Comment allier persistance et performance

Arthur Charguéraud

Inria

Séminaire au Collège de France

13 avril 2023

Persistance

Une opération sur une structure persistante fournit une version modifiée de la structure, sans invalider les versions précédentes.

```
get : 'a parray -> i -> 'a
set : 'a parray -> i -> 'a -> 'a parray
```

Persistance

Une opération sur une structure persistante fournit une version modifiée de la structure, sans invalider les versions précédentes.

```
get : 'a parray -> i -> 'a
set : 'a parray -> i -> 'a -> 'a parray
```

Bénéfices :

1. Facilite grandement le raisonnement sur les programmes
2. Simplifie le code en évitant de défaire des modifications (*backtrack*)
3. Peut éviter de faire des copies coûteuses de structures éphémères

Persistance

Une opération sur une structure persistante fournit une version modifiée de la structure, sans invalider les versions précédentes.

```
get : 'a parray -> i -> 'a
set : 'a parray -> i -> 'a -> 'a parray
```

Bénéfices :

1. Facilite grandement le raisonnement sur les programmes
2. Simplifie le code en évitant de défaire des modifications (*backtrack*)
3. Peut éviter de faire des copies coûteuses de structures éphémères

Limitation : les structures persistantes ont des facteurs constants plus importants que les structures éphémères correspondantes.

Comment optimiser les structures persistantes ?

Trois pistes :

1. Améliorer les implémentations purement fonctionnelles
   $\rightarrow$ en optimisant la représentation en mémoire
2. Utiliser des effets de bords pour réaliser la persistance
   $\rightarrow$ pour optimiser les opérations persistantes sur la dernière version
3. Modifier l'interface : versions transitoires non persistantes
   $\rightarrow$ la *transience*, pour réduire le nombre d'opérations persistantes

Plan de d'exposé

1. Optimiser les listes pures, avec des cellules à plusieurs éléments
2. Optimiser les arbres purs, avec des arbres d'arité K
3. Exploiter les effets, avec une variante des tableaux semi-persistants
4. Exploiter la transience, et intégrer le tout dans Sek

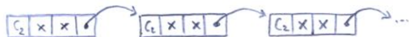
Partie I

Optimiser la représentation des listes pures

Est-il intéressant de grouper les éléments ?

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

```
type 'a list2 =
  | Nil
  | Cons1 of 'a * 'a list2
  | Cons2 of 'a * 'a * 'a list2
```



Effet sur la manipulation de listes de longueur fixe

```
let a = List.init length (fun i -> i) in
let b = List.map (fun x -> x + 1) a in
let r = ref 0 in
List.iter (fun x -> r := !r + x) b;
```

Effet sur la manipulation de listes de longueur fixe

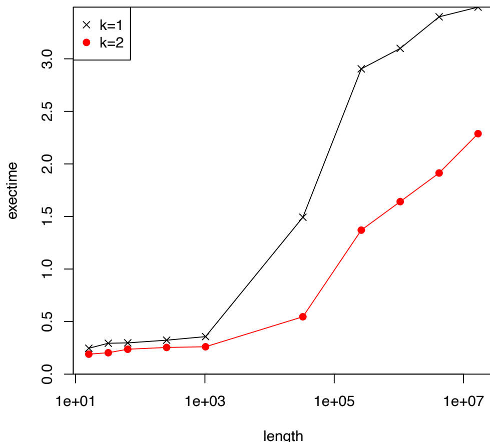
```
let a = List.init length (fun i -> i) in
let b = List.map (fun x -> x + 1) a in
let r = ref 0 in
List.iter (fun x -> r := !r + x) b;
```

On répète le tout `N/length` fois, de sorte à toujours faire le même nombre d'opérations au total.

Effet sur la manipulation de listes de longueur fixe

```
let a = List.init length (fun i -> i) in
let b = List.map (fun x -> x + 1) a in
let r = ref 0 in
List.iter (fun x -> r := !r + x) b;
```

On répète le tout `N/length` fois, de sorte à toujours faire le même nombre d'opérations au total.



Jusqu'à quel point regrouper les éléments ?

```
type 'a listK =
  | Nil
  | Cons1 of 'a * 'a listK
  | Cons2 of 'a * 'a * 'a listK
  | ..
  | ConsK of 'a * 'a * .. * 'a * 'a listK
```

Consommation mémoire :

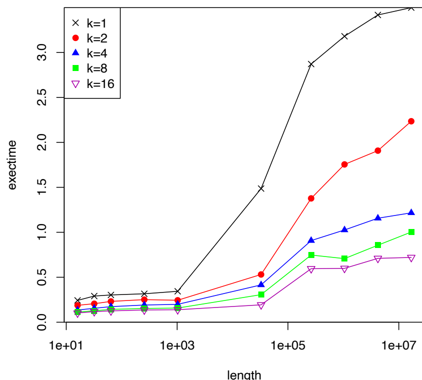
- $3n$ pour `list`
- $(1 + \frac{2}{K}) \cdot n$ pour `listK`
  $\rightarrow$ $1{,}125n$ pour $K{=}16$

Jusqu'à quel point regrouper les éléments ?

```
type 'a listK =
  | Nil
  | Cons1 of 'a * 'a listK
  | Cons2 of 'a * 'a * 'a listK
  | ..
  | ConsK of 'a * 'a * .. * 'a * 'a listK
```

Consommation mémoire :

- $3n$ pour `list`
- $(1 + \frac{2}{K}) \cdot n$ pour `listK`
  $\rightarrow$ $1{,}125n$ pour $K{=}16$



Code : ça pique les yeux !

```
let init n f =
  let rec aux i =
    if i >= n then Nil
    else if i = n-1 then
      let x1 = f(i) in
      Cons1 (x1, Nil)
    else if i = n-2 then
      let x1 = f(i) in
      let x2 = f(i+1) in
      Cons2 (x1, x2, Nil)
      ...
    else (* i <= n-K *)
      let x1 = f(i) in
      let x2 = f(i+1) in
      let .. in
      let xK = f(i+K) in
      let r = aux (i+K+1) in
      ConsK (x1, x2, .., xK, r)
    in
  aux 0
```

```
let rec map f s =
  match s with
  | Nil -> Nil
  | Cons1(x1, r) ->
    let y1 = f x1 in
    let t = map f r in
    Cons1(y1, t)
  | Cons2(x1, x2, r) ->
    let y1 = f x1 in
    let y2 = f x2 in
    let t = map f r in
    Cons2(y1, y2, t)
  ..
  | ConsK(x1, x2, .., xK, r) ->
    let y1 = f x1 in
    let y2 = f x2 in
    ..
    let yK = f xK in
    let t = map f r in
    ConsK(y1, y2, .. , yK, t)
```

Note pour les experts : l'optimisation *tail-modulo-cons* reste applicable.

Code : mieux vaudrait le générer automatiquement

Avec de la méta-programmation ou bien des transformations de code.

Travail en cours sur des scripts de transformation : projet OptiTrust.

