Structures de données persistantes, cinquième cours

Systèmes de numération et types non réguliers

Xavier Leroy

2023-04-07

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

Systèmes de numération

Structures de données et systèmes de numération

Pour mieux comprendre ou pour concevoir une structure de données, il peut être utile de la réduire à un nombre.

Typiquement : une collection $\rightarrow$ son nombre d'éléments.

Les opérations sur la structure correspondent alors à des opérations arithmétiques :

insertion	$\rightarrow$	incrément
suppression	$\rightarrow$	décrément
fusion (union disjointe)	$\rightarrow$	addition

La représentation en mémoire de la structure de données correspond à une certaine manière d'écrire le nombre, p.ex :

liste simplement chaînée	$\rightarrow$	entier de Peano

Listes et entiers de Peano

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

```
type num =
  | Zero
  | Succ of num
```

Opérations en temps constant :

- `cons` ($\ell \to \texttt{Cons}(x, \ell)$)
- `tail` ($\texttt{Cons}(x, \ell) \to \ell$)
- incrément ($n \to \texttt{Succ}\ n$)
- décrément ($\texttt{Succ}\ n \to n$)

Opérations en temps linéaire :

- concaténation (ℓ_1 `@` ℓ_2)
- n^{e} élément (`List.nth` $\ell\ n$)
- addition ($n_1 + n_2$)
- comparaison ($> n$)

Les entiers binaires

Nombre	Représentation	Nombre	Représentation
0 :		8 :	**0001**
1 :	**1**	9 :	**1001**
2 :	**01**	10 :	**0101**
3 :	**11**	11 :	**1101**
4 :	**001**	12 :	**0011**
5 :	**101**	13 :	**1011**
6 :	**011**	14 :	**0111**
7 :	**111**	15 :	**1111**

Représentation ≪petit-boutiste≫ (poids faibles en premier) :
une liste de chiffres $d_0, d_1, \ldots, d_{p-1}$ avec $d_i \in \{\mathbf{0}, \mathbf{1}\}$.

Cette liste dénote le nombre entier $\sum_{i=0}^{p-1} d_i \cdot 2^i$.

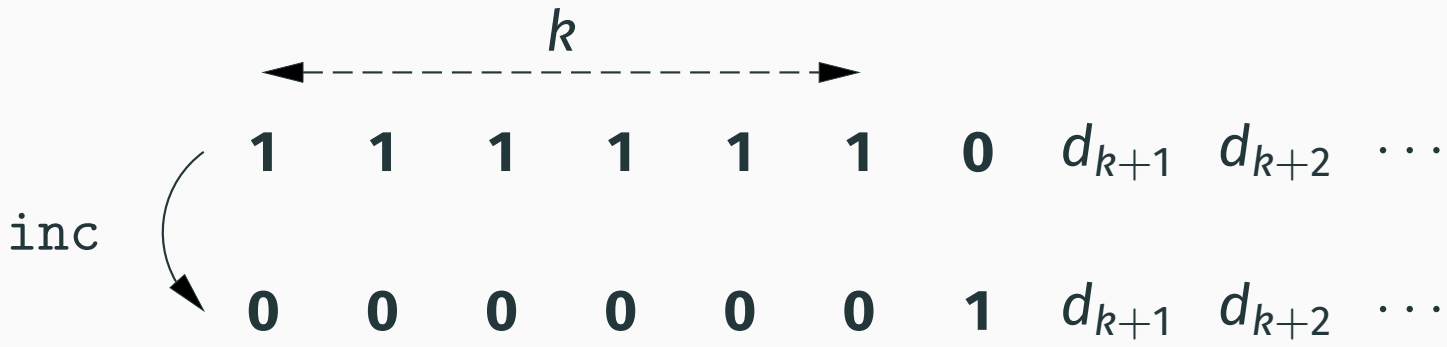
Représentation et opérations élémentaires

```
type digit = Zero | One
type num = digit list

let rec inc = function
  | [] -> [One]
  | Zero :: n -> One :: n
  | One :: n -> Zero :: inc n

let rec dec = function
  | [] -> raise Error
  | [One] -> []
  | One :: n -> Zero :: n
  | Zero :: n -> One :: dec n
```

Complexité algorithmique de l'incrément

```
let rec inc = function
  | [] -> [One]
  | Zero :: n -> One :: n
  | One :: n -> Zero :: inc n
```

`inc` prend un temps proportionnel à $k + 1$,
où k est le nombre de **1** qui précèdent le premier **0** :



Si n est le nombre dénoté par la liste, on a $n \geq 2^k - 1$.

Donc, `inc` s'exécute en temps $\mathcal{O}(\log n)$ dans le pire cas.

Analyse amortie de l'incrément

```
let rec inc = function
  | [] -> [One]
  | Zero :: n -> One :: n
  | One :: n -> Zero :: inc n
```

On dit qu'un chiffre est dangereux s'il peut provoquer une retenue qu'il faut propager, et non dangereux s'il arrête le calcul. Pour `inc`, **1** est dangereux, **0** non dangereux.

On prend $\Phi(n) =$ nombre de chiffres dangereux dans la liste n.

Si k est le nombre de **1** qui précèdent le premier **0**,

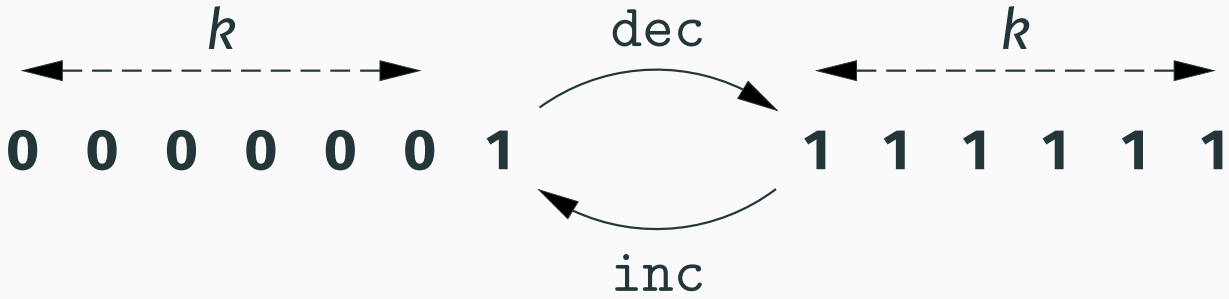
- `inc` prend le temps réel $k + 1$
- $\Delta\Phi = 1 - k$ (car un **1** apparaît et k **1** deviennent des **0**)

Donc `inc` s'exécute en temps amorti constant.

Analyse amortie de l'incrément et du décrément

Une analyse similaire montre que `dec` est en temps amorti constant. (En prenant **0** comme chiffre dangereux.)

Et pourtant une séquence de n `inc` et `dec` peut prendre un temps $n \log n$ …



On fait $n = 2^k$ opérations `inc` pour aller de 0 à 2^k,
puis n séquences `dec; inc`, chacune prenant un temps $2k$

$\rightarrow 3n$ opérations en temps $2n \log n$.

Erreur d'analyse : on a utilisé des potentiels Φ différents pour analyser `inc` et `dec`!

Un système de numération

Pour chaque position i,

un poids $w_i \in \mathbb{N}+$;

un ensemble de chiffres autorisés $D_i \subseteq \mathbb{N}$.

La suite $d_0, d_1, \ldots$ avec $d_i \in D_i$ dénote le nombre $n = \sum_{i=0} d_i w_i$.

Exemples

Nombres binaires usuels : $D_i = \{\mathbf{0}, \mathbf{1}\}$ et $w_i = 2^i$.

Nombres en base 10 : $D_i = \{\mathbf{0}, \ldots, \mathbf{9}\}$ et $w_i = 10^i$.

