Structures de données persistantes, quatrième cours

Comment rendre persistante une structure impérative?

Xavier Leroy

2023-03-30

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

Objectifs du cours

Les structures de données persistantes vues jusqu'ici ont des implémentations purement fonctionnelles + éventuellement des suspensions pour l'évaluation paresseuse.

Aujourd'hui nous allons étudier des structures persistantes dont l'implémentation est impérative. Plus encore, la structure persistante est dérivée de manière semi-systématique d'une structure éphémère :

- Tableaux $\rightarrow$ tableaux persistants
  - Par ajout d'un historique (approche de Baker)
  - En utilisant des ≪gros éléments≫ (O'Neill et Burton)
- Arbres mutables $\rightarrow$ arbres persistants
  - En utilisant des ≪gros nœuds≫ (*fat nodes*) (Driscoll, Sarnak, Sleator et Tarjan)

Persistante complète, persistance partielle

Persistance partielle :

- seule la version la plus récente de la structure peut être modifiée ; → une suite de versions
- les versions antérieures sont en lecture seule.

(Note : c'est moins restrictif que l'utilisation linéaire (*single threaded*) mentionnée au cours précédent.)

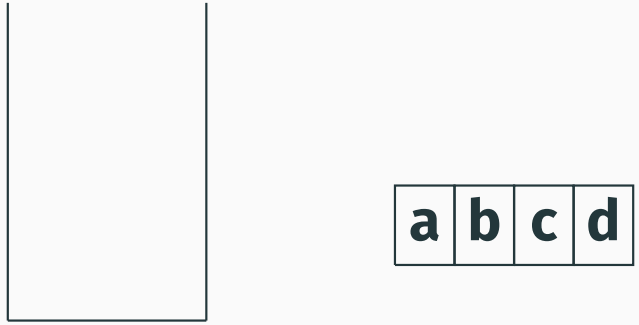
Persistance complète :

- toutes les versions peuvent être modifiées pour créer de nouvelles versions. → un arbre de versions

La persistance complète s'obtient ≪gratuitement≫ avec des implémentations fonctionnelles pures. Mais pour beaucoup d'algorithmes (p.ex. en géométrie), la persistance partielle suffit.

Tableaux persistants, 1 : ajout d'un historique

Retour en arrière (*backtracking*) sur un tableau mutable



Un tableau mutable A plus une pile S contenant des paires (indice i, précédente valeur de $A[i]$).

Pour écrire v dans $A[i]$:

empiler $(i, A[i])$ sur S

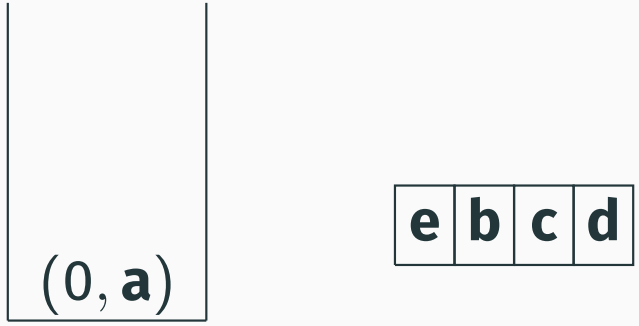
modifier $A[i] := v$.

Pour revenir en arrière :

tant que S n'est pas vide :

dépiler (i, v) et modifier $A[i] := v$.

Retour en arrière (*backtracking*) sur un tableau mutable



Un tableau mutable A plus une pile S contenant des paires (indice i, précédente valeur de $A[i]$).

Pour écrire v dans $A[i]$:

empiler $(i, A[i])$ sur S

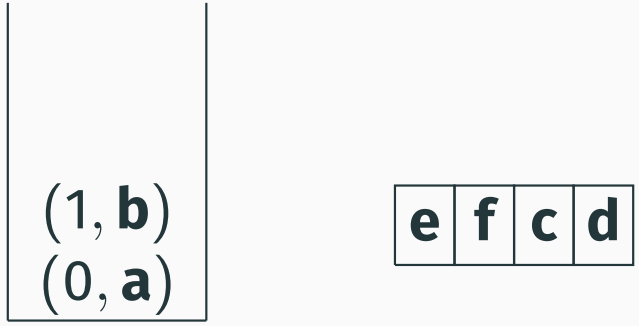
modifier $A[i] := v$.

Pour revenir en arrière :

tant que S n'est pas vide :

dépiler (i, v) et modifier $A[i] := v$.

Retour en arrière (*backtracking*) sur un tableau mutable



Un tableau mutable A plus une pile S contenant des paires (indice i, précédente valeur de $A[i]$).

Pour écrire v dans $A[i]$:

empiler $(i, A[i])$ sur S

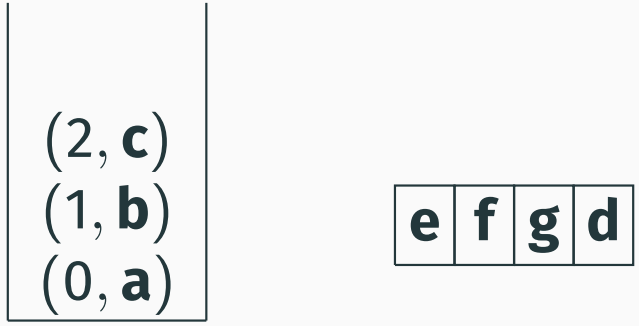
modifier $A[i] := v$.

Pour revenir en arrière :

tant que S n'est pas vide :

dépiler (i, v) et modifier $A[i] := v$.

Retour en arrière (*backtracking*) sur un tableau mutable



Un tableau mutable A plus une pile S contenant des paires (indice i, précédente valeur de $A[i]$).

Pour écrire v dans $A[i]$:

empiler $(i, A[i])$ sur S

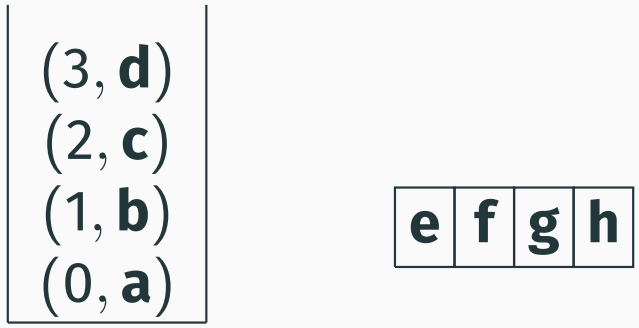
modifier $A[i] := v$.

Pour revenir en arrière :

tant que S n'est pas vide :

dépiler (i, v) et modifier $A[i] := v$.

Retour en arrière (*backtracking*) sur un tableau mutable



Un tableau mutable A plus une pile S contenant des paires (indice i, précédente valeur de $A[i]$).