Code d'entrée

```
type 'a listA =
  | Nil
  | ConsK of 'a array * 'a listA

let rec map f s =
  match s with
  | Nil -> Nil
  | ConsK(a, r) ->
      let b = Array.map f a in
      let t = map f r in
      ConsK(b, t)
```

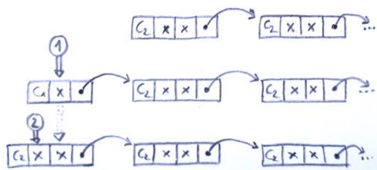
Code de sortie

```
type 'a listK =
  | Nil
  | Cons1 of 'a * 'a listK
  | Cons2 of 'a * 'a * 'a listK
  | ..
  | ConsK of 'a * 'a * .. * 'a
             * 'a listK

let rec map f s =
  match s with
  | Nil -> Nil
  | Cons1(x1, r) -> ..
  ..
  | ConsK(x1, x2, .., xK, r) -> ..
```

Opérations sur des piles

Lorsque la longueur de la séquence n'est pas connue d'avance, le bénéfice de regrouper les éléments est moins évident.

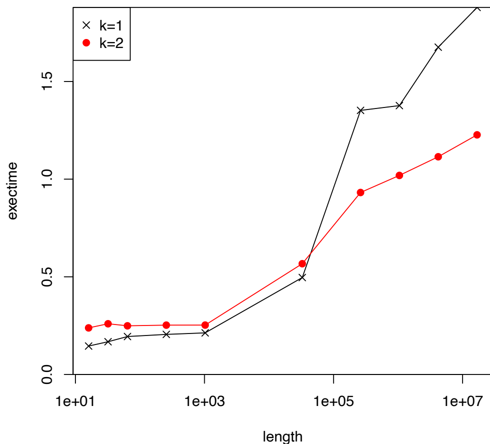


Deux insertions successives

- avec liste standard : alloue 3 mots, puis 3 mots
- avec liste `Cons2` : alloue 3 mots, puis 4 mots

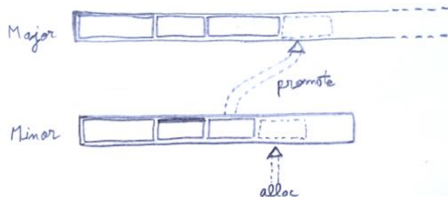
Impact du groupement sur la performance des piles

Test avec `push` de `length` éléments, suivi d'un appel à `iter`.
Le tout répété `N/length` fois, de sorte à faire `N` `push` en tout.



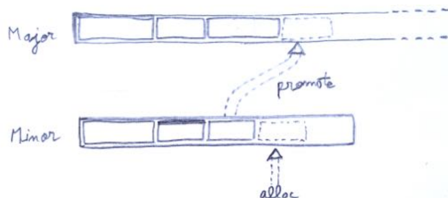
Vie des blocs mémoire en OCaml

Allocation dans le tas mineur, puis promotion possible dans le tas majeur.



Vie des blocs mémoire en OCaml

Allocation dans le tas mineur, puis promotion possible dans le tas majeur.

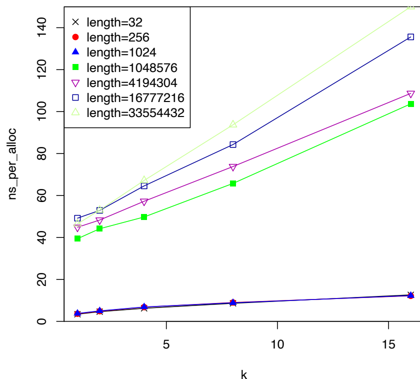


Par ailleurs, les données du tas majeur peuvent sortir des caches L2/L3. Récupérer ensuite ces données dans la mémoire principale est coûteux.

Les effets du GC et du cache se superposent.

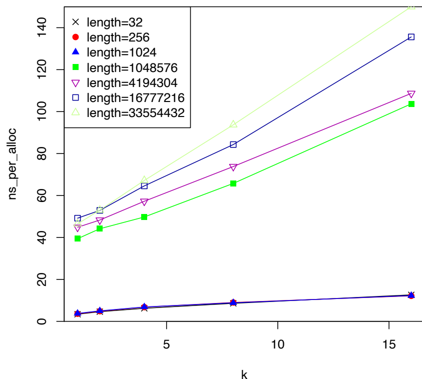
Étude expérimentale du coût de l'allocation

Étude du coût d'allocation de blocs de taille $K + 1$
en fonction de la durée de vie des blocs, contrôlée par `length`



Étude expérimentale du coût de l'allocation

Étude du coût d'allocation de blocs de taille $K + 1$
en fonction de la durée de vie des blocs, contrôlée par `length`



Modèle de coût pour un bloc de taille s :

- Allocation ne sortant pas du tas mineur coûte : $as + b$
- Allocation avec promotion dans le tas majeur coûte : $As + B$

Analyse de coût pour une pile dans le tas majeur

On compte $as + b$ pour tas mineur et $As + B$ pour tas majeur.

Comparaison d'une séquence du `push` :

(1) Allouer deux fois 3 mots dans le tas majeur
$\rightarrow\ 6A + 2B$

(2) Allouer 3 mots dans le tas mineur, plus 4 mots dans le tas majeur
$\rightarrow\ 4A + B + 3a + b$

Analyse de coût pour une pile dans le tas majeur

On compte $as + b$ pour tas mineur et $As + B$ pour tas majeur.

Comparaison d'une séquence du `push` :

(1) Allouer deux fois 3 mots dans le tas majeur
$\rightarrow\ 6A + 2B$

(2) Allouer 3 mots dans le tas mineur, plus 4 mots dans le tas majeur
$\rightarrow\ 4A + B + 3a + b$

Différence de coût : $(2A - 3a) + (B - b)$.
En pratique, $A > 1{,}5\,a$ et $B > b$, donc (2) est plus performante.

Piles avec cellules multi-éléments

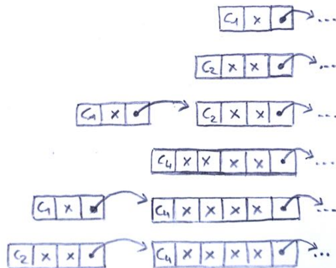
Idée : faire progresser des tailles des cellules selon les multiples de 2 comme pour les tableaux redimensionnables, mais jusqu'à une taille K.

```
type 'a listV =
  | Nil
  | Cons1 of 'a * 'a listV
  | Cons2 of 'a * 'a * 'a listV
  | Cons4 of 'a * 'a * 'a * 'a * 'a listV
  ..
  | ConsK of ...
```

Piles avec cellules multi-éléments

Idée : faire progresser des tailles des cellules selon les multiples de 2 comme pour les tableaux redimensionnables, mais jusqu'à une taille K.