Jours, heures, minutes, secondes :
$D_0 = D_1 = \{\mathbf{0}, \ldots, \mathbf{59}\}$, $D_2 = \{\mathbf{0}, \ldots, \mathbf{23}\}$, $D_3 = \mathbb{N}$
$w_0 = 1$, $w_1 = 60$, $w_2 = 60 \times 60$, $w_3 = 60 \times 60 \times 24$.

Nombres binaires redondants : $D_i = \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$ et $w_i = 2^i$.

Nombres binaires redondants

On utilise trois chiffres **0**, **1** et **2**.

Un nombre donné peut avoir plusieurs représentations.

0 :	
1 :	**1**
2 :	**01, 2**
3 :	**11**
4 :	**001, 02, 21**
5 :	**101, 12**
6 :	**011, 201, 22**
7 :	**111**
8 :	**0001, 002, 021, 211**
9 :	**1001, 102, 121**
10 :	**0101, 012, 2001, 202, 221**
11 :	**1101, 112**
12 :	**0011, 0201, 022, 2101, 212**
13 :	**1011, 1201, 122**
14 :	**0111, 2011, 2201, 222**
15 :	**1111**
16 :	**00001, 0002, 0021, 0211, 2111**
17 :	**10001, 1002, 1021, 1211**

Incrément et décrément en représentation redondante

```
let rec inc = function
  | [] -> [One]
  | Zero :: n -> One :: n
  | One :: n -> Two :: n
  | Two :: n -> One :: inc n
```

Le dernier cas s'explique par $(2 + 2n) + 1 = 1 + 2(n + 1)$.

```
let rec dec = function
  | [] -> raise Error
  | [One] -> []
  | Two :: n -> One :: n
  | One :: n -> Zero :: n
  | Zero :: n -> One :: dec n
```

Le dernier cas s'explique par $(0 + 2n) - 1 = 1 + 2(n - 1)$.

Incrément et décrément en représentation redondante

Décrémenter n'est pas l'inverse d'incrémenter !

Nombre	Incréments ↓	Décréments ↑
1	**1**	**1**
2	**2**	**01**
3	**11**	**11**
4	**21**	**001**
5	**12**	**101**
6	**22**	**011**
7	**111**	**111**
8	**211**	**0001**
9	**121**	**1001**
10	**221**	**0101**
11	**112**	**1101**
12	**212**	**0011**
13	**122**	**1011**
14	**222**	**0111**
15	**1111**	**1111**

Analyse amortie

```
let rec inc = function ... | Two :: n -> One :: inc n
let rec dec = function ... | Zero :: n -> One :: dec n
```

On classifie les chiffres **0** et **2** comme dangereux.
Seul **1** n'est pas dangereux.

On prend comme potentiel
$\Phi(n) =$ nombre de chiffres dangereux dans la liste n.

À chaque fois que `inc` ou `dec` fait un appel récursif,
Φ diminue de 1 (un **2** ou un **0** devient un **1**).

Donc `inc` et `dec` sont en temps amorti constant même lorsqu'on les entrelace.

Amortissement et persistance

Comme dans le 3e cours, on peut étendre cette complexité amortie aux cas d'utilisation persistante de nombres en utilisant des listes paresseuses.

```
type digit = Zero | One | Two
type num = digit stream

let rec inc n =
  lazy (match Lazy.force n with
        | Nil -> Cons(One, lazy Nil)
        | Cons(Zero, n) -> Cons(One, n)
        | Cons(One, n) -> Cons(Two, n)
        | Cons(Two, n) -> Cons(One, inc n))
```

Avec la méthode du banquier 2.0, on peut mettre deux unités de dette sur chaque chiffre **1** et une unité sur **0** et **2**.

Le problème du chiffre zéro

On peut avoir un nombre quelconque de zéros à la fin d'un nombre : **1** $=$ **10** $=$ **100000000000000000000**.

Cela ne change pas la complexité de `inc` et `dec`, mais rend le test à zéro arbitrairement coûteux.

```
let rec iszero = function
  | [] -> true
  | One :: _ -> false
  | Zero :: n -> iszero n
```

Le temps que prend `iszero n` n'est pas borné par une fonction du nombre dénoté par la liste…

Solution 1 : garantir qu'une liste de chiffres ne se termine jamais par **0**. (Au prix de quelques complications dans les calculs.)

Solution 2 : représenter les nombres sans utiliser de zéros !

Représentation binaire sans chiffre zéro

Par exemple avec les chiffres **1, 2, 3**.

0	
1	**1**
2	**2**
3	**11, 3**
4	**21**
5	**12, 31**
6	**22**
7	**111, 13, 32**
8	**211, 23**
9	**121, 311, 33**
10	**221**
11	**112, 131, 321**
12	**212, 231**
13	**122, 312, 331**
14	**222**
15	**1111, 113, 132, 322**
16	**2111, 213, 232**
17	**1211, 123, 3111, 313, 332**

Quelques opérations sans zéro

```
type digit = One | Two | Three
type num = digit list

let iszero = function [] -> true | _ -> false

let rec inc = function
  | [] -> [One]
  | One :: n -> Two :: n
  | Two :: n -> Three :: n
  | Three :: n -> Two :: inc n  (* (3 + 2n) + 1 = 2 + 2(n+1) *)

let rec dec = function
  | [] -> raise Error
  | [One] -> []
  | Three :: n -> Two :: n
  | Two :: n -> One :: n
  | One :: n -> Two :: dec n    (* (1 + 2n) - 1 = 2 + 2(n-1) *)
```

Représentation creuse

Au lieu de la représentation positionnelle

nombre = liste de chiffres

on peut utiliser une représentation creuse (*sparse*)

nombre = liste de (chiffre non nul, poids)

(poids strictement croissants)

ou, si les seuls chiffres utilisés sont **0** et **1**,

nombre = liste de poids (strictement croissants)

Exemple : 13 = **1011** (repr. positionnelle) = 1, 4, 8 (repr. creuse)

Incrément et décrément en représentation binaire creuse

```
type num = int list (* powers of 2, in strictly increasing order *)

let iszero = function [] -> true | _ -> false

let rec carry c n =
  match n with
  | [] -> [c]
  | w :: n' -> if c < w then c :: n else carry (2 * c) n'

let rec borrow c n =
  match n with
  | [] -> raise Error
  | w :: n' -> if c = w then n' else w :: borrow (2 * c) n'

let inc n = carry 1 n
let dec n = borrow 1 n
```

Structures de données inspirées par des systèmes de numération

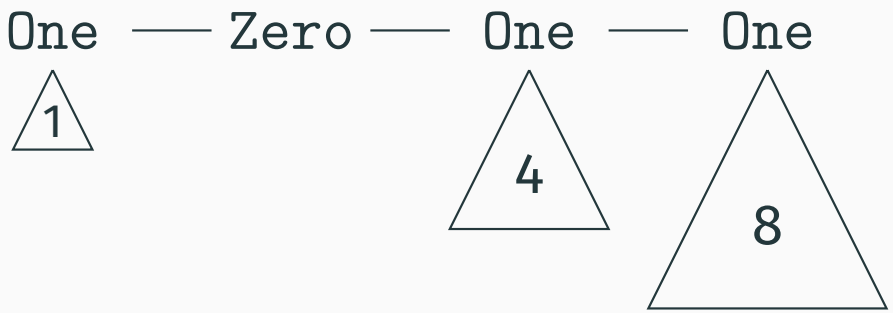
D’un système de numération à une structure de données

Idée générale :

Une structure = une liste de chiffres

Un chiffre d au rang i = d arbres de w_i éléments chacun

Exemple : en base 2 ($w_i = 2^i$), avec les chiffres **0** et **1**, une structure à 13 éléments aura la forme suivante.