Pour écrire v dans $A[i]$:

empiler $(i, A[i])$ sur S

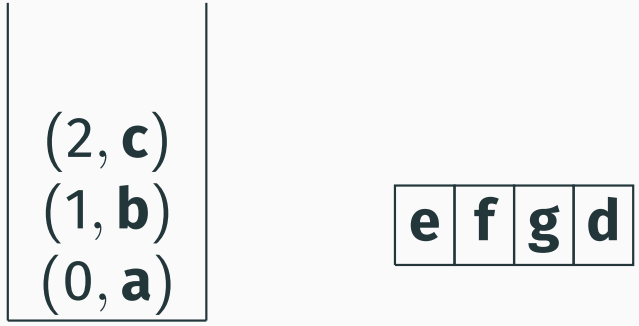
modifier $A[i] := v$.

Pour revenir en arrière :

tant que S n'est pas vide :

dépiler (i, v) et modifier $A[i] := v$.

Retour en arrière (*backtracking*) sur un tableau mutable



Un tableau mutable A plus une pile S contenant des paires (indice i, précédente valeur de $A[i]$).

Pour écrire v dans $A[i]$:

empiler $(i, A[i])$ sur S

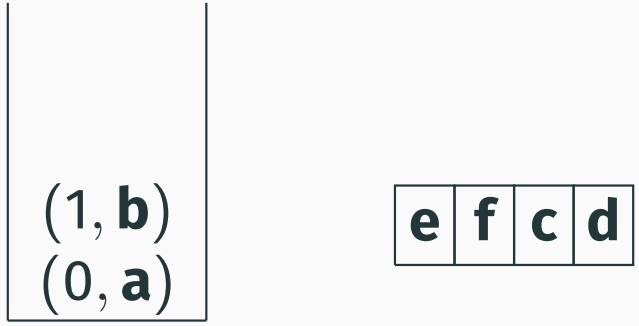
modifier $A[i] := v$.

Pour revenir en arrière :

tant que S n'est pas vide :

dépiler (i, v) et modifier $A[i] := v$.

Retour en arrière (*backtracking*) sur un tableau mutable



Un tableau mutable A plus une pile S contenant des paires (indice i, précédente valeur de $A[i]$).

Pour écrire v dans $A[i]$:

empiler $(i, A[i])$ sur S

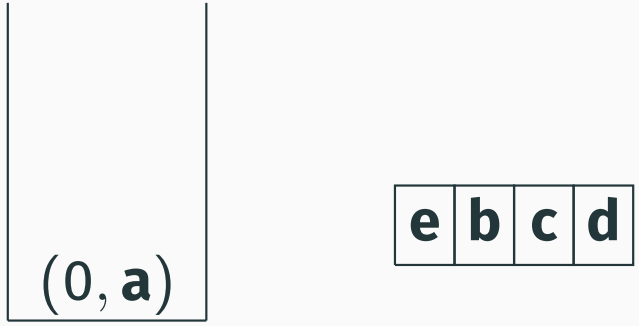
modifier $A[i] := v$.

Pour revenir en arrière :

tant que S n'est pas vide :

dépiler (i, v) et modifier $A[i] := v$.

Retour en arrière (*backtracking*) sur un tableau mutable



Un tableau mutable A plus une pile S contenant des paires (indice i, précédente valeur de $A[i]$).

Pour écrire v dans $A[i]$:

empiler $(i, A[i])$ sur S

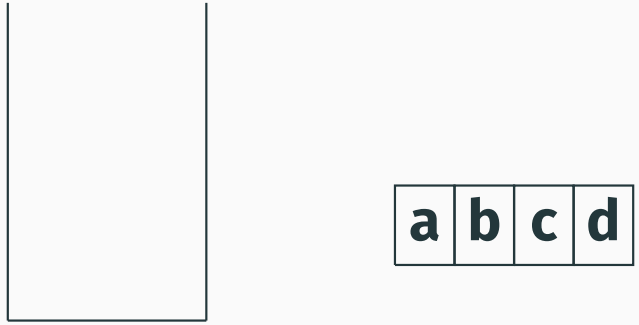
modifier $A[i] := v$.

Pour revenir en arrière :

tant que S n'est pas vide :

dépiler (i, v) et modifier $A[i] := v$.

Retour en arrière (*backtracking*) sur un tableau mutable



Un tableau mutable A plus une pile S contenant des paires (indice i, précédente valeur de $A[i]$).

Pour écrire v dans $A[i]$:

empiler $(i, A[i])$ sur S

modifier $A[i] := v$.

Pour revenir en arrière :

tant que S n'est pas vide :

dépiler (i, v) et modifier $A[i] := v$.

Retrouver la valeur initiale d'une case du tableau

Pas besoin de faire le retour en arrière complet si tout ce qui nous intéresse est de retrouver la valeur initiale de $A[i]$:

pour $k = 0, \ldots, |S| - 1$:
    si la k^{e} entrée de S est de la forme (i, v) :
        renvoyer v
renvoyer $A[i]$.

Retrouver la valeur d'une case du tableau à la date t

Pour retrouver la valeur qu'avait $A[i]$ à la date t
(c.à.d. après t écritures dans le tableau),
il suffit de commencer la recherche à $k = t$ au lieu de $k = 0$:

pour $k = t, \ldots, |S| - 1$:
    si la k^{e} entrée de S est de la forme (i, v) :
        renvoyer v
renvoyer $A[i]$.

Nous avons donc rendu le tableau A partiellement persistant :

- on peut consulter son état à toute date t
- et modifier son état à la dernière date t,
  obtenant la date $t + 1$.

Une représentation chaînée de l'historique

Utilisant des références (= cellules d'indirection mutables).

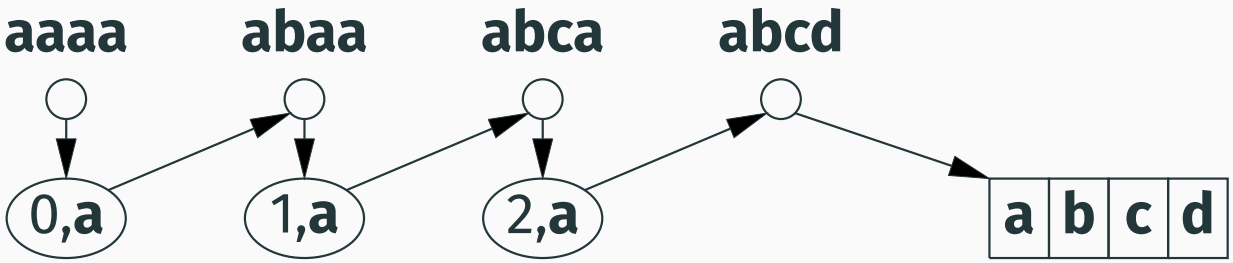
```
type 'a parray = 'a cell ref
and 'a cell =
  | Base of 'a array
  | Diff of int * 'a * 'a parray

let make size init = ref (Base (Array.make size init))

let rec get p i =
  match !p with
  | Base a -> a.(i)
  | Diff(j, v, q) -> if i = j then v else get q i
```

Note : `get` est en temps linéaire en la longueur de la chaîne de `Diff`.