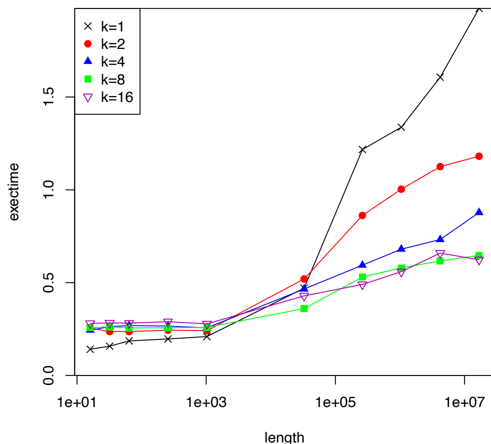
```
type 'a listV =
  | Nil
  | Cons1 of 'a * 'a listV
  | Cons2 of 'a * 'a * 'a listV
  | Cons4 of 'a * 'a * 'a * 'a * 'a listV
  ..
  | ConsK of ...
```



```
let push x0 = function
  | Cons1(x1,Cons2(x2,x3,Cons4(x4,x5,x6,x7,t))) ->
     Cons8(x0,x1,x2,x3,x4,x5,x6,x7,t)
  | Cons1(x1,Cons2(x2,x3,t)) -> Cons4(x0,x1,x2,x3,t)
  | Cons1(x1,t) -> Cons2(x0,x1,t)
  | t -> Cons1(x0,t)
```

Performances des piles avec cellules multi-éléments

Test avec `push` de `length` éléments, suivi d'un appel à `iter`.



Optimisations des opérations pop

Idée : éviter de réallouer des gros blocs lors des pop, grâce à Drop(p,s).

```
type 'a listV =
 | Nil
 | Cons1 of 'a * 'a listV
 | Cons2 of 'a * 'a * 'a listV
 | Cons4 of 'a * 'a * 'a * 'a
           * 'a listV
 | Cons8 of ..
 | Drop of int * 'a listV
```

Optimisations des opérations `pop`

Idée : éviter de réallouer des gros blocs lors des `pop`, grâce à `Drop(p,s)`.

```
type 'a listV =
 | Nil
 | Cons1 of 'a * 'a listV
 | Cons2 of 'a * 'a * 'a listV
 | Cons4 of 'a * 'a * 'a * 'a
          * 'a listV
 | Cons8 of ..
 | Drop of int * 'a listV
```

```
let pop s =
  match s with
  | Nil -> raise Not_found
  | Cons1(x,r) -> x, r
  | Cons2(x,y,r) ->
     x, Cons1(y,r)
  | Cons4(x,_,_,_) as t ->
     x, Drop(1,t)
  | Drop(1,(Cons4(_,x,_,_,r) as t)) ->
     x, Drop(2,t)
  | Drop(2,Cons4(_,_,x,y,r)) ->
     x, Cons1(y,r)
```

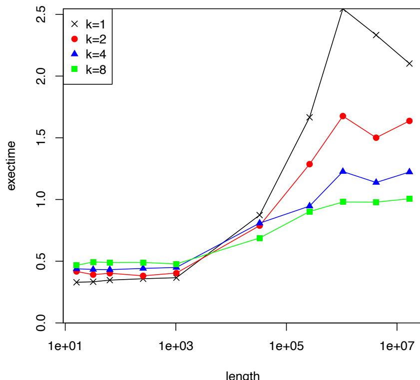
Optimisations des opérations pop

Idée : éviter de réallouer des gros blocs lors des pop, grâce à Drop(p,s).

```
type 'a listV =
  | Nil
  | Cons1 of 'a * 'a listV
  | Cons2 of 'a * 'a * 'a listV
  | Cons4 of 'a * 'a * 'a * 'a
             * 'a listV
  | Cons8 of ..
  | Drop of int * 'a listV

let pop s =
  match s with
  | Nil -> raise Not_found
  | Cons1(x,r) -> x, r
  | Cons2(x,y,r) ->
     x, Cons1(y,r)
  | Cons4(x,_,_,_,_) as t ->
     x, Drop(1,t)
  | Drop(1,(Cons4(_,x,_,_,r) as t)) ->
     x, Drop(2,t)
  | Drop(2,Cons4(_,_,x,y,r)) ->
     x, Cons1(y,r)
```

Test avec push de length éléments, suivi d'autant d'appels à pop.



Conclusions de la partie I

Utilisation de cellules de liste d'arité K :

- Toujours plus rapide si on connaît la taille au départ
- Un peu plus lent si on construit uniquement des petites piles
- La consommation mémoire passe de $3n$ à $(1 + \frac{2}{K}) \cdot n$
  - les données sortent moins vite du tas mineur
  - les données sortent moins vite des caches
  - moins de traffic sur le bus mémoire pour récupérer les données

Conclusions de la partie I

Utilisation de cellules de liste d'arité K :

- Toujours plus rapide si on connaît la taille au départ
- Un peu plus lent si on construit uniquement des petites piles
- La consommation mémoire passe de $3n$ à $(1 + \frac{2}{K}) \cdot n$
  - les données sortent moins vite du tas mineur
  - les données sortent moins vite des caches
  - moins de traffic sur le bus mémoire pour récupérer les données
- Les fonctions `length` et `nth` peuvent aller jusqu'à K fois plus vite
- Certains appels à `iter` ou `map` peuvent, en théorie, être vectorisés

Partie II

Optimiser la représentation des arbres purs

Arbres binaires

Données dans les nœuds

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

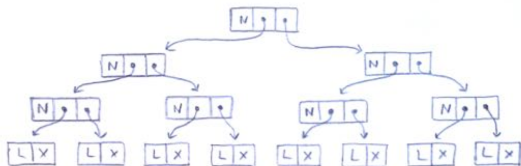
Données dans les feuilles

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

Arbres binaires

Données dans les nœuds

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



Données dans les feuilles

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

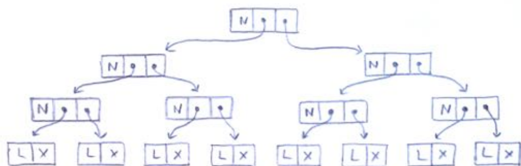
Arbres binaires

Données dans les nœuds

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

Données dans les feuilles

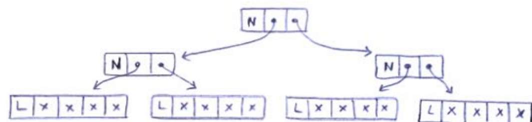
```
type 'a t =
  | Leaf of 'a
  | Node of 'a t * 'a t
```



- Mémoire : $5n$ — n nœuds à 3 mots, n feuilles à 2 mots
  $\rightarrow$ impact sur le coût des accès mémoire
- Hauteur : $\log_2 n$ — 32 niveaux pour 4 milliards d'éléments
  $\rightarrow$ impact sur le coût de `get`
- Mise à jour : $\log_2 n - 1$ allocations de 3 mots, et une de 2 mots
  $\rightarrow$ impact sur le coût de `set`

Arbres aux feuilles lourdes

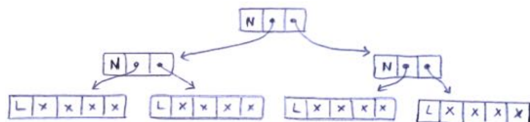
Grouper les feuilles par paquet de K. Par exemple avec $K = 4$:




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

Arbres aux feuilles lourdes

Grouper les feuilles par paquet de K. Par exemple avec $K = 4$:



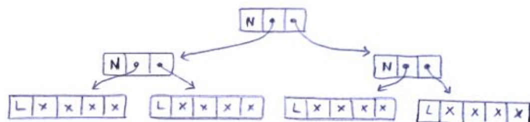

```
type 'a t =
  | Leaf of 'a * 'a * 'a * 'a
  | Node of 'a t * 'a t
```

- Mémoire : $(1 + \frac{4}{K}) \cdot n$ $\quad$ $1{,}25\,n$ mots pour $K{=}16$

$\frac{n}{K}$ noeuds à 3 mots, $\frac{n}{K}$ feuilles à $K + 1$ mots

Arbres aux feuilles lourdes

Grouper les feuilles par paquet de K. Par exemple avec $K = 4$:



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

- Mémoire : $(1 + \frac{4}{K}) \cdot n$ $1{,}25\, n$ mots pour $K{=}16$

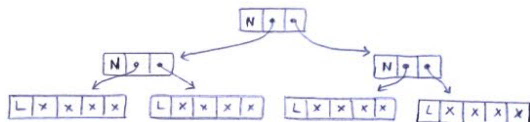
  $\frac{n}{K}$ noeuds à 3 mots, $\frac{n}{K}$ feuilles à $K + 1$ mots

- Hauteur : $\log_2 \frac{n}{K} = \log_2 n - \log_2 K$

  pour $K{=}16$, la hauteur diminue de 4

Arbres aux feuilles lourdes

Grouper les feuilles par paquet de K. Par exemple avec $K = 4$:




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

- Mémoire : $(1 + \frac{4}{K}) \cdot n$ $\quad$ $1{,}25\,n$ mots pour $K{=}16$

  $\frac{n}{K}$ noeuds à 3 mots, $\frac{n}{K}$ feuilles à $K + 1$ mots

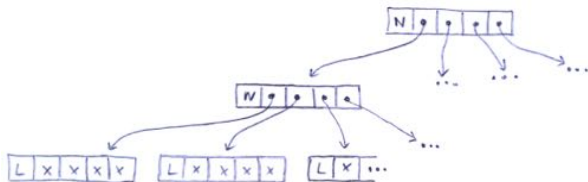
- Hauteur : $\log_2 \frac{n}{K} \;=\; \log_2 n - \log_2 K$

  pour $K{=}16$, la hauteur diminue de 4

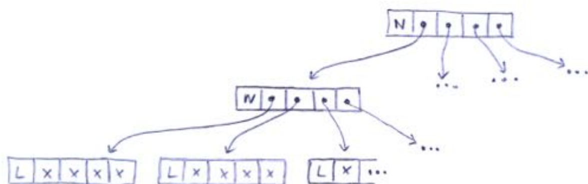
- Mise à jour : $\log_2 \frac{n}{K}$ allocations de 3 mots, et une de $K + 1$ mots

  pour $K = 16$ et $n = 2^{20}$, passe de 20 à 16 allocs, de 59 à 65 mots

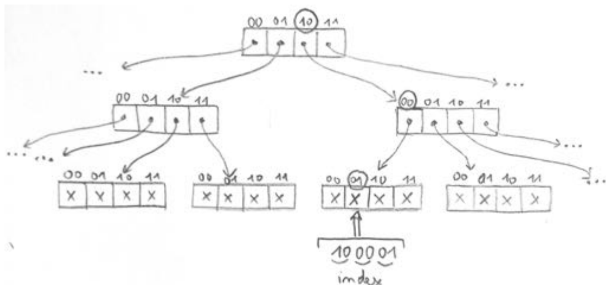
Arbres K-aires



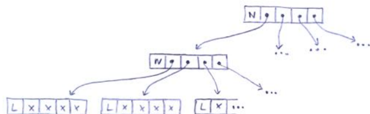
Arbres K-aires



Indexation des éléments lorsque K est une puissance de 2 :



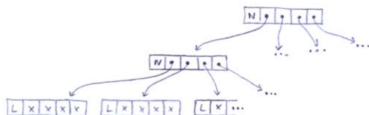
Analyse des arbres K-aires



- Mémoire : $(1 + \frac{2}{K}) \cdot n$ $\qquad$ $1{,}0625\, n$ pour $K = 16$

  $\frac{n}{K}$ nœuds et $\frac{n}{K}$ feuilles, à $K + 1$ mots

Analyse des arbres K-aires



- Mémoire : $(1 + \frac{2}{K}) \cdot n$ $\quad$ $1{,}0625\, n$ pour $K = 16$

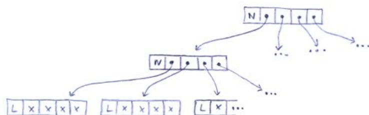
  $\frac{n}{K}$ nœuds et $\frac{n}{K}$ feuilles, à $K + 1$ mots

- Hauteur : $\log_K n = \frac{\log_2 n}{\log_2 K}$

  K=16 : on a h=4 jusqu'à 65k éléments, h=8 jusqu'à 4 milliards

  K=256 : on a h=2 jusqu'à 65k éléments, h=4 jusqu'à 4 milliards

Analyse des arbres K-aires



- Mémoire : $(1 + \frac{2}{K}) \cdot n$ $\quad$ $1{,}0625\,n$ pour $K = 16$

  $\frac{n}{K}$ nœuds et $\frac{n}{K}$ feuilles, à $K+1$ mots

- Hauteur : $\log_K n = \frac{\log_2 n}{\log_2 K}$

  $K{=}16$: on a $h{=}4$ jusqu'à 65k éléments, $h{=}8$ jusqu'à 4 milliards

  $K{=}256$: on a $h{=}2$ jusqu'à 65k éléments, $h{=}4$ jusqu'à 4 milliards

- Mise à jour : $\log_K n$ allocations de $K+1$ mots

Analyse de l'empreinte mémoire d'une mise à jour

- Arbre binaire : $\log_2 n$ allocations de 3 mots (sauf une de 2 mots)
  $n{=}2^{24}$: il faut 24 allocations, au total 71 mots
- Arbre K-aire : $\log_K n$ allocations de $K+1$ mots
  $n{=}2^{24}$ et $K{=}8$: il faut 8 allocations, au total 72 mots
  $n{=}2^{24}$ et $K{=}16$: il faut 6 allocations, au total 102 mots

Analyse de l'empreinte mémoire d'une mise à jour

- Arbre binaire : $\log_2 n$ allocations de 3 mots (sauf une de 2 mots)
  $n{=}2^{24}$: il faut 24 allocations, au total 71 mots
- Arbre K-aire : $\log_K n$ allocations de $K+1$ mots
  $n{=}2^{24}$ et $K{=}8$: il faut 8 allocations, au total 72 mots
  $n{=}2^{24}$ et $K{=}16$: il faut 6 allocations, au total 102 mots

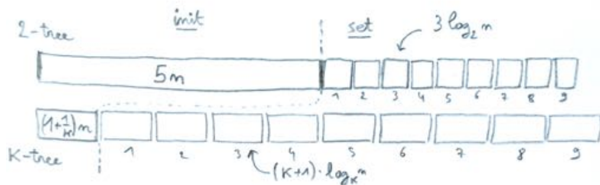
$$\frac{\text{mots alloués pour set sur arbres } K\text{-aires}}{\text{mots alloués pour set sur arbres binaires}} = \frac{(K+1)\cdot\log_K n}{3\cdot\log_2 n} = \frac{K+1}{3\cdot\log_2 K}$$

K	2	4	8	16	32	64	128	256
ratio	1,00	0,83	1,00	1,42	2,20	3,61	6,14	10,7

Pour $K \leqslant 8$, les arbres K-aires consomment *toujours* moins de mémoire, à la fois à l'initialisation et aux mises à jour.