Quels arbres correspondent aux poids ?

Pour une représentation en base 2, il faut des arbres de taille 2^i.

Pour ≪propager les retenues≫ pendant l'insertion (= l'incrément), il faut un moyen simple de combiner deux arbres de taille 2^i en un arbre de taille 2^{i+1}.

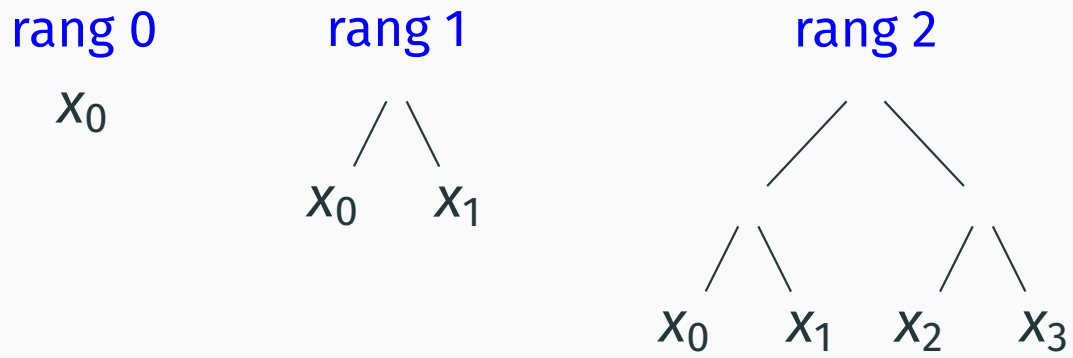
Deux exemples utilisés par la suite :

- Arbres binaires parfaits avec valeurs aux feuilles (utilisation : listes avec accès direct).
- Arbres binomiaux (utilisation : files de priorité).

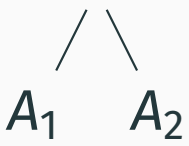
Arbres binaires parfaits avec valeurs aux feuilles

A.B.P. de rang 0 = une valeur x.

A.B.P. de rang $i+1$ = deux A.B.P. de rang i.



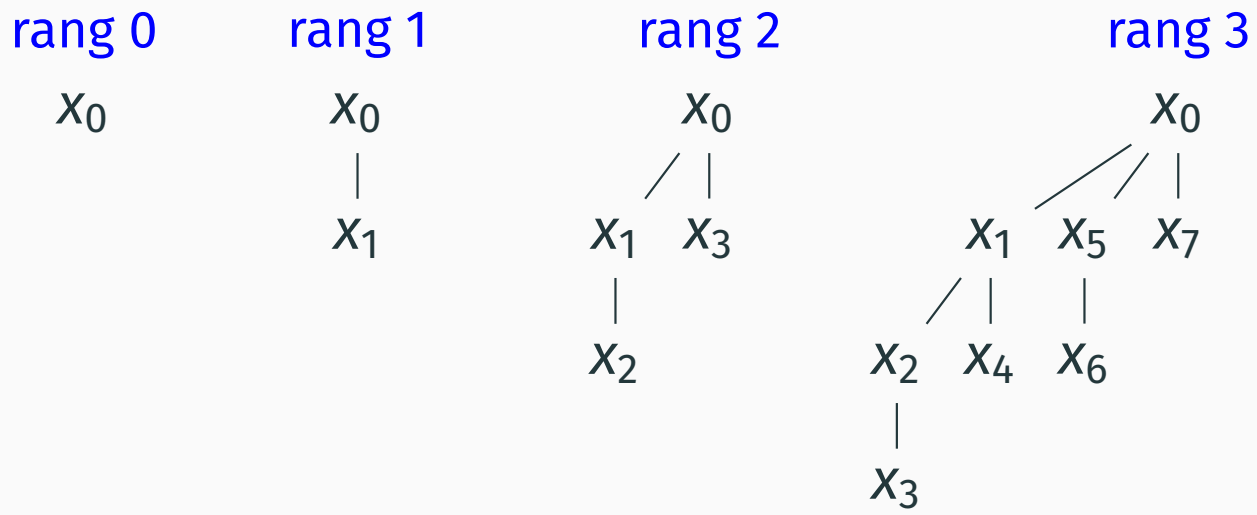
Bien adaptés à l'implémentation de séquences indicées :
l'accès à la j^{e} valeur x_j se fait en temps $i = \log n$ par dichotomie.

Combiner A_1 et A_2 de rang i :  (de rang $i+1$).

Arbres binomiaux (J. Vuillemin, 1978)

Arbre binomial de rang i =

une valeur x et i arbres binomiaux de rangs $i-1, \ldots, 1, 0$.

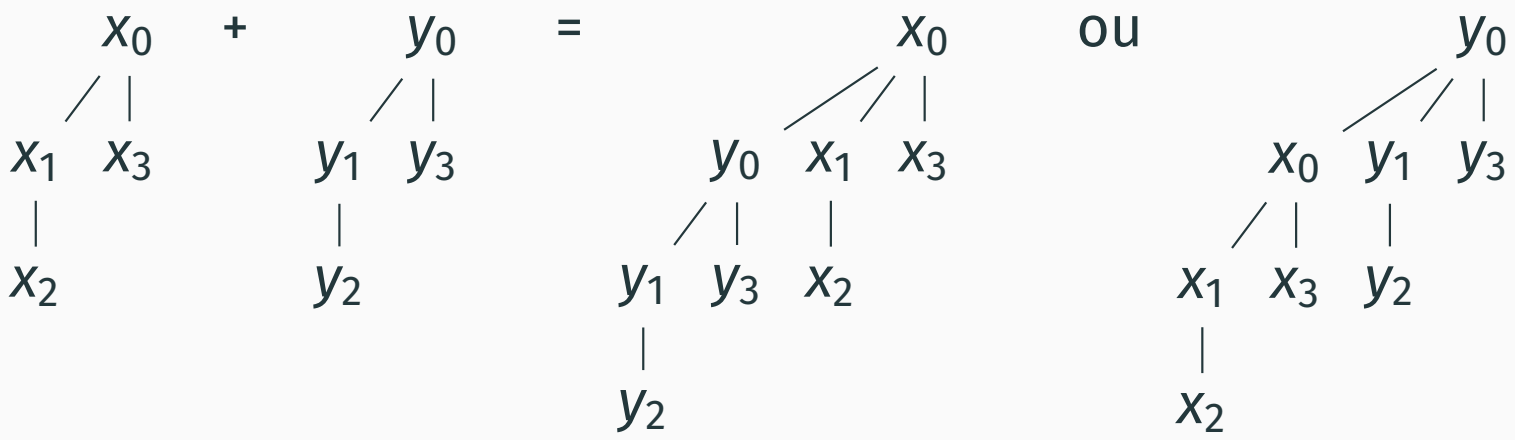


Un arbre binomial de rang i a 2^i éléments,
dont $\binom{i}{d}$ éléments à la profondeur d.

Arbres binomiaux

Combiner deux arbres binomiaux de rang i =

ajouter l'un comme premier sous-arbre de l'autre.



Une forme bien adaptée à la structure de tas (*heap*)

(avec pour chaque sous-arbre le plus petit élément à la racine).