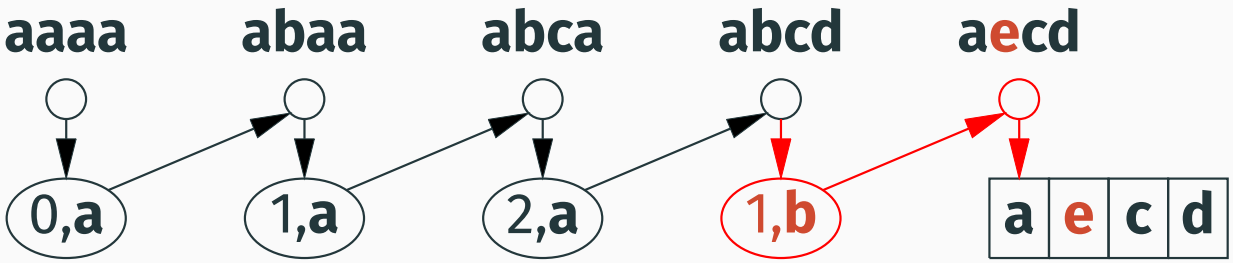
Modifier la base du tableau persistant



On modifie le tableau en place et on compense en insérant un nœud `Diff` dans l'historique :

```
let set p i v =
  match !p with
  | Base a ->
      let q = ref (Base a) in
      p := Diff(i, a.(i), q); a.(i) <- v; q
  | _ ->
      raise (Failure "full persistence not supported")
```

Modifier la base du tableau persistant



On modifie le tableau en place et on compense en insérant un nœud `Diff` dans l'historique :

```
let set p i v =
  match !p with
  | Base a ->
      let q = ref (Base a) in
      p := Diff(i, a.(i), q); a.(i) <- v; q
  | _ ->
      raise (Failure "full persistence not supported")
```

Un tableau partiellement persistant

On obtient donc un tableau partiellement persistant, avec

- `set` en temps $\mathcal{O}(1)$, applicable seulement à la dernière version du tableau ;
- `get` sur la dernière version du tableau en temps $\mathcal{O}(1)$;
- `get` sur une version quelconque en temps $\mathcal{O}(w)$, où w est le nombre total de versions (= d'écritures) ;
- Espace total $\mathcal{O}(n + w)$, où n est la taille du tableau.

En utilisant une technique de reconstruction globale (*global rebuilding*), on peut se ramener à `get` en temps garanti $\mathcal{O}(n)$ et `set` en temps amorti $\mathcal{O}(1)$.

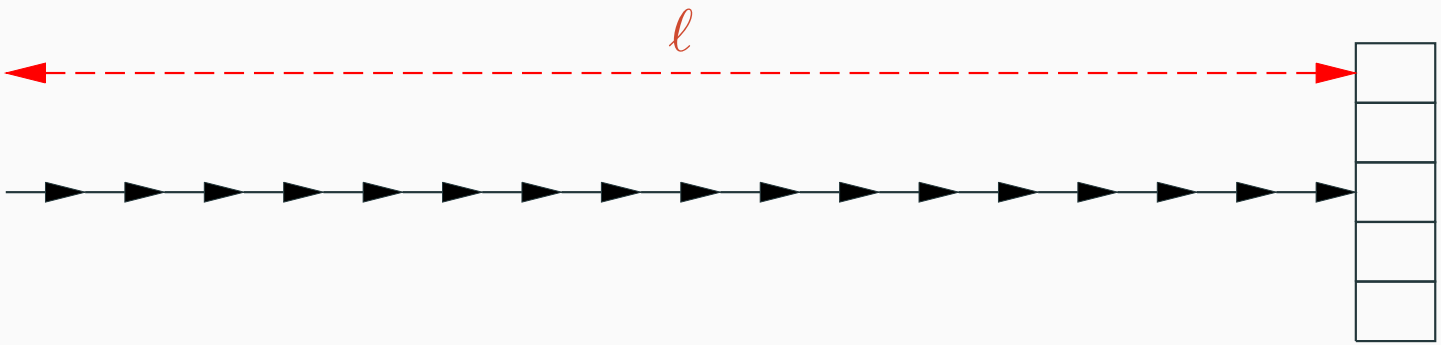
Reconstruction d'un tableau

```
let rec to_array p =
  match !p with
  | Base a -> Array.copy a
  | Diff(i, v, q) -> let a = to_array q in a.(i) <- v; a

let rebuild p =
  p := Base (to_array p)
```

On a un nouveau tableau de base où tous les `Diff` ont été appliqués. Les opérations `get` sur `p` redeviennent en temps $\mathcal{O}(1)$, mais la reconstruction a coûté $\mathcal{O}(n + \ell)$ en temps et $\mathcal{O}(n)$ en espace. (ℓ = longueur de la chaîne de `Diff`)

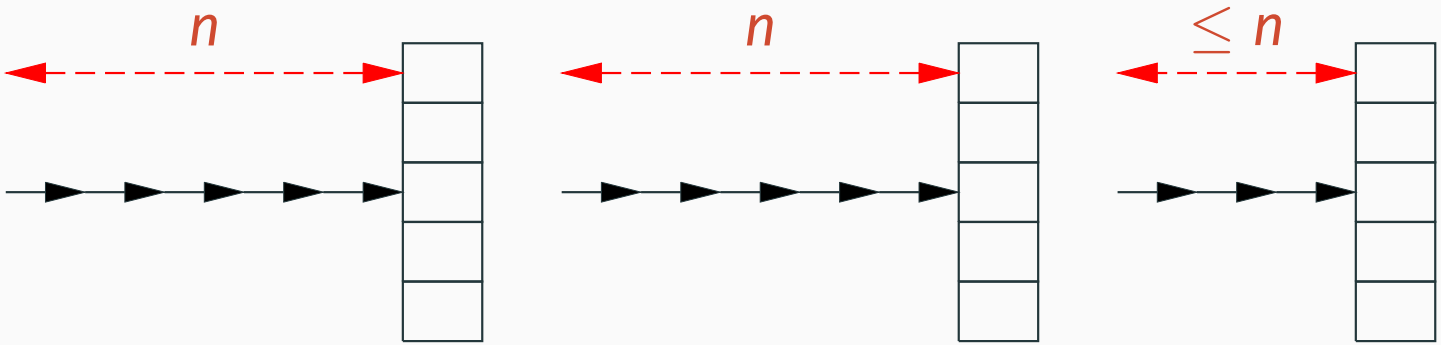
Reconstruction lors d'un `get`



Si une opération `get` parcours $\ell > n$ nœuds `Diff`, elle peut reconstruire $k = \lceil \ell/n \rceil - 1$ tableaux intermédiaires, obtenant $k + 1$ chaînes de `Diff` de longueur $\leq n$.