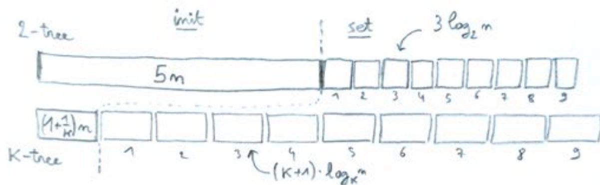
Analyse de l'empreinte mémoire totale pour $K > 8$

Conso. mémoire	Arbres binaires	Arbres K-aires
Arbre initial	$5 \cdot n$	$(1 + 1/K) \cdot n$
Mise à jour	$3 \cdot \log_2 n$	$(K + 1) \cdot \log_K n$



Analyse de l'empreinte mémoire totale pour $K > 8$

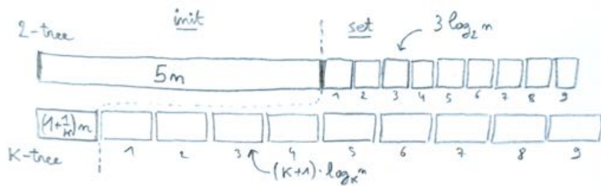
Conso. mémoire	Arbres binaires	Arbres K-aires
Arbre initial	$5 \cdot n$	$(1 + 1/K) \cdot n$
Mise à jour	$3 \cdot \log_2 n$	$(K+1) \cdot \log_K n$



Combien d'arbres vivants faut-il générer avec set avant d'être perdant ?

Analyse de l'empreinte mémoire totale pour $K > 8$

Conso. mémoire	Arbres binaires	Arbres K-aires
Arbre initial	$5 \cdot n$	$(1 + 1/K) \cdot n$
Mise à jour	$3 \cdot \log_2 n$	$(K + 1) \cdot \log_K n$



Combien d'arbres vivants faut-il générer avec `set` avant d'être perdant ?
Au moins :

$$\frac{4 - \frac{1}{K}}{\frac{K+1}{\log_2 K} - 3} \cdot \frac{n}{\log_2 n} \approx 4 \cdot \frac{\log_2 K}{K} \cdot \frac{n}{\log_2 n} \rightarrow 0{,}5 \cdot \frac{n}{\log_2 n} \text{ pour } K{=}64$$

Analyse du coût en temps d'une mise à jour (1/2)

Rappels :

- Arbre binaire : $\log_2 n$ allocations de 3 mots (sauf une de 2 mots)
- Arbre K-aire : $\log_K n$ allocations de $K+1$ mots
- Coût d'allocation : $As + B$ pour allouer s mots.

Étude du ratio, en notant $C = \frac{B}{A}$.

$$\frac{\text{coût de set sur arbres } K\text{-aires}}{\text{coût de set sur arbres binaires}} = \frac{(\log_K n)\cdot(K+1+C)}{(\log_2 n)\cdot(3+C)} = \frac{K+1+C}{(\log_2 K)\cdot(3+C)}$$

Analyse du coût en temps d'une mise à jour (2/2)

Valeurs de $\frac{\text{coût de set sur arbres } K\text{-aires}}{\text{coût de set sur arbres binaires}}$ en fonction de K et de C,

pour une plage de valeurs réalistes pour C.

C \ K	2	4	8	16	32	64	128	256
4	1.0	0.6	0.6	0.8	1.1	1.6	2.7	4.7
8	1.0	0.6	0.5	0.6	0.7	1.1	1.8	3.0
12	1.0	0.6	0.5	0.5	0.6	0.9	1.3	2.2
16	1.0	0.6	0.4	0.4	0.5	0.7	1.1	1.8

Analyse du coût en temps d'une mise à jour (2/2)

Valeurs de $\frac{\text{coût de set sur arbres } K\text{-aires}}{\text{coût de set sur arbres binaires}}$ en fonction de K et de C,

pour une plage de valeurs réalistes pour C.

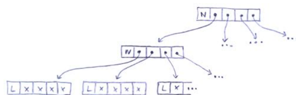
C \ K	2	4	8	16	32	64	128	256
4	1.0	0.6	0.6	0.8	1.1	1.6	2.7	4.7
8	1.0	0.6	0.5	0.6	0.7	1.1	1.8	3.0
12	1.0	0.6	0.5	0.5	0.6	0.9	1.3	2.2
16	1.0	0.6	0.4	0.4	0.5	0.7	1.1	1.8

- Pour $K \leqslant 16$, les arbres K-aires sont toujours gagnants
- $K = 32$ ou $K = 64$ sont très souvent des valeurs pertinentes
- $K \geqslant 128$: intéressant si beaucoup plus d'appels à `get` qu'à `set`

Représentation des arbres K-aires

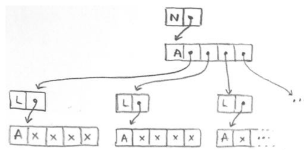
1. Constructeurs d'arité K

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



2. Constructeurs portant des tableaux

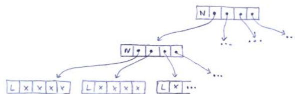
```
type 'a t =
  | Leaf of 'a array
  | Node of ('a t) array
```



Représentation des arbres K-aires

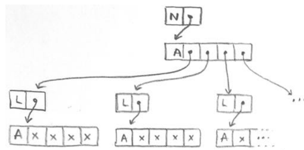
1. Constructeurs d'arité K

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



2. Constructeurs portant des tableaux

```
type 'a t =
  | Leaf of 'a array
  | Node of ('a t) array
```



3. Construction par tableaux imbriqués

- Pour hauteur 2, type `'a array array`
- Pour hauteur 3, type `'a array array array`
- Pour hauteur 4, type `'a array array array array`

Représentation par tableaux imbriqués

On ajoute un constructeur à la racine de l'arbre pour indiquer la hauteur, et pour stocker la valeur de $\log_2 K$.

```
type 'a tree =
  | Tree1 of int * 'a array
  | Tree2 of int * 'a array array
  | Tree3 of int * 'a array array array
  | Tree4 of int * 'a array array array array
  ...
```

En pratique $K \geqslant 8$ et $n \leqslant 2^{36}$, donc hauteur $h \leqslant 12$.