Une structure de liste à accès direct (*random-access list*)

Mêmes opérations qu'une liste simplement chaînée :

`cons`, `head`, `tail`, `isempty`

plus un accès direct au i^{e} élément de la liste :

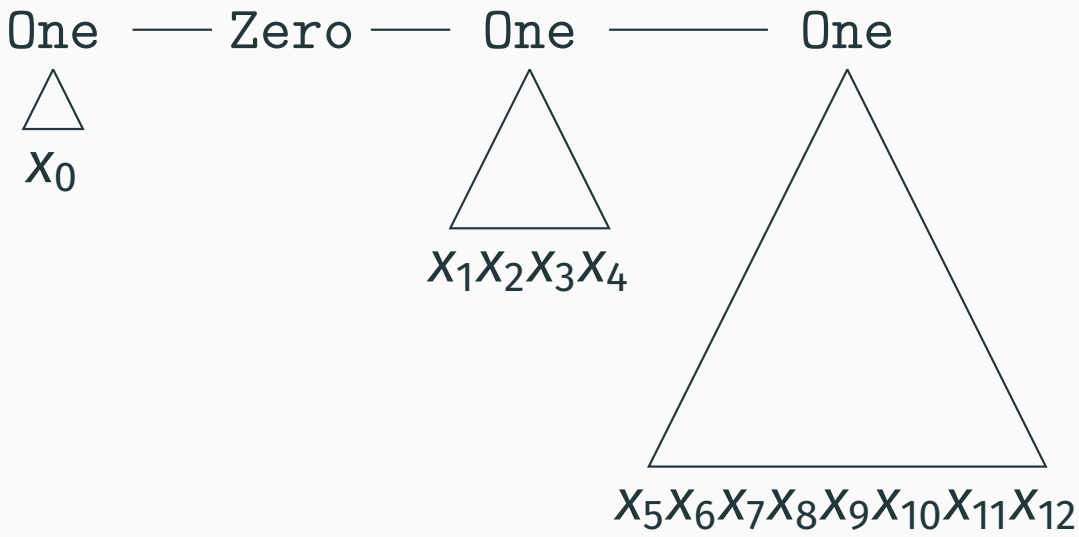
`get` $i\,\ell$, `set` $i\,v\,\ell$

Objectif de complexité : $\mathcal{O}(1)$ pour `head`, $\mathcal{O}(1)$ amorti pour `tail` et `cons`, $\mathcal{O}(\log n)$ pour `get` et `set`.

Une structure de liste à accès direct «en base 2»

On choisit une représentation structurée comme des nombres en base 2, avec chiffres **0** et **1**, et comme poids des arbres binaires parfaits avec des valeurs aux feuilles.

Exemple : la liste $[x_0, \ldots, x_{12}]$.



Remarque : pour n éléments on a $\mathcal{O}(\log n)$ chiffres.

Insertion dans la liste : l'opération `cons`

```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
type 'a digit = Zero | One of 'a tree
type 'a seq = 'a digit list

let rec cons_tree t r =
  match r with
  | [] -> [One t]
  | Zero :: r -> One t :: r
  | One t' :: r -> Zero :: cons_tree (Node(t, t')) r

let cons x r = cons_tree (Leaf x) r
```

L'opération `cons` suit le même patron que l'incrément d'un nombre en base 2.

Les opérations `head` et `tail`

```
let rec uncons_tree = function
  | [] -> raise Empty
  | [One t] -> (t, [])
  | One t :: r -> (t, Zero :: r)
  | Zero :: r ->
      let (Node(t1, t2), r') = uncons_tree r in
      (t1, One t2 :: r')

let head r =
  let (Leaf x, _) = uncons_tree r in x

let tail r =
  let (_, r') = uncons_tree r in r'
```

La fonction `uncons_tree` est similaire à la décrémentation d'un nombre en base 2, mais renvoie également le premier arbre.

Accès direct : l'opération `get`

```
let rec get_tree i t w =
  match t with
  | Leaf x -> assert (i = 0 && w = 1); x
  | Node(t1, t2) ->
      let w = w / 2 in
      if i < w then get_tree i t1 w else get_tree (i - w) t2 w

let rec get_rec i r w =
  match r with
  | [] -> raise Out_of_bounds
  | Zero :: r' -> get_rec i r' (w * 2)
  | One t :: r' ->
      if i < w then get_tree i t w
               else get_rec (i - w) r' (w * 2)

let get i r = get_rec i r 1
```

Analyse de complexité

Même analyse que pour les nombres en base 2 :

Opération	Chiffres **0**, **1**	
`head`	$\mathcal{O}(\log n)$ ✘	
`cons`, `tail`	$\mathcal{O}(\log n)$ ✘ (*)	
`get`, `set`	$\mathcal{O}(\log n)$ ✔	

(*) Une suite de n `cons` est en temps $\mathcal{O}(n)$, ainsi qu'une suite de n `tail`, mais pas une suite de n `cons` puis `tail`.

Analyse de complexité

Même analyse que pour les nombres en base 2 :

Opération	Chiffres **0**, **1**	Chiffres **1**, **2**, **3**
`head`	$\mathcal{O}(\log n)$ ✘	$\mathcal{O}(1)$ ✔
`cons`, `tail`	$\mathcal{O}(\log n)$ ✘ (*)	$\mathcal{O}(1)$ amorti ✔
`get`, `set`	$\mathcal{O}(\log n)$ ✔	$\mathcal{O}(\log n)$ ✔

(*) Une suite de n `cons` est en temps $\mathcal{O}(n)$, ainsi qu'une suite de n `tail`, mais pas une suite de n `cons` puis `tail`.

On va passer à une représentation à trois chiffres **1**, **2**, **3** :

- représentation sans zéro $\rightarrow$ `head` en $\mathcal{O}(1)$
- représentation redondante $\rightarrow$ `cons`, `tail` en $\mathcal{O}(1)$ amorti.

Redondant et sans zéro : l'opération `cons`

```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
type 'a digit =
  | One of 'a tree
  | Two of 'a tree * 'a tree
  | Three of 'a tree * 'a tree * 'a tree
type 'a seq = 'a digit list

let rec cons_tree t r =
  match r with
  | [] -> [One t]
  | One t1 :: r -> Two(t, t1) :: r
  | Two(t1, t2) :: r -> Three(t, t1, t2) :: r
  | Three(t1, t2, t3) :: r ->
      Two(t, t1) :: cons_tree (Node(t2, t3)) r

let cons x r = cons_tree (Leaf x) r
```

Redondant et sans zéro : les opérations `head` et `tail`

```
let head = function
  | [] -> raise Empty
  | One(Leaf x) :: _ -> x
  | Two(Leaf x, _) :: _ -> x
  | Three(Leaf x, _, _) -> x
  | _ -> assert false

let rec uncons_tree = function
  | [] -> raise Empty
  | [One t] -> (t, [])
  | Three(t1, t2, t3) :: r -> (t1, Two(t2, t3) :: r)
  | Two(t1, t2) :: r -> (t1, One t2 :: r)
  | One t :: r ->
      let (Node(t1, t2), r') = uncons_tree r in
      (t, Two(t1, t2) :: r')

let tail r =
  let (_, r') = uncons_tree r in r'
```

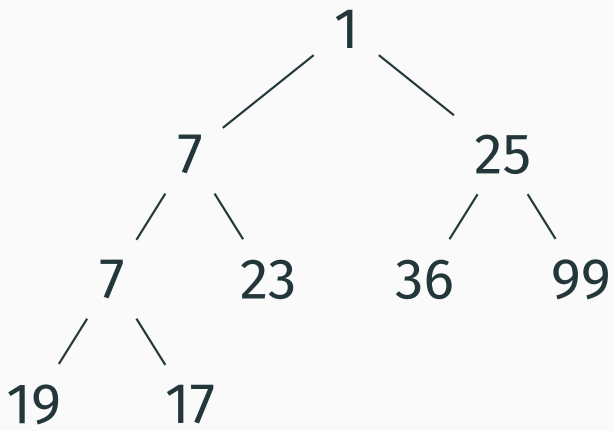
Une file de priorité (*priority queue*)

Un multi-ensemble d'éléments, avec comme principales opérations

- `insert` x h : ajouter l'élément x
- `find_min` h : renvoyer le plus petit élément de h (plus généralement : l'élément le plus prioritaire)
- `remove_min` h : supprimer le plus petit élément de h
- `merge` h_1 h_2 : renvoyer l'union de h_1 et h_2.