Cette reconstruction prend un temps $\mathcal{O}(kn) = \mathcal{O}(\ell)$, qui peut être amorti sur les ℓ opérations `set` nécessaires à créér une chaîne aussi longue.

Reconstruction lors d'un `get`



Si une opération `get` parcours $\ell > n$ nœuds `Diff`, elle peut reconstruire $k = \lceil \ell / n \rceil - 1$ tableaux intermédiaires, obtenant $k + 1$ chaînes de `Diff` de longueur $\leq n$.

Cette reconstruction prend un temps $\mathcal{O}(kn) = \mathcal{O}(\ell)$, qui peut être amorti sur les ℓ opérations `set` nécessaires à créér une chaîne aussi longue.

Vers la persistance complète

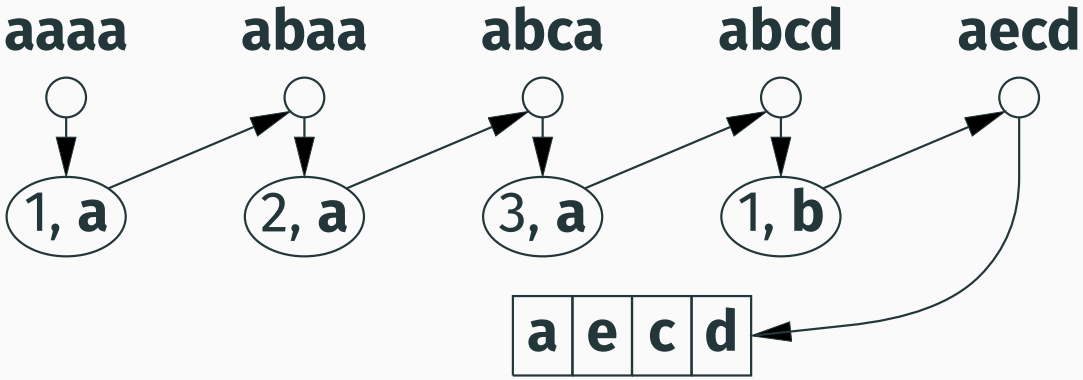
Un nœud `Diff` peut stocker l'ancienne valeur d'une case, mais aussi sa nouvelle valeur !

```
let set p i v =
  match !p with
  | Base a ->
      let q = ref (Base a) in
      p := Diff(i, a.(i), q); a.(i) <- v; q
  | _ ->
      ref (Diff(i, v, p))
```

Le tableau est maintenant complètement persistant.

On perd la garantie que `get` est toujours en $\mathcal{O}(1)$ sur le résultat du dernier `set` effectué.

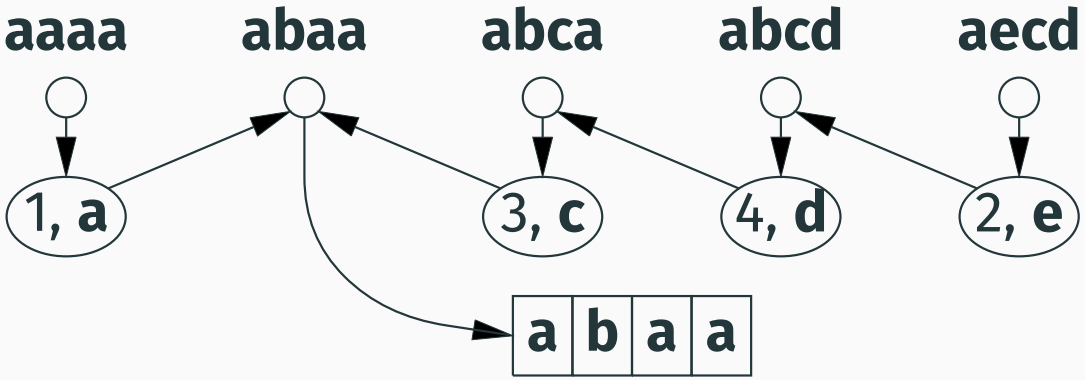
Changement de racine (H. Baker, 1991)



Par renversement de nœuds `Diff`, un tableau persistant peut toujours être ramené à l'état `Base`, sans allocation de nouveau tableau.

```
let rec reroot p =
  match !p with
  | Base a -> a
  | Diff(i, v, q) ->
      let a = reroot q in
      p := Base a; q := Diff(i, a.(i), p); a.(i) <- v; a
```

Changement de racine (H. Baker, 1991)



Par renversement de nœuds `Diff`, un tableau persistant peut toujours être ramené à l'état `Base`, sans allocation de nouveau tableau.

```
let rec reroot p =
  match !p with
  | Base a -> a
  | Diff(i, v, q) ->
      let a = reroot q in
      p := Base a; q := Diff(i, a.(i), p); a.(i) <- v; a
```

Persistance complète par changement de racine

```
let set p i v =
  let a = reroot p in
  let q = ref (Base a) in
  p := Diff(i, a.(i), q);
  a.(i) <- v;
  q
```

Les opérations `get` et `set` sont maintenant en temps $\mathcal{O}(1)$ sur le résultat du dernier `set`, et en temps $\mathcal{O}(w)$ sur les autres versions du tableau. (w = nombre total de versions)

Généralisation à d'autres structures de données

L'approche ≪structure persistante = structure éphémère + historique de modifications≫ n'est en rien spécifique aux tableaux.

Elle s'applique également à d'autres structures éphémères, pourvu que leurs opérations soient ≪suffisamment inversibles≫. Par exemple :

- Tables de hachage (`add–remove`)
- Files d'attente doubles (*dequeue*) (`cons–tail`, `add–take`)

D'autres structures ne sont pas assez inversibles pour cet usage :

- Files d'attente (`tail–?`, `add–?`)
- Files de priorité (tas) (`delMin–add`, `add–?`)

Tableaux persistants, 2 : utilisation de «gros éléments»

Tableaux de gros éléments (*fat elements*)

Proposés par M. E. O'Neill et F. W. Burton, *A new method for functional arrays*, JFP, 1997.

Une variante de la technique des ≪gros nœuds≫ de Driscoll, Sarnak, Sleator et Tarjan (voir plus loin).