Représentation par tableaux imbriqués

On ajoute un constructeur à la racine de l'arbre pour indiquer la hauteur, et pour stocker la valeur de $\log_2 K$.

```
type 'a tree =
  | Tree1 of int * 'a array
  | Tree2 of int * 'a array array
  | Tree3 of int * 'a array array array
  | Tree4 of int * 'a array array array array
  ...
```

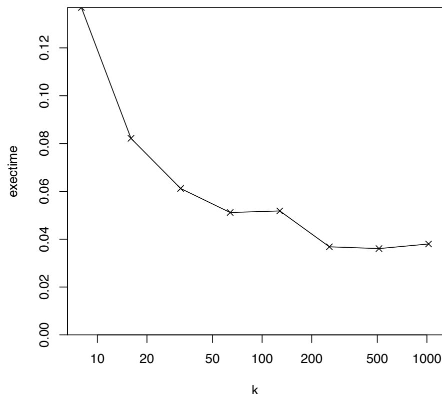
En pratique $K \geqslant 8$ et $n \leqslant 2^{36}$, donc hauteur $h \leqslant 12$.

Alternative : passer en code non typé.

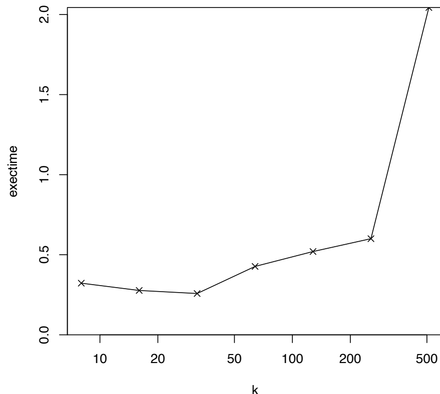
Impact du choix de K sur les arbres K-aires

Test : lectures et écritures aléatoires dans un tableau de longueur $n = 2^{24}$.

500k opérations `get`



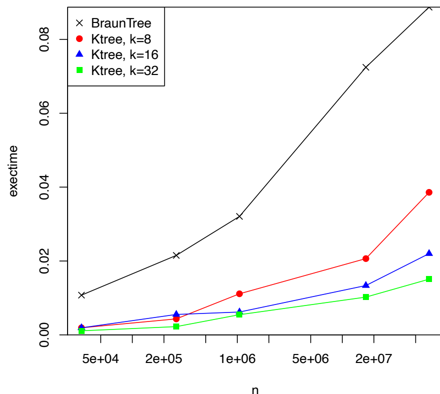
200k opérations `set`



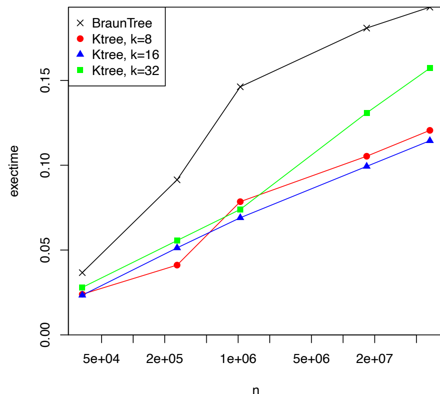
Performance des arbres K-aires

Test : lectures et écritures aléatoires dans un tableau de longueur n.

500k opérations `get`



200k opérations `set`



Implémentation des arbres de Braun de : `https://github.com/backtracking/flex-array`

Conclusions de la partie II

- Passage de $4n$ ou $5n$ mots à $(1 + \frac{2}{K})n$ mots
- Plus K est grand, plus `get` est rapide
- $K{=}8$ est toujours gagnant en temps et en mémoire
- $K{=}16$ est toujours gagnant en temps
- $K{=}32$ ou $K{=}64$ est souvent pertinent

Partie III

La persistance avec des effets internes

Optimiser les mises à jour persistantes

Situation :

- plus K est grand, plus `get` sur l'arbre est rapide
- une mise à jour fonctionnelle sur un tableau de taille K coûte $O(K)$
- `set` coûte $O(K \log_K n)$ en temps et espace

Optimiser les mises à jour persistantes

Situation :

- plus K est grand, plus `get` sur l'arbre est rapide
- une mise à jour fonctionnelle sur un tableau de taille K coûte $O(K)$
- `set` coûte $O(K \log_K n)$ en temps et espace

  $\rightarrow$ c'est le facteur limitant à $K{=}16$ ou $K{=}32$

Optimiser les mises à jour persistantes

Situation :

- plus K est grand, plus `get` sur l'arbre est rapide
- une mise à jour fonctionnelle sur un tableau de taille K coûte $O(K)$
- `set` coûte $O(K \log_K n)$ en temps et espace
  $\rightarrow$ c'est le facteur limitant à $K{=}16$ ou $K{=}32$

Objectifs :

- mise-à-jour fonctionnelle sur tableau de taille K en $O(1)$, plus un coût amorti de $O(K)$ pour les accès futurs aux anciennes versions
- `set` sur un arbre K-aire en $O(\log_K n)$ en temps et espace

Tableaux persistants

Représentons les nœuds des arbres persistants d'arité K
avec des tableaux persistants de taille K.

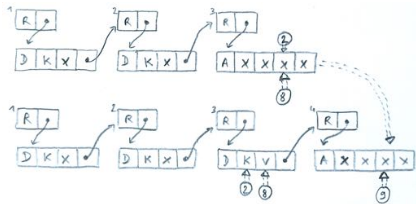
Tableaux persistants :

- avec des *gros éléments* ou *gros nœuds*
  Voir 4ème cours, Driscoll et al. '89, et O'Neill et Burton '97
- avec des chaînes de modifications Baker'91
  → plus simple, et permet au GC de libérer la mémoire

Note : d'autres techniques, non discutées ici, sont applicables pour les opérations `push` et `pop`.

Tableaux persistants de Baker

```
type 'a parray = ('a parray_desc) ref
and 'a parray_desc =
  | Array of { data : 'a array }
  | Diff of { index : int;
              value : 'a;
              rest : 'a parray }
```



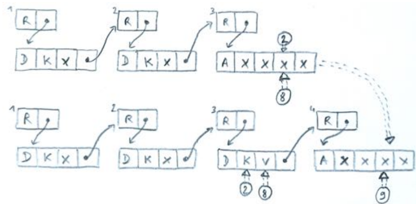
Intérêt :

- $O(1)$ pour une modification sur la dernière version
- $O(1)$ amorti pour usage dans un algorithme qui *backtrack*

cf. séminaire de J.C. Filliâtre

Tableaux persistants de Baker

```
type 'a parray = ('a parray_desc) ref
and 'a parray_desc =
  | Array of { data : 'a array }
  | Diff of { index : int;
              value : 'a;
              rest : 'a parray }
```



Intérêt :

- $O(1)$ pour une modification sur la dernière version
- $O(1)$ amorti pour usage dans un algorithme qui *backtrack*

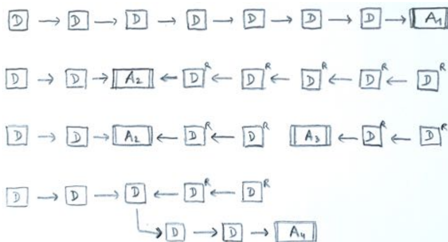
cf. séminaire de J.C. Filliâtre

Limitations :

- coût non borné pour traverser une chaîne de modifications
- allers-retours coûteux si usage de la persistance totale

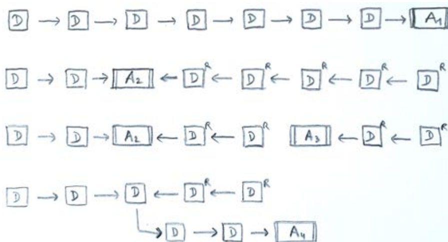
Adapter les tableaux de Baker

1. Empêcher la construction de chaîne de taille supérieur à $O(K)$, et payer une copie de tableau en $O(K)$ lorsqu'on atteint la limite.
2. Renverser au plus une fois les chaînes de modifications ; sinon, reconstruire un tableau indépendant en $O(K)$.