Utilisations : ordonnancement de tâches ; algorithmes de graphe (plus courts chemins) ; tri *heapsort*.

La structure de tas (*heap*)



Un arbre portant des valeurs aux nœuds.

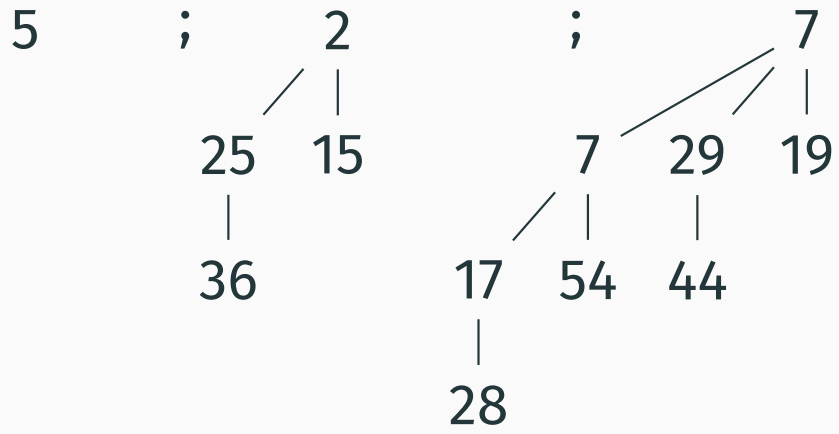
Les valeurs vont croissant le long de chaque branche.

En conséquence, la plus petite valeur est toujours au sommet.

La file de priorité binomiale (*binomial heap*) (J. Vuillemin, 1978)

Une représentation binaire creuse du nombre d'éléments dans la file de priorité, avec comme poids 2^i un arbre binomial de rang i.

Exemple : une file de priorité à 13 éléments.



La liste est ordonnée par rang (des arbres) strictement croissant.

Chaque arbre est ordonné comme un tas.

Implémentation des arbres binomiaux

```
type 'a tree = { rank: int; value: 'a; children: 'a tree list }

let link t1 t2 =
  assert (t1.rank = t2.rank);
  if t1.value <= t2.value then
    { t1 with rank = t1.rank + 1; children = t2 :: t1.children }
  else
    { t2 with rank = t2.rank + 1; children = t1 :: t2.children }
```

La combinaison de deux arbres respecte la structure de tas.

Insertion

```
type 'a heap = 'a tree list

let rec insert_tree t h =
  match h with
  | [] -> [t]
  | t' :: h' ->
      if t.rank < t'.rank
      then t :: h
      else insert_tree (link t t') h'

let insert x h =
  insert_tree { rank = 0; value = x; children = [] } h
```

Similaire à l'incrément d'un nombre binaire en représentation creuse.

Fusion de deux files

```
let rec merge h1 h2 =
  match h1, h2 with
  | [], _ -> h2
  | _, [] -> h1
  | t1 :: h1', t2 :: h2' ->
      if t1.rank < t2.rank then t1 :: merge h1' h2
      else if t2.rank < t1.rank then t2 :: merge h1 h2'
      else insert_tree (link t1 t2) (merge h1' h2')
```

Similaire à l'addition de deux nombres binaire en représentation creuse.

Extraction du plus petit élément

```
let rec extract_min = function
  | [] -> raise Empty
  | [t] -> (t, [])
  | t :: h ->
      let (t', h') = extract_min h in
      if t.value <= t'.value then (t, h) else (t', t :: h')

let find_min h =
  let (t, _) = extract_min h in t.value

let remove_min h =
  let (t, h') = extract_min h in
  merge (List.rev t.children) h'
```

Si `t` est un arbre binomial bien formé, `List.rev t.children` est une file de priorité bien formée !

Analyse de complexité

Pour une file contenant n éléments, la liste contient au plus $\log n$ arbres binomiaux

$\rightarrow$ toutes les opérations sont en temps garanti $\mathcal{O}(\log n)$

L'opération `insert` est en temps amorti $\mathcal{O}(1)$, exactement comme l'incrément d'un nombre en base 2.

(Potentiel Φ = longueur de la liste = nombre de bits à 1 dans la représentation binaire de n.)

Note : on ne peut pas avoir `insert`, `find_min` et `remove_min` en temps $\mathcal{O}(1)$ amorti, sinon on saurait trier en temps linéaire !

Types non réguliers

Types algébriques réguliers ou non réguliers

Un type algébrique avec un ou plusieurs paramètres de types est régulier si toutes les récursions se font avec les mêmes paramètres de types.

```
type 'a list = Nil | Cons of 'a * 'a list
```

Types algébriques réguliers ou non réguliers

Un type algébrique avec un ou plusieurs paramètres de types est régulier si toutes les récursions se font avec les mêmes paramètres de types.

```
type 'a list = Nil | Cons of 'a * 'a list
```

Il est non régulier ou encore emboîté (*nested*) si les récursions se font sur des paramètres de types ≪plus compliqués≫.

```
type 'a nest = Nil | Cons of 'a * ('a * 'a) nest
```

Exemple de valeur de type `int nest` :

```
Cons(1, Cons((2,3), Cons(((4,5),(6,7)), Nil))).
```

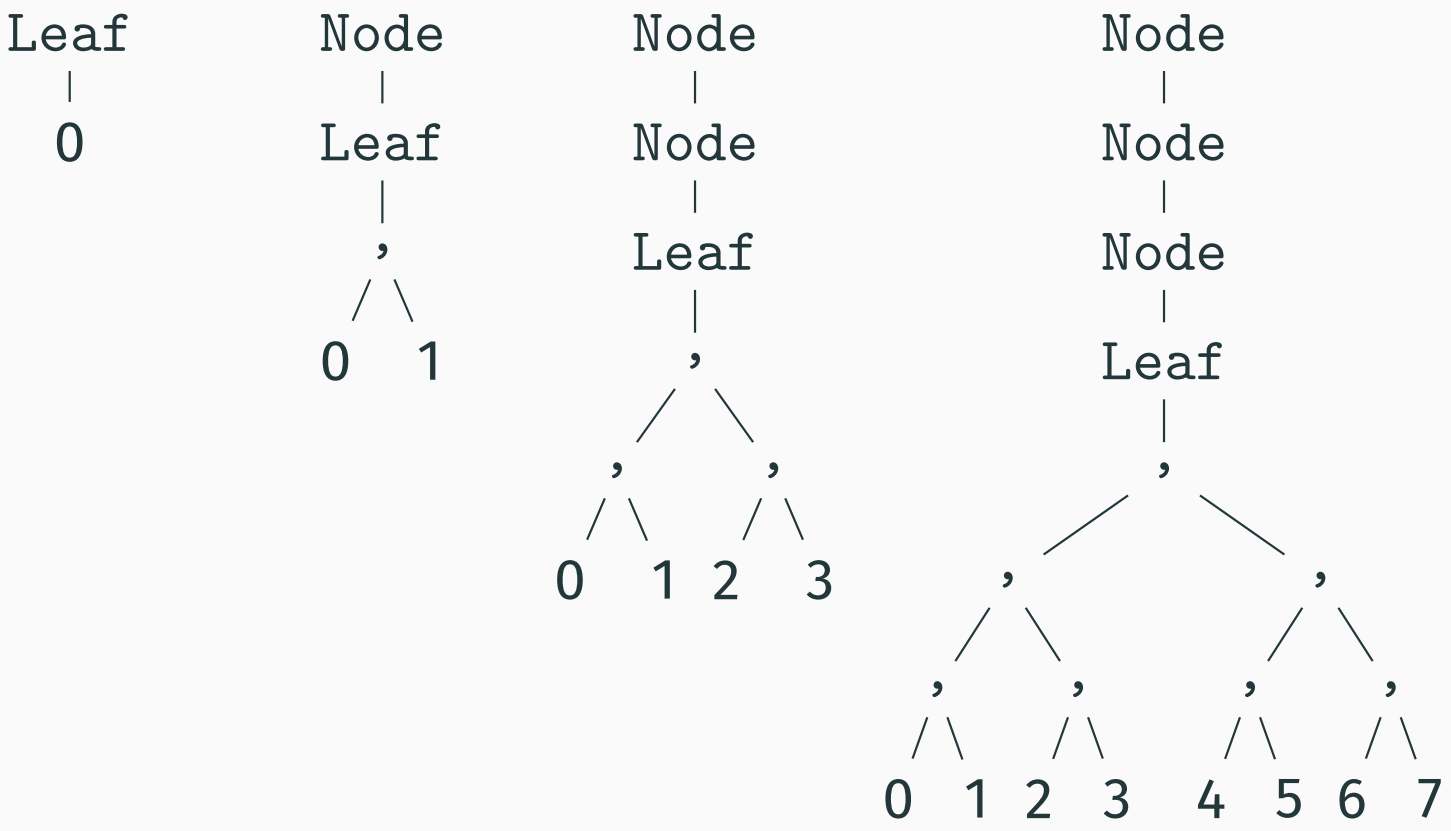
Un type non régulier : les arbres binaires parfaits

```
type 'a ptree = Leaf of 'a | Node of ('a * 'a) ptree
```

