

et `size` est donc en $\mathcal{O}(\log^2 n)$

un arbre de Braun de taille n où tous les éléments sont identiques à v

comme pour `size`, ce serait trivial à faire en temps $\mathcal{O}(n)$

mais on peut sûrement exploiter le partage de sous-arbres,
comme par exemple pour $n = 7$:

```
let t1 = Node (Empty, v, Empty) in
let t3 = Node (t1,    v, t1   ) in
         Node (t3,    v, t3   )
```

idée : renvoyer un arbre de taille n et un arbre de taille $n+1$

```
let rec make2 (n: int) (x: 'a) : 'a t * 'a t =
```

cas de base

```
  if n = 0 then
    Node (Empty, x, Empty), Empty
```

sinon

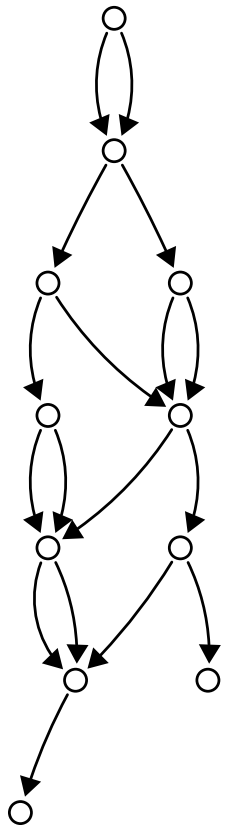
```
  else if n mod 2 = 1 then
    let l, r = make2 (n / 2) x in
    Node (l, x, r), Node (r, x, r)
  else
    let l, r = make2 ((n - 1) / 2) x in
    Node (l, x, l), Node (l, x, r)
```

on en déduit la fonction voulue

```
let make n x =
  snd (make2 n x)
```

et sa complexité est clairement $\mathcal{O}(\log n)$

pour $n = 89$, on a seulement 11 nœuds



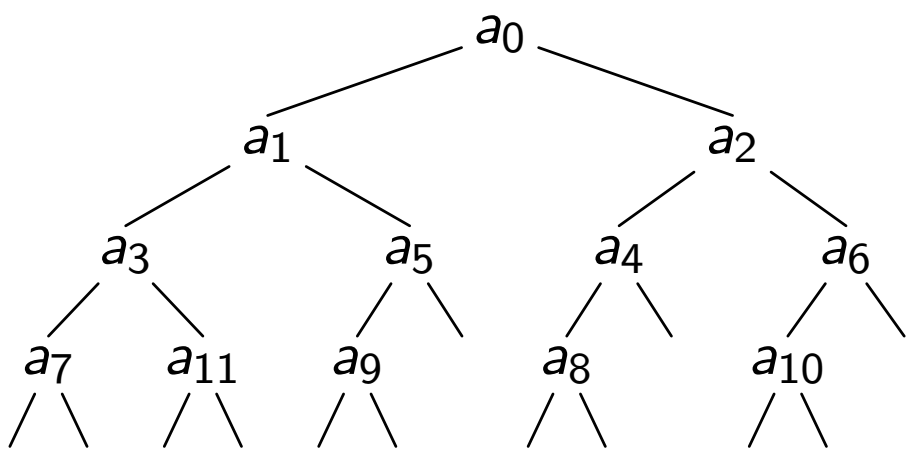
- exercice : remarquer que `make2` ne réalise pas un partage maximal pour $n = 1$ et rectifier
- exercice : améliorer `make` pour ne construire que des nœuds utilisés dans le résultat final (en gardant le principe de `make2`)
- exercice : écrire `make` avec de la mémoïsation

```
iter: (int -> 'a -> unit) -> 'a t -> unit
```

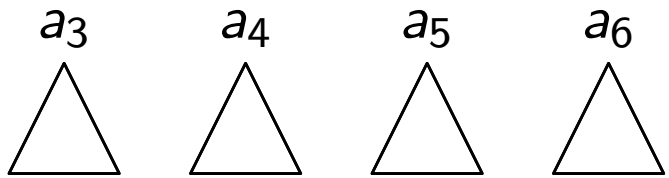
on veut appliquer la fonction à i et a_i, dans l'ordre croissant des i

parcourir les éléments dans l'ordre n'est pas immédiat

car il faut alterner entre les sous-arbres



utilisons une file C contenant, dans l'ordre, les arbres d'un même niveau, par exemple,



pendant leur parcours, on met

- leurs sous-arbres gauche $(a_7, a_8, \dots)$ dans une deuxième file L
- leurs sous-arbres droit $(a_{11}, a_{12}, \dots)$ dans une troisième file R

une fois le parcours de C terminé, il suffit de faire $C \leftarrow L + R$
(en $\mathcal{O}(1)$ avec `Queue.transfer` !)

ça donne

```
let iter (f: int -> 'a -> unit) (a: 'a t) : unit =
  let n = size a in
  let add t q = if t <> Empty then Queue.add t q in
  let rec loop i current left right = if i < n then (
    if Queue.is_empty current then (
      Queue.transfer right left;
      loop i left current right
    ) else match Queue.pop current with
      | Empty ->
          assert false
      | Node (l, x, r) ->
          f i x;
          add l left;
          add r right;
          loop (i + 1) current left right) in
  let current = Queue.create () in
  add a current;
  loop 0 current (Queue.create ()) (Queue.create ())
```

complexité $\mathcal{O}(n)$

exercice : écrire une fonction

```
init: (int -> 'a) -> int -> 'a t
```

qui renvoie le tableau des $f(i)$ pour une taille donnée

exercice : écrire une fonction

```
of_list: 'a list -> 'a t
```

on peut avantageusement utiliser les tableaux flexibles pour écrire un interprète sans substitution du λ-calcul avec indices de de Bruijn

(voir par exemple la machine de Krivine)

on peut également utiliser les arbres de Braun pour réaliser des files de priorité

on peut vérifier formellement du code OCaml

- avec Rocq
- avec Why3
  - tableaux flexibles
  - files de priorités avec des arbres de Braun

- W. Braun, M. Rem, A logarithmic implementation of flexible arrays. Memorandum MR83/4. Eindhoven University of Technology. 1983.

- Hoogerwoord, R. R. A logarithmic implementation of flexible arrays. Conference on Mathematics of Program Construction. 1992.

- Chris Okasaki. Three Algorithms on Braun Trees (Functional Pearl). J. Functional Programming 7 (6) 661–666. 1997

- Xavier Leroy. Structures de données persistantes. Cours au Collège de France. 2023

récemment traduit en anglais

Learn Programming with OCaml

PDF librement accessible sur
`usr.lmf.cnrs.fr/lpo/`

les atouts d'OCaml

- le typage statique nous aide
- le filtrage nous aide
- on peut se laisser guider par les types et par les invariants

la force de la programmation fonctionnelle, c'est le partage