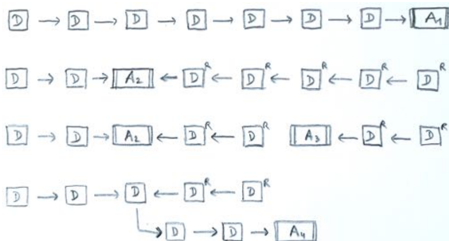
Adapter les tableaux de Baker

1. Empêcher la construction de chaîne de taille supérieur à $O(K)$, et payer une copie de tableau en $O(K)$ lorsqu'on atteint la limite.
2. Renverser au plus une fois les chaînes de modifications ; sinon, reconstruire un tableau indépendant en $O(K)$.



Chaque nœud retient la longueur maximale d'une chaîne qui l'atteint.

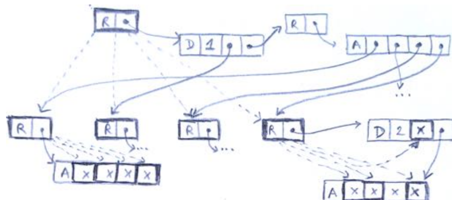
Analyse des tableaux persistants



- une opération `set` est en coût immédiat $O(1)$
- une opération `get` ou `set` sur un antécédent est en $O(1)$ amorti, et on peut remonter les antécédents en cascade (semi-persistance)
- si on utilise la persistence totale, alors le `set` d'origine est $O(K)$

Arbres K-aires avec tableaux persistants

Pas facile à visualiser...



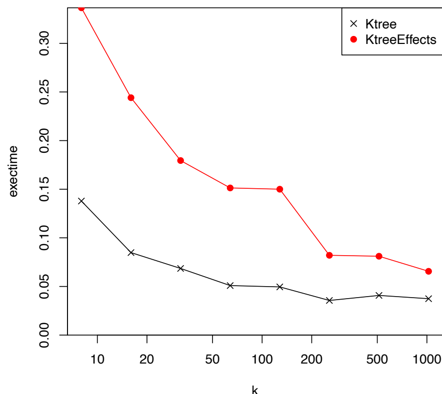
`set` effectue $\log_K n$ mises à jour dans des tableaux persistants :

- $O(\log_K n)$ si accès semi-persistant
- $O(K \log_K n)$ si accès en persistance totale

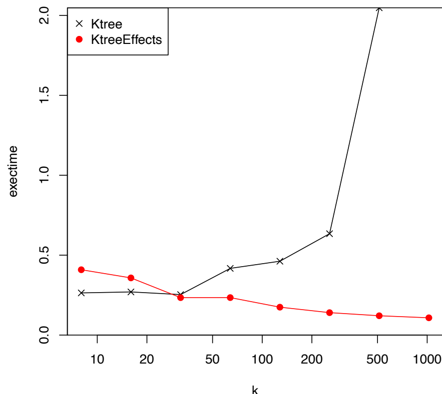
Impact du choix de K, avec tableaux persistants

Test dans le **cas favorable** où l'on repart toujours de la dernière version.

500k opérations `get`



200k opérations `set`

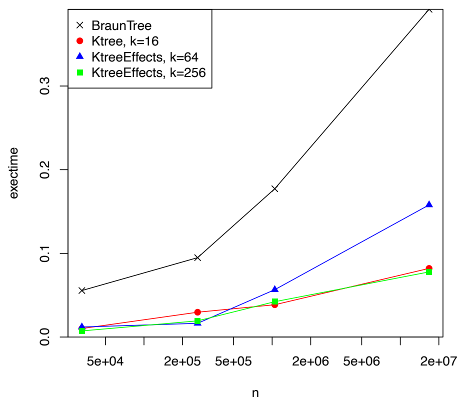


$\rightarrow$ `get` plus cher : il faut traverser des références en plus

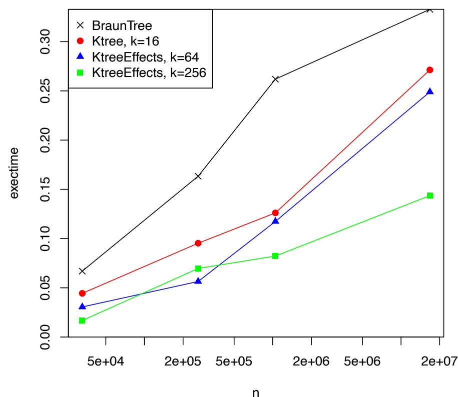
$\rightarrow$ `set` moins cher : on peut choisir K bien plus grand

Performance des arbres K-aires avec tableaux persistants

500k opérations `get`



200k opérations `set`

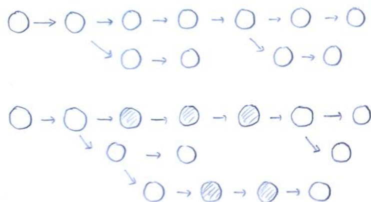


Partie IV

Transience : la persistance par intermittence

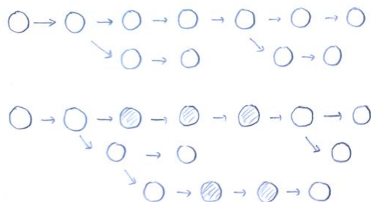
Optimisation des écritures dans les arbres persistants

Le programmeur n'a pas forcément besoin que tous les états intermédiaires soient persistants.



Optimisation des écritures dans les arbres persistants

Le programmeur n'a pas forcément besoin que tous les états intermédiaires soient persistants.



Une fois un tableau frais alloué, on peut le modifier en place tant qu'on est dans des états intermédiaires non persistants.

Interface des tableaux transients

Opérations éphémères $\rightarrow$ avec certaines écritures en place

```
type 'a t
val emake : int -> 'a -> 'a t
val eget : 'a t -> int -> 'a
val eset : 'a t -> int -> 'a -> unit
```

Opérations persistantes

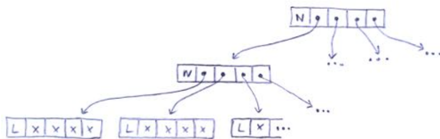
```
type 'a u
val pmake : int -> 'a -> 'a u
val pget : 'a u -> int -> 'a
val pset : 'a u -> int -> 'a -> 'a u
```

Opérations de conversions en $O(1)$

```
val persistent_to_ephemeral : 'a u -> 'a t
val ephemeral_to_persistent : 'a t -> 'a u
```

Tracer la possession unique

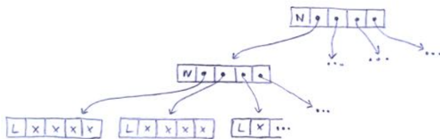
Dans un arbre éphémère, on veut distinguer les nœuds qui ont déjà été copiés depuis le point de conversion en arbre éphémère.