Un type non régulier : les arbres binaires parfaits

```
type 'a ptree = Leaf of 'a | Node of ('a * 'a) ptree
```

Quelques valeurs du type `int ptree` :



Quelques opérations sur les arbres binaires parfaits

```
let rec size : 'a. 'a ptree -> int = function
  | Leaf x -> 1
  | Node t -> 2 * size t

let rec leftmost : 'a. 'a ptree -> 'a = function
  | Leaf x -> x
  | Node t -> fst (leftmost t)

let rec rightmost : 'a. 'a ptree -> 'a = function
  | Leaf x -> x
  | Node t -> snd (rightmost t)
```

NB : il faut annoter les fonctions par leur type polymorphe ($\forall \alpha, \alpha$ `ptree` $\to \ldots$) car il s'agit d'une récursion polymorphe pour laquelle l'inférence de types est indécidable en général.

Comparaison avec le type régulier usuel

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

let rec size = function
  | Leaf x -> 1
  | Node(t1, t2) ->
      size t1 + size t2

let rec leftmost = function
  | Leaf x -> x
  | Node(t1, t2) -> leftmost t1
```

```
type 'a ptree =
  | Leaf of 'a
  | Node of ('a * 'a) ptree

let rec size : ... = function
  | Leaf x -> 1
  | Node t -> 2 * size t

let rec leftmost : ... = function
  | Leaf x -> x
  | Node t -> fst (leftmost t)
```

Une structure de liste à accès direct

Au lieu d'une liste ordinaire de chiffres (`digit`) portant des arbres binaires parfaits, on utilise une liste de type «nid» (`nest`) avec récursion non régulière (`'a` devient `'a * 'a`).

```
type 'a digit = Zero | One of 'a
type 'a seq = Nil | Cons of 'a digit * ('a * 'a) seq
```

Exemples de séquences à $1, \ldots, 6$ éléments : (`::` est `Cons` infixe)

```
One 1 :: Nil
Zero  :: One(2,1) :: Nil
One 3 :: One(2,1) :: Nil
Zero  :: Zero     :: One((4,3),(2,1)) :: Nil
One 5 :: Zero     :: One((4,3),(2,1)) :: Nil
Zero  :: One(6,5) :: One((4,3),(2,1)) :: Nil
```

Les opérations `cons` et `uncons`

```
let rec cons : 'a. 'a -> 'a seq -> 'a seq = fun x s ->
  match s with
  | Nil -> Cons(One x, Nil)
  | Cons(Zero, s) -> Cons(One x, s)
  | Cons(One y, s) -> Cons(Zero, cons (x, y) s)

let rec uncons : 'a. 'a seq -> 'a * 'a seq = function
  | Nil -> raise Empty
  | Cons(One x, s) -> (x, Cons(Zero, s))
  | Cons(Zero, s) ->
      let ((x, y), t) = uncons s in (x, Cons(One y, t))
```

L'accès direct : lecture

```
let rec get : 'a. int -> 'a seq -> 'a = fun i s ->
  match s with
  | Nil -> raise Out_of_bounds
  | Cons(Zero, s) -> get2 i s
  | Cons(One x, s) -> if i = 0 then x else get2 (i - 1) s

and get2 : 'a. int -> ('a * 'a) seq -> 'a = fun i s ->
  let (x0, x1) = get (i / 2) s in
  if i mod 2 = 0 then x0 else x1
```

L'accès direct : écriture, modification

Pour ≪passer la récursion≫, il faut généraliser l'écriture en une modification par une fonction `f : 'a -> 'a` quelconque.

```
let rec update : 'a. int -> ('a -> 'a) -> 'a seq -> 'a seq
= fun i f s ->
  match s with
  | Nil -> raise Out_of_bounds
  | Cons(Zero, s) -> Cons(Zero, update2 i f s)
  | Cons(One x, s) ->
      if i = 0 then Cons(One(f x), s)
               else Cons(One x, update2 (i - 1) f s)
and update2 : 'a. int -> ('a -> 'a) -> ('a * 'a) seq -> ('a * 'a) seq
= fun i f s2 ->
  let f2 (x0, x1) = if i mod 2 = 0 then (f x0, x1) else (x0, f x1) in
  update (i / 2) f2 s2

let set : 'a. int -> 'a -> 'a seq -> 'a seq = fun i v s ->
  update i (fun _ -> v) s
```

Finger trees

Les *finger trees* (Hinze et Paterson, 2006)

Une structure de type ≪séquence d'éléments≫, purement fonctionnelle, avec beaucoup d'opérations efficaces :

- Accès, insertion, suppression aux deux extrémités en temps $\mathcal{O}(1)$ amorti et $\mathcal{O}(\log n)$ garanti (file à double entrée, *deque*)
- Concaténation de deux séquences en temps $\mathcal{O}(\log n)$ (structure de cordes, *ropes*)
- Après annotation par un monoïde (voir prochain cours) :
  accès direct au i^{e} élément en temps $\mathcal{O}(\log n)$ (tableau fonctionnel)
  accès direct au plus petit élément en temps $\mathcal{O}(\log n)$ (file de priorité)

Elle combine deux techniques vues dans ce cours : systèmes de numération et types non réguliers.

Une construction progressive des *finger trees* (Claessen, 2020)

Une sorte de liste avec accès direct au premier et au dernier élément :

```
type 'a seq =
  | Nil
  | Unit of 'a
  | More of 'a * 'a seq * 'a
```

Les opérations `head` et `last` sont en temps constant, mais `cons` et `add` sont en temps linéaire :

```
let rec cons x = function
  | Nil -> Unit x
  | Unit y -> More(x, Nil, y)
  | More(y, s, z) -> More(x, cons y s, z)
```

Utilisation d'un type non régulier

Les séquences intermédiaires (`s` dans `More(x, s, y)`) seraient beaucoup plus courtes si elles contenaient des paires `'a * 'a` au lieu d'éléments simples `'a`.