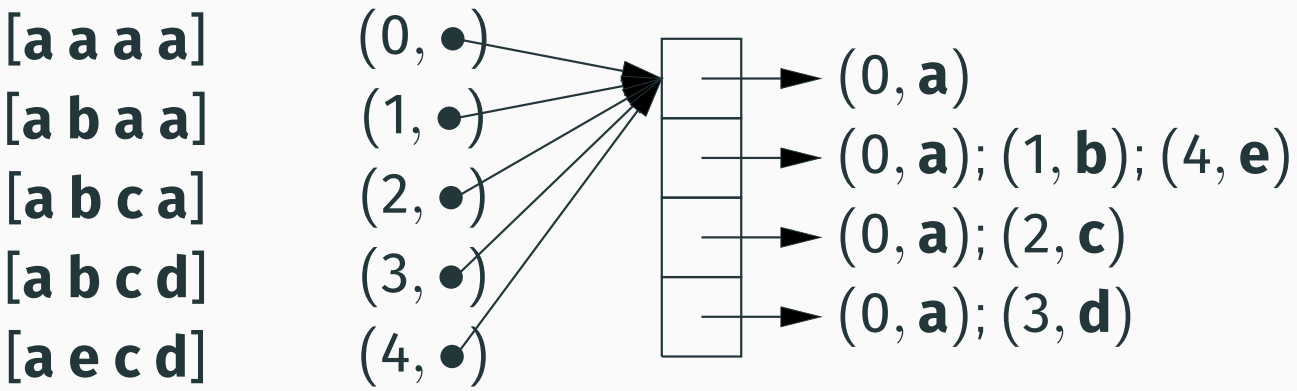
Idée : au lieu de représenter un tableau persistant par

un tableau mutable de valeurs
+ un historique des modifications du tableau,

on va le représenter par

un tableau mutable de gros éléments (*fat elements*),
avec un gros élément =
un historique des modifications de l'élément.

Tableaux de gros éléments



Un gros élément = un ensemble ordonné de paires
(date de modification, nouvelle valeur à cette date.)

Un tableau persistant = une paire

date t à laquelle on observe,

tableau mutable A de gros éléments (partagé).

Valeur v de l'élément i à la date t =

élément (t', v) de $A[i]$ tel que $t' \leq t$ et t' maximal.

Une implémentation en OCaml

```
module QSet = Set.Make(Q)
module QMap = Map.Make(Q)

type timestamp = Q.t
type 'a fat_element = 'a QMap.t
type 'a collection =
  { arr: 'a fat_element array;
    mutable timestamps: QSet.t }
type 'a parray = timestamp * 'a collection

let make size init =
  let t0 = Q.zero in
  let c = { arr = Array.make size (QMap.singleton t0 init);
            timestamps = QSet.singleton t0 } in
  (t0, c)
```

Lecture d'un élément

```
let get_elt e t =
    match QMap.find_last_opt (fun t' -> Q.leq t' t) e with
    | None -> assert false
    | Some(_, v) -> v

let get (t, c) i =
    get_elt c.arr.(i) t
```

`QMap` étant implémenté par des arbres binaires de recherche équilibrés, `get p i` est en temps $\mathcal{O}(\log w_i)$,
où w_i est le nombre d'écritures à la position i.

Comparer avec les tableaux persistants de Baker, où `get` est en temps $\mathcal{O}(w)$, où w est le nombre total d'écritures.

Écriture d'un élément dans la dernière version

```
let set (t, c) i v =
  match QSet.find_first_opt (fun t' -> Q.gt t' t) c.timestamps
  with
  | None ->
      (* Updating the latest version of the array *)
      let t' = Q.(add t one) in
      c.timestamps <- QSet.add t' c.timestamps;
      let e = c.arr.(i) in
      let e' = QMap.add t' v e in
      c.arr.(i) <- e';
      (t', c)
  | Some ->
      raise (Failure "full persistence not supported yet")
```

On crée une nouvelle version datée $t + 1$, avec une entrée de plus dans le i^{e} gros élément. Temps : $\mathcal{O}(\log w_i)$.

Analyse dans le cas de la persistance partielle

Au bout de n écritures, on peut reconstruire un nouveau tableau en temps $\mathcal{O}(n \log n)$ (n requêtes `get` dans l'ancien tableau).

Ce temps et espace sont amortis sur les n écritures qui précèdent.

Dès lors, chaque gros élément est de taille $\leq n$, d'où :

- lecture en temps garanti $\mathcal{O}(\log n)$
- écriture en temps amorti $\mathcal{O}(\log n)$
- taille du tableau $\mathcal{O}(n + w)$ si les gros éléments sont représentés par des arbres équilibrés modifiés en place.

En utilisant des arbres splay (*splay trees*), on a des accès en temps $\mathcal{O}(1)$ pour les éléments qui ont été précédemment accédés à une date proche.

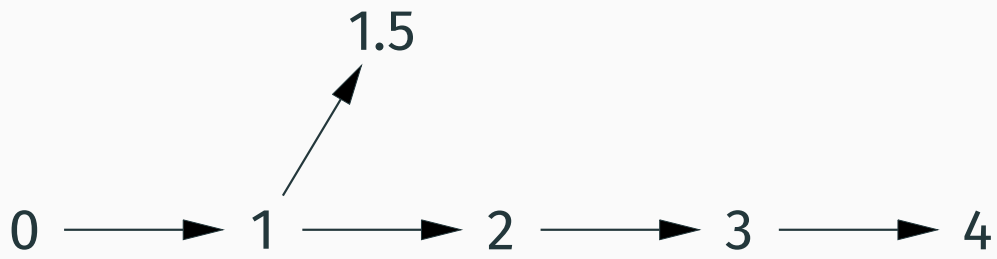
Écriture dans une version quelconque du tableau

Lors d'une écriture sur la version datée t, si des versions datées $> t$ existent déjà, on choisit une date intermédiaire entre t et ces versions ultérieures.

C'est un moyen simple pour injecter l'ordre partiel des versions dans l'ordre total des dates.

Écriture dans une version quelconque du tableau

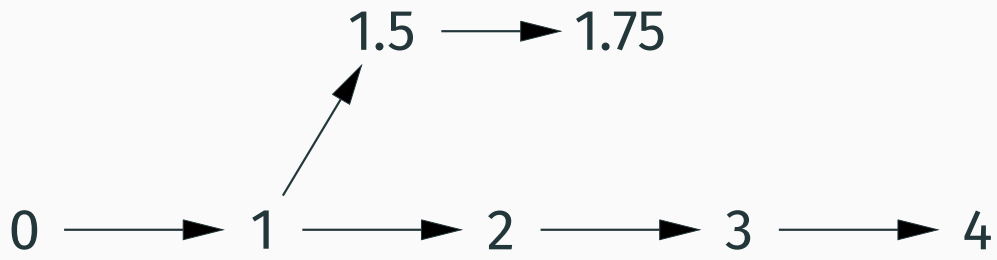
Lors d'une écriture sur la version datée t, si des versions datées $> t$ existent déjà, on choisit une date intermédiaire entre t et ces versions ultérieures.



C'est un moyen simple pour injecter l'ordre partiel des versions dans l'ordre total des dates.