Idée naïve : marquer les tableaux déjà copiés avec un booléen.
Problème : conversion en $O(\frac{n}{K})$ pour effacer les marques.

Tracer la possession unique

Dans un arbre éphémère, on veut distinguer les nœuds qui ont déjà été copiés depuis le point de conversion en arbre éphémère.



Idée naïve : marquer les tableaux déjà copiés avec un booléen.
Problème : conversion en $O(\frac{n}{K})$ pour effacer les marques.

Solution : associer un identifiant unique à l'arbre éphémère, et chaque nœud de l'arbre peut porter un identifiant. Invalider l'identifiant démarque tous les tableaux en $O(1)$.

Objectifs de Sek

On va construire une structure de séquence basée sur un type d'arbre particulier utilisant des tableaux de taille K.

On suppose ici K être une petite constante, de sorte que $O(K) = O(1)$.

Séquence transiente avec :

- `push` et `pop`, aux deux extrémités, en $O(1)$
- `concat` et `split` en $O(\log_K n)$
- `get` et `set` en $O(\log_K n)$
- itérateurs d'ordre supérieur : `iter` et `fold`
- itérateur du premier ordre : `next`, `prev`, `reach`

Application en vue : chaînes de caractères.

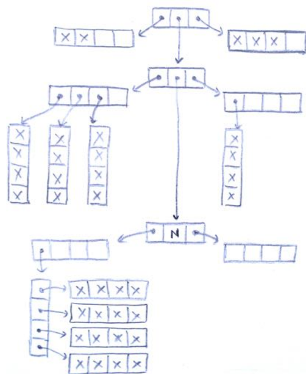
Arbres supportant la concaténation et le découpage

Kaplan and Tarjan (1996)
Purely functional representations of catenable sorted lists
→ purement fonctionnel

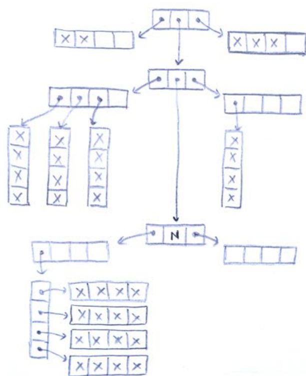
Hinze and Paterson (2006)
Finger trees : a simple general-purpose data structure
→ plus simple, avec paresse comme dans *implicit deques* de Okasaki

Acar, Charguéraud, Rainey (2014)
Theory and practice of chunked sequences
→ encore plus simple, avec accès aux bords généralement en $O(1)$ mais en $O(\log_K n)$ dans de rares cas pathologiques.

Représentation des chunked sequences persistantes

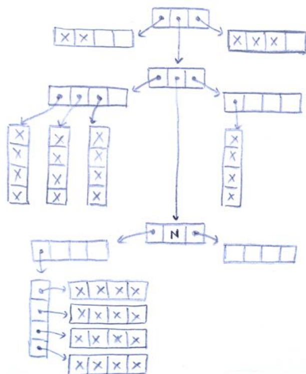


Représentation des chunked sequences persistantes



Chunk :
tableau persistant de capacité K,
rempli sur un segment, et
pouvant porter un identifiant

Représentation des chunked sequences persistantes



Chunk :
tableau persistant de capacité K, rempli sur un segment, et pouvant porter un identifiant

```
type 'a chunk = {
   support : 'a parray;
   head : int;
   size : int;
   id : unit ref; }

type 'a tree = {
  front : 'a chunk;
  middle : (('a chunk) tree) option;
  back : 'a chunk; }
```

Il faut également stocker des poids pour supporter `get` et `split`.

Représentation de Sek

Variant persistant

- tableau pour les séquences courtes, arbre persistant sinon
- tous les chunks sont persistants

Variant éphémère

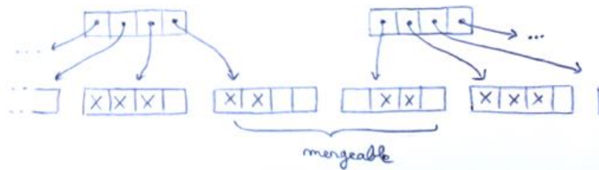
- identifiant unique
- chunks mutables en surface, toujours uniquement possédés
- arbre aux nœuds éphémères ou persistants selon leur identifiant

```
type 'a u =
  | Empty of { default : 'a; }
  | Short of {
      default : 'a;
      elems : 'a array }
  | Tree of {
      front : 'a chunk;
      middle : ('a chunk) u;
      back : 'a chunk; }
```

```
type 'a t = {
  id : unit ref;
  mutable front : 'a chunk;
  mutable middle : ('a chunk) u;
  mutable back : 'a chunk; }
```

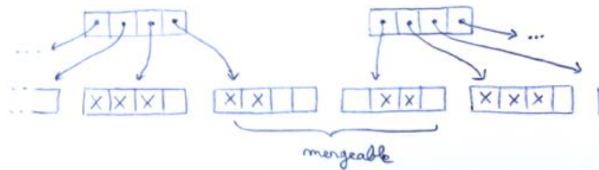
Compaction et analyse de l'empreinte mémoire

Difficulté : introduction de chunks incomplets lors des concaténations.



Compaction et analyse de l'empreinte mémoire

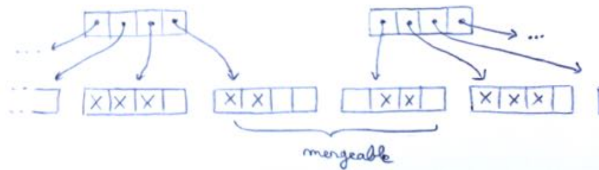
Difficulté : introduction de chunks incomplets lors des concaténations.



Invariant : 2 chunks consécutifs doivent contenir plus de K éléments.

Compaction et analyse de l'empreinte mémoire

Difficulté : introduction de chunks incomplets lors des concaténations.

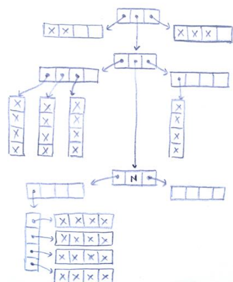


Invariant : 2 chunks consécutifs doivent contenir plus de K éléments.

Consommation mémoire, en mots :

- Si pas de concaténation : $(1 + \epsilon) \cdot n$
- Sinon, en pire cas : $2 \cdot (1 + \epsilon) \cdot n$

Travaux futurs



Le paquet `sek` est déjà disponible sur `opam`.

Reste à faire :

- Intégrer les "chaînes de modifications"
- Générer du code via des transformations
- Compléter les mesures de performance
- Finir la vérification en Coq avec l'outil CFML

Merci !