```
type 'a seq =
  | Nil
  | Unit of 'a
  | More of 'a * ('a * 'a) seq * 'a
```

Problème : une séquence de longueur 3 n'est pas représentable !

Plus généralement, les longueurs de séquences représentables sont $L = \{0, 1\} \cup \{2 + 2\ell \mid \ell \in L\} = \{0, 1, 2, 4, 6, 10, 14, 22, \ldots\}$.

Utilisation de chiffres

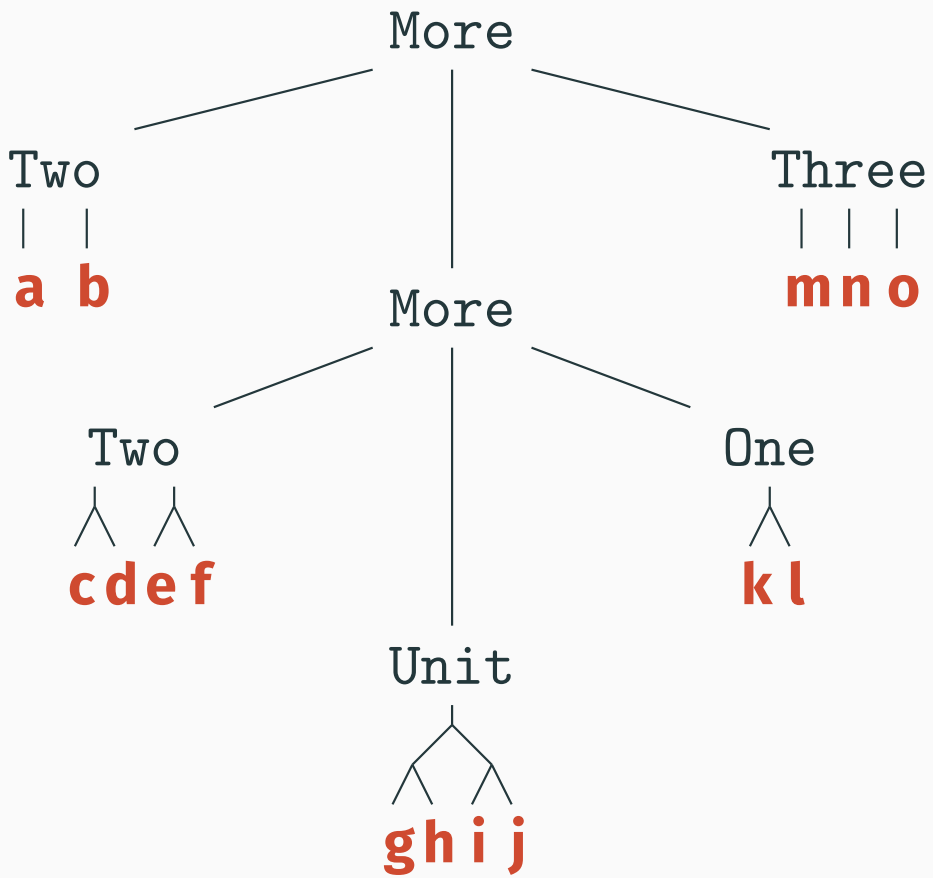
Pour pouvoir représenter toutes les longueurs, on met un chiffre (= un petit nombre d'éléments) de part et d'autre de la sous-séquence.

```
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 * 'a) seq * 'a digit
```

On reconnaît un système de numération en base 2 avec des chiffres redondants et sans zéro

$\rightarrow$ les opérations similaires à l'incrément et au décrément (`cons`, `tail`, `add`, `take`) vont être en $\mathcal{O}(1)$ amorti.

Un exemple de *finger tree*



Chacune des frontières (gauche et droite) ressemble à une des listes à accès direct vues précédemment.

L'opération `cons`

```
let rec cons : 'a. 'a -> 'a seq -> 'a seq = fun x t ->
  match t with
  | Nil -> Unit x
  | Unit y -> More(One x, Nil, One y)
  | More(One y, s, z) -> More(Two(x, y), s, z)
  | More(Two(y1, y2), s, z) -> More(Three(x, y1, y2), s, z)
  | More(Three(y1, y2, y3), s, z) ->
      More(Two(x, y1), cons (y2, y3) s, z)
```

Exercice : définir `add` (l'insertion en fin de séquence) de manière complètement symétrique.

Les opérations `head` et `tail`

```
let rec uncons : 'a. 'a seq -> 'a * 'a seq = fun t ->
  match t with
  | Nil -> raise Empty
  | Unit y -> (y, Nil)
  | More(Three(y1, y2, y3), s, z) -> (y1, More(Two(y2, y3), s, z))
  | More(Two(y1, y2), s, z) -> (y1, More(One y2, s, z))
  | More(One y, Nil, One z) -> (y, Unit z)
  | More(One y, Nil, Two(z1, z2)) -> (y, More(One z1, Nil, One z2))
  | More(One y, Nil, Three(z1, z2, z3)) ->
      (y, More(One z1, Nil, Two(z2, z3)))
  | More(One y, s, z) ->
      let ((y1, y2), s') = uncons s in (y, More(Two(y1, y2), s', z))

let head s = fst (uncons s)
let tail s = snd (uncons s)
```

La concaténation de deux séquences

Les cas de base sont faciles :

$$\texttt{concat}\ \texttt{Nil}\ s = s \qquad \texttt{concat}\ s\ \texttt{Nil} = s$$

$$\texttt{concat}\ (\texttt{Unit}\ x)\ s = \texttt{cons}\ x\ s \qquad \texttt{concat}\ s\ (\texttt{Unit}\ x) = \texttt{add}\ x\ s$$

Le cas récursif est plus délicat :

$$\texttt{concat}\ (\texttt{More}(x_1, s_1, y_1))\ (\texttt{More}(x_2, s_2, y_2)) = \texttt{More}(x_1, \texttt{??}, y_2)$$

La séquence marquée `??` doit être la concaténation de s_1, des éléments du chiffre y_1, des éléments du chiffre x_2, et de s_2.

La concaténation de deux séquences

On généralise la concaténation en une fonction `glue`

```
glue : 'a seq -> 'a list -> 'a seq -> 'a seq
```

`glue` $s_1\ \ell\ s_2$ est une séquence contenant les éléments de s_1 suivis de la (courte) liste d'éléments ℓ, suivis des éléments de s_2.

On a bien sûr `concat` $s_1\ s_2 =$ `glue` $s_1\ [\,]\ s_2$.

Le cas récursif pour `glue` est de la forme

$$\begin{aligned}&\texttt{glue}\ (\texttt{More}(x_1, s_1, y_1))\ \ell\ (\texttt{More}(x_2, s_2, y_2))\\ &\quad = \quad \texttt{More}(x_1, \texttt{glue}\ s_1\ (\texttt{elements}\ y_1\ \texttt{@}\ \ell\ \texttt{@}\ \texttt{elements}\ x_1)\ s_2, y_2)\end{aligned}$$

où `@` est la concaténation de listes usuelle

et `elements: 'a digit -> 'a list`.

La concaténation de deux séquences

```
glue : 'a seq -> 'a list -> 'a seq -> 'a seq
```

$\texttt{glue}\,(\texttt{More}(x_1, s_1, y_1))\,\ell\,(\texttt{More}(x_2, s_2, y_2)) = \texttt{More}(x_1, \texttt{glue}\,s_1\,\ell'\,s_2, y_2)$

avec $\ell' = \texttt{elements}\,y_1\,\texttt{@}\,\ell\,\texttt{@}\,\texttt{elements}\,x_2$.

Erreur de typage : ℓ' est une liste d'éléments (type `'a list`) alors que l'appel récursif à `glue` attend une liste de paires d'éléments `('a * 'a) list`.

Erreur de conception : ℓ' peut avoir un nombre impair d'éléments! On ne peut donc pas la concaténer avec les séquences de paires d'éléments s_1, s_2.

Une récursion non structurelle plus souple

On va utiliser des sous-séquences contenant non seulement des paires `'a * 'a` mais aussi des triplets `'a * 'a * 'a`.