Écriture dans une version quelconque du tableau

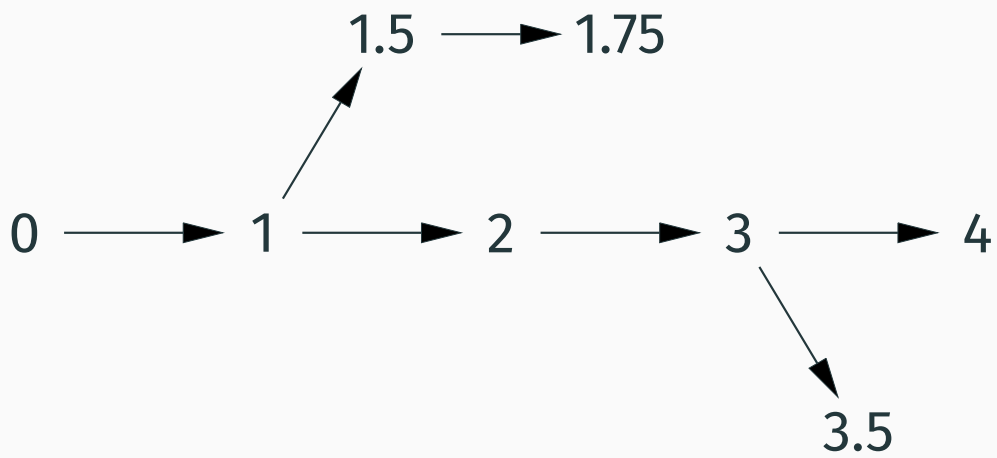
Lors d'une écriture sur la version datée t, si des versions datées $> t$ existent déjà, on choisit une date intermédiaire entre t et ces versions ultérieures.



C'est un moyen simple pour injecter l'ordre partiel des versions dans l'ordre total des dates.

Écriture dans une version quelconque du tableau

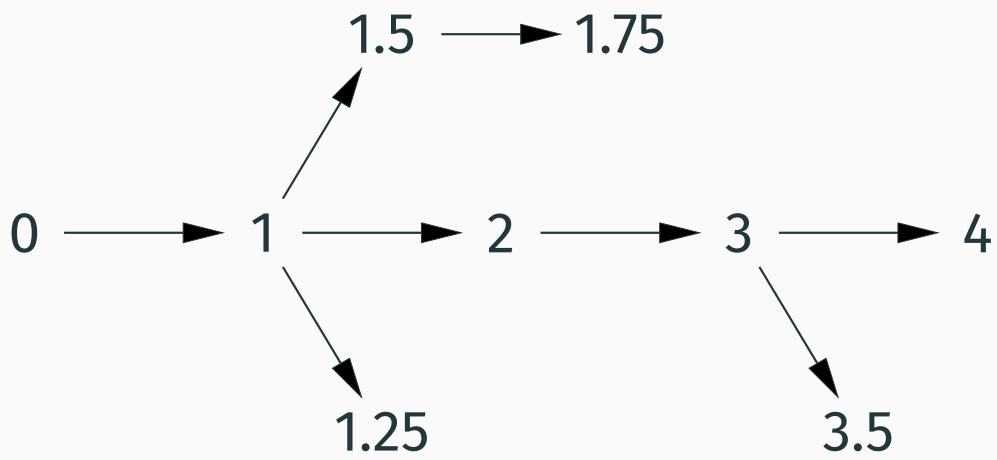
Lors d'une écriture sur la version datée t, si des versions datées $> t$ existent déjà, on choisit une date intermédiaire entre t et ces versions ultérieures.



C'est un moyen simple pour injecter l'ordre partiel des versions dans l'ordre total des dates.

Écriture dans une version quelconque du tableau

Lors d'une écriture sur la version datée t, si des versions datées $> t$ existent déjà, on choisit une date intermédiaire entre t et ces versions ultérieures.



C'est un moyen simple pour injecter l'ordre partiel des versions dans l'ordre total des dates.

Les dates intermédiaires

Étant données deux dates t, t'' il faut savoir créer une date intermédiaire t', p.ex. $t' = (t + t'')/2$.

- Solution simpliste : une date = un nombre rationnel ou au moins une fraction dyadique $p/2^q$.
  $\rightarrow$ Les comparaisons entre dates deviennent coûteuses ($\mathcal{O}(w)$ dans le cas le pire).
- Solution intelligente : le problème de la liste ordonnée (*order maintenance problem*), voir plus loin.

Écriture dans une version quelconque du tableau

Lors d'une écriture de la valeur v à la date intermédiaire $t' = (t + t'')/2$, il peut être nécessaire d'ajouter deux entrées dans le gros élément :

- une à la date t' avec la valeur v
- l'autre à la date t'' avec la valeur de l'élément à la date t.

Exemple avec $t = 1$ et $t'' = 2$:

Gros élément	Valeur à la date				
	0	1	2	3	
$(0, \mathbf{a}); (3, \mathbf{b})$	**a**	**a**	**a**	**b**	
$(0, \mathbf{a}); (1, 5, \mathbf{c}); (3, \mathbf{b})$	**a**	**a**	**c**	**b**	✘
$(0, \mathbf{a}); (1, 5, \mathbf{c}); (2, \mathbf{a}); (3, \mathbf{b})$	**a**	**a**	**a**	**b**	✔

Écriture d'un élément dans une version quelconque

```
let set (t, c) i v =
  match QSet.find_first_opt (fun t' -> Q.gt t' t) c.timestamps with
  | None ->
      (* Updating the latest version of the array *)
      ...
  | Some t'' ->
      (* Updating an earlier version *)
      let t' = Q.(div (add t t'') (of_int 2)) in
      c.timestamps <- QSet.add t' c.timestamps;
      let e = c.arr.(i) in
      let e1 =
        if QMap.mem t'' e then e else QMap.add t'' (get_elt t e) e in
      let e' = QMap.add t' v e1 in
      c.arr.(i) <- e';
      (t', c)
```

Analyse dans le cas de la persistance complète

Quand le nombre de versions atteint n, on peut diviser le tableau en deux tableaux contenant chacun $\approx n/2$ versions.

(Voir O'Neill et Burton pour plus de détails.)

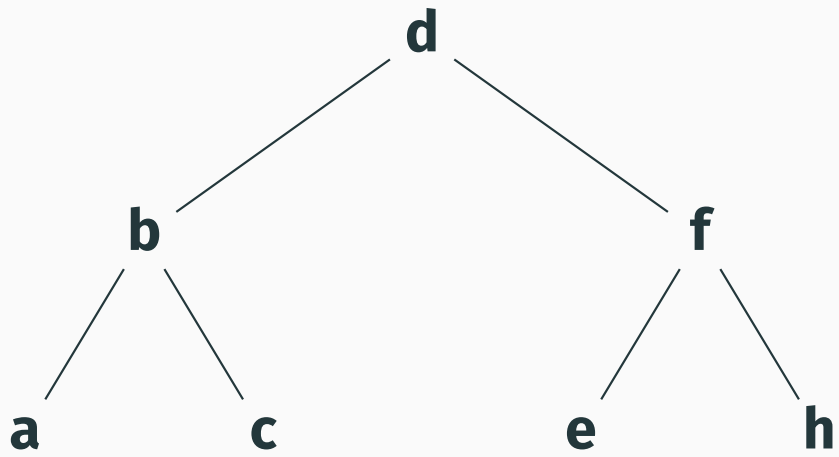
L'analyse précédente s'applique encore :

- lecture en temps garanti $\mathcal{O}(\log n)$
- écriture en temps amorti $\mathcal{O}(\log n)$
- taille du tableau $\mathcal{O}(n + w)$ si les gros éléments sont représentés par des arbres équilibrés modifiés en place.

… à condition d'avoir une solution efficace au problème de la liste ordonnée, c.à.d. des dates représentables par un nombre fixe de mots machine et que l'on peut comparer en temps $\mathcal{O}(1)$.

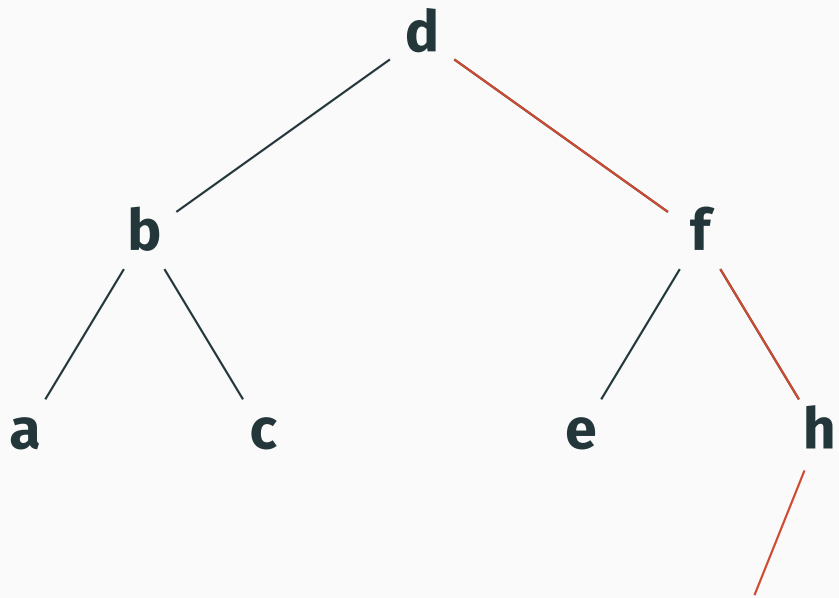
Arbres binaires persistants : utilisation de «gros nœuds»

Insertion dans un A.B.R. : version impérative (rappel du 2e cours)



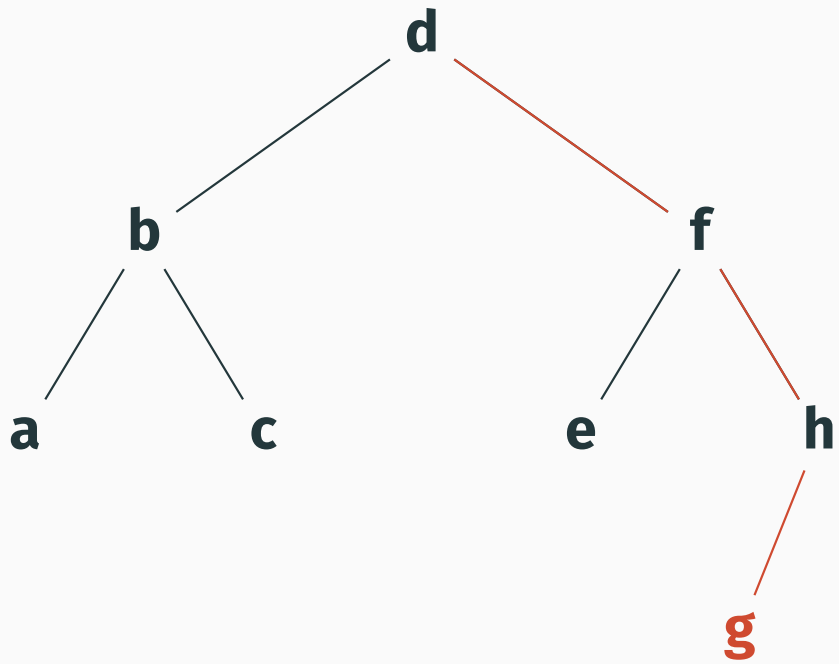
1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.

Insertion dans un A.B.R. : version impérative (rappel du 2e cours)



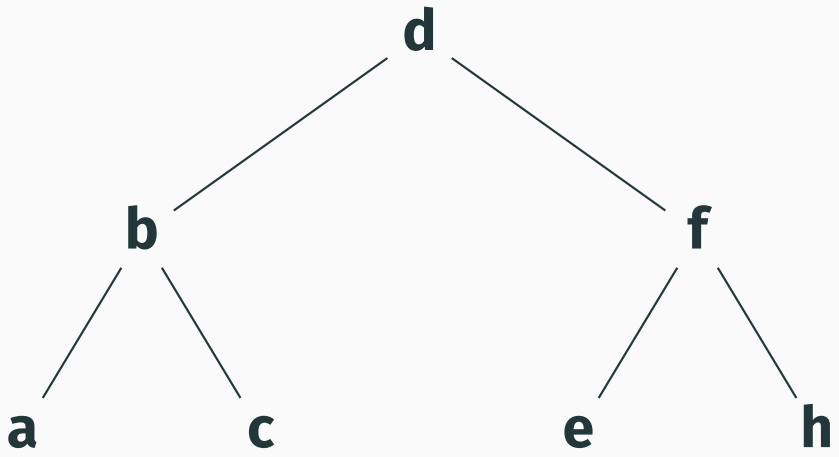
1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.

Insertion dans un A.B.R. : version impérative (rappel du 2e cours)



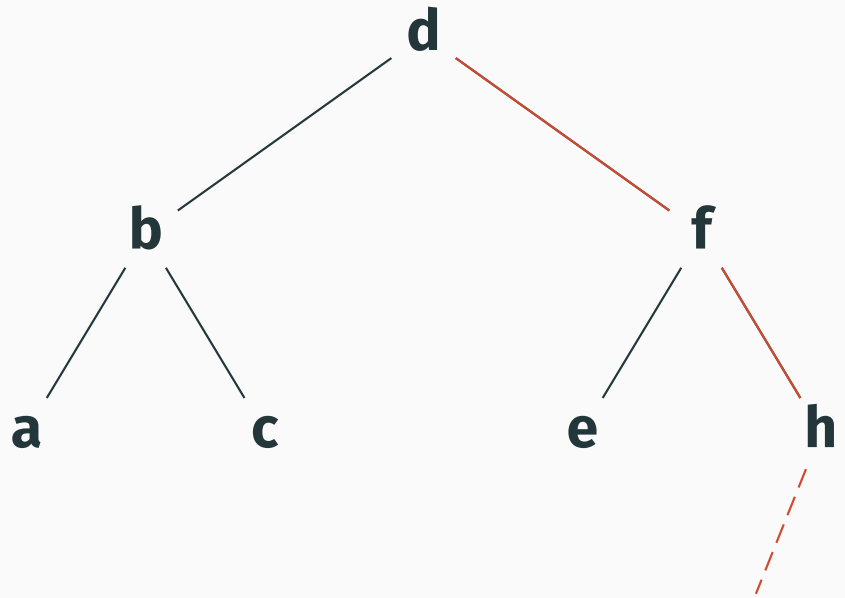
1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.
2. Quand on tombe sur une feuille, la remplacer par le nœud $\langle \bullet, \mathbf{g}, \bullet \rangle$.

Insertion dans un A.B.R. : version persistante (rappel du 2e cours)



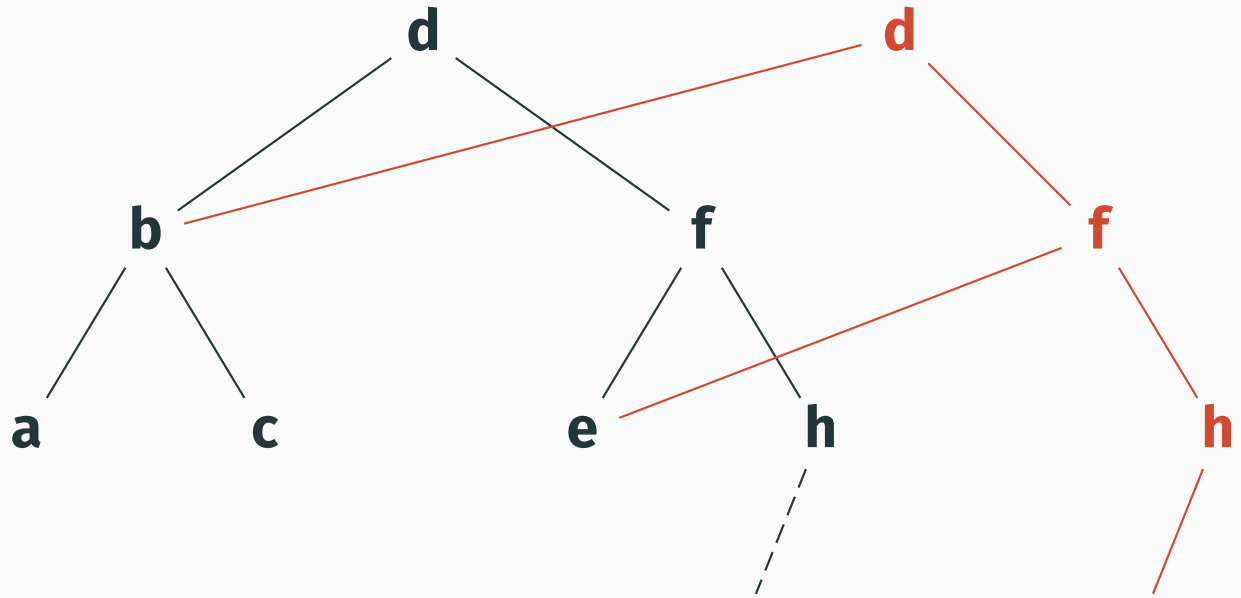
1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.

Insertion dans un A.B.R. : version persistante (rappel du 2e cours)



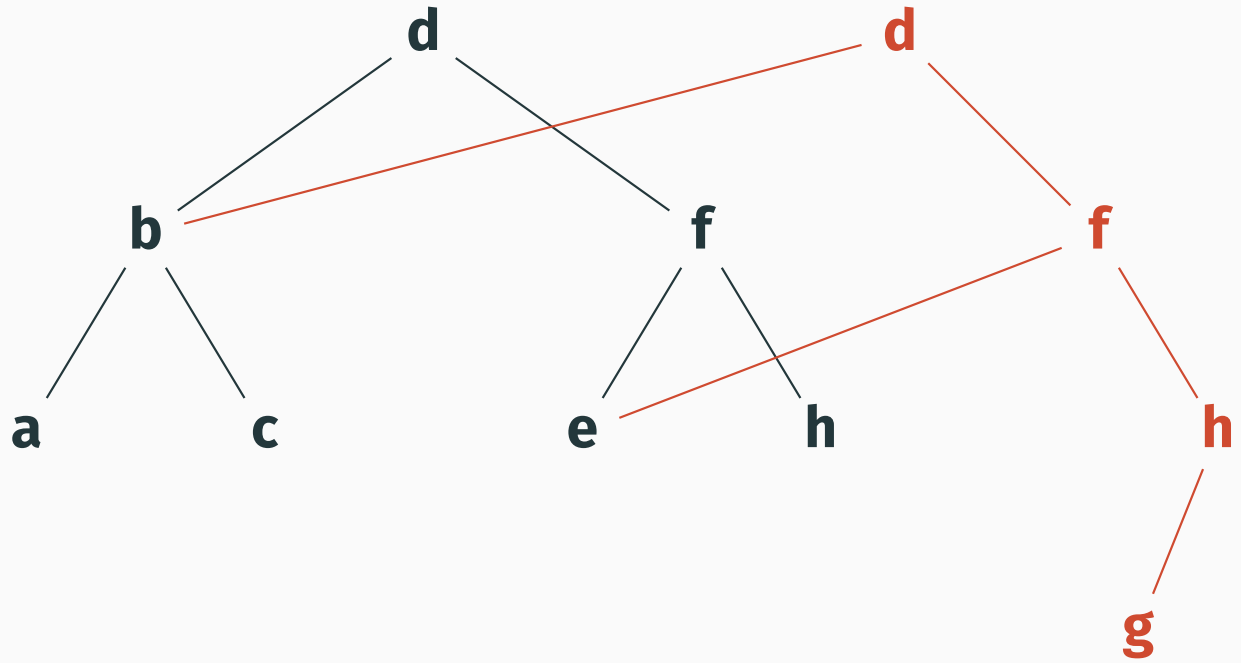
1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.

Insertion dans un A.B.R. : version persistante (rappel du 2e cours)



1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.
2. Quand on tombe sur une feuille, copier le chemin de la racine vers cette feuille, en partageant les sous-arbres de l'arbre initial.

Insertion dans un A.B.R. : version persistante (rappel du 2e cours)



1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.
2. Quand on tombe sur une feuille, copier le chemin de la racine vers cette feuille, en partageant les sous-arbres de l'arbre initial.
3. À la fin du chemin copié, ajouter le nœud $\langle \bullet, \mathbf{g}, \bullet \rangle$.

Insertion dans un arbre binaire de recherche

L'implémentation purement fonctionnelle (par copie de branche) est aussi efficace en temps que l'implémentation impérative, mais moins efficace en espace :

	Temps	Espace
Purement fonctionnelle	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Impérative	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

Si on fait n insertions depuis l'arbre vide, l'ensemble des n versions successives de l'arbre occupe donc un espace $\mathcal{O}(n \log n)$ avec l'implémentation fonctionnelle.

N. Sarnak et R. E. Tarjan (1986) : on peut descendre à $\mathcal{O}(n)$ en rendant persistante l'implémentation impérative.