```
type 'a node = Pair of 'a * 'a | Triple of 'a * 'a * 'a
type 'a seq =
  | Nil
  | Unit of 'a
  | More of 'a digit * 'a node seq * 'a digit
```

Cela rappelle les arbres 2-3 vus au 2e cours : des arbres parfaits avec des nœuds de degré 2 ou 3.

Les opérations `cons`, `uncons`, `add`, `unadd` s'étendent facilement (exercice).

La concaténation de deux séquences

$\texttt{glue}\,(\texttt{More}(x_1, s_1, y_1))\,\ell\,(\texttt{More}(x_2, s_2, y_2)) = \texttt{More}(x_1, \texttt{glue}\,s_1\,\ell'\,s_2, y_2)$

avec $\ell' = \texttt{to\_nodes}(\texttt{elements}\,y_1\,\texttt{@}\,\ell\,\texttt{@}\,\texttt{elements}\,x_2)$.

`to_nodes` prend une liste d'éléments de longueur $\neq 1$ et la transforme en liste de nœuds `Pair` et `Triple` :

```
let rec to_nodes = function
  | [] -> []
  | [x] -> assert false
  | [x1; x2] -> [Pair(x1, x2)]
  | [x1; x2; x3; x4] -> [Pair(x1, x2); Pair(x3, x4)]
  | x1 :: x2 :: x3 :: xs -> Triple(x1, x2, x3) :: to_nodes xs
```

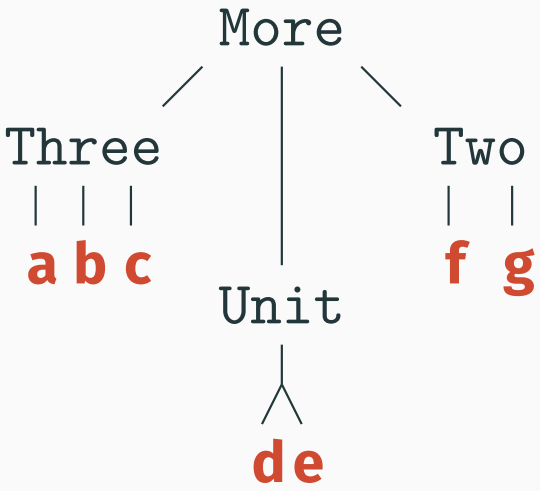
Si ℓ a entre 0 et 3 éléments, l'argument de `to_nodes` a entre 2 et 9 éléments, et ℓ' a entre 1 et 3 éléments.

Le code complet de la concaténation

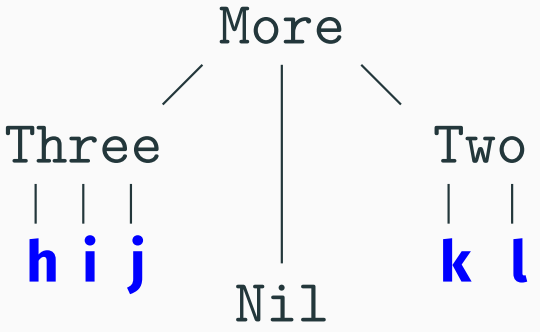
```
let elements = function
  | One x -> [x]
  | Two(x1, x2) -> [x1; x2]
  | Three(x1, x2, x3) -> [x1; x2; x3]

let rec glue: 'a. 'a seq -> 'a list -> 'a seq -> 'a seq = fun s1 a s2 ->
  match s1, s2 with
  | Nil, _ -> List.fold_right cons a s2
  | _, Nil -> List.fold_left (Fun.flip add) s1 a
  | Unit x1, _ -> List.fold_right cons (x1 :: a) s2
  | _, Unit x2 -> List.fold_left (Fun.flip add) s1 (a @ [x2])
  | More(x1, s1, y1), More(x2, s2, y2) ->
      More(x1, glue s1 (to_nodes (elements y1 @ a @ elements x2)) s2, y2)

let concat s1 s2 = glue s1 [] s2
```

Complexité $\mathcal{O}(\min(\log n_1, \log n_2))$.

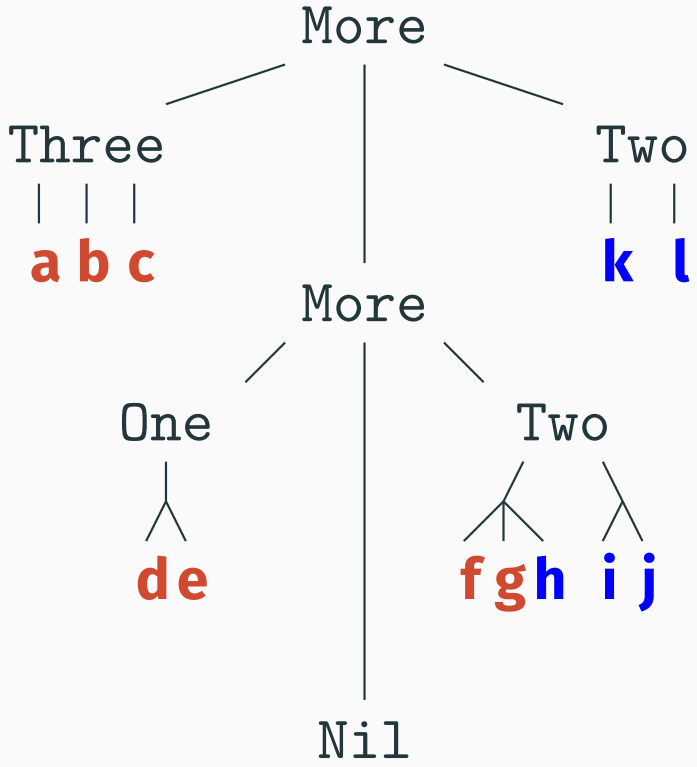
Un exemple de concaténation



++



=



Point d’étape

Point d'étape

Les systèmes de numération sont des ≪patrons de conception≫ pour des structures de données de type ≪liste≫ et cependant efficaces car la taille des éléments de la liste augmente exponentiellement.

(C'est ce que C. Okasaki appelle *implicit recursive slowdown*.)

L'utilisation de types algébriques non réguliers permet de refléter dans les types certains invariants sur les tailles, et guide encore plus l'écriture du code.

Les *finger trees* sont simples, mais on peut obtenir de meilleures performances avec des structures plus complexes.

(Kaplan & Tarjan 1996, 1999 : toutes opérations en $\mathcal{O}(1)$;
voir aussi le séminaire d'Arthur Charguéraud le 13 avril.)

Pour aller plus loin : *data structural bootstrapping*

(A. Buchsbaum, PhD Princeton, 1995.)

Un ensemble de techniques permettant d'obtenir de bonnes structures de données à partir de structures plus simples mais moins efficaces ou aux fonctionnalités limitées.

Dans ce cours nous avons vu un exemple de *bootstrapping* :

- Conteneurs de capacité fixe (paires, chiffres)
  $\rightarrow$ conteneur de capacité arbitraire (séquence).

Okasaki (chap. 10) montre d'autres exemples :

- Ajouter des opérations manquantes
  (ex : file d'attente $\rightarrow$ liste avec concaténation)
- Diminuer la complexité de certaines opérations
  (ex : tas avec `merge` en $\mathcal{O}(\log n)$ $\rightarrow$ tas avec `merge` en $\mathcal{O}(1)$)

Bibliographie

Bibliographie

Le principal support de ce cours :

- Chris Okasaki, *Purely Functional Data Structures*, chapitres 9 et 11.

L'article original sur les *finger trees* :

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

Une présentation plus facile d'accès :

- Koen Claessen, *Finger trees explained anew, and slightly simplified*, Haskell symposium 2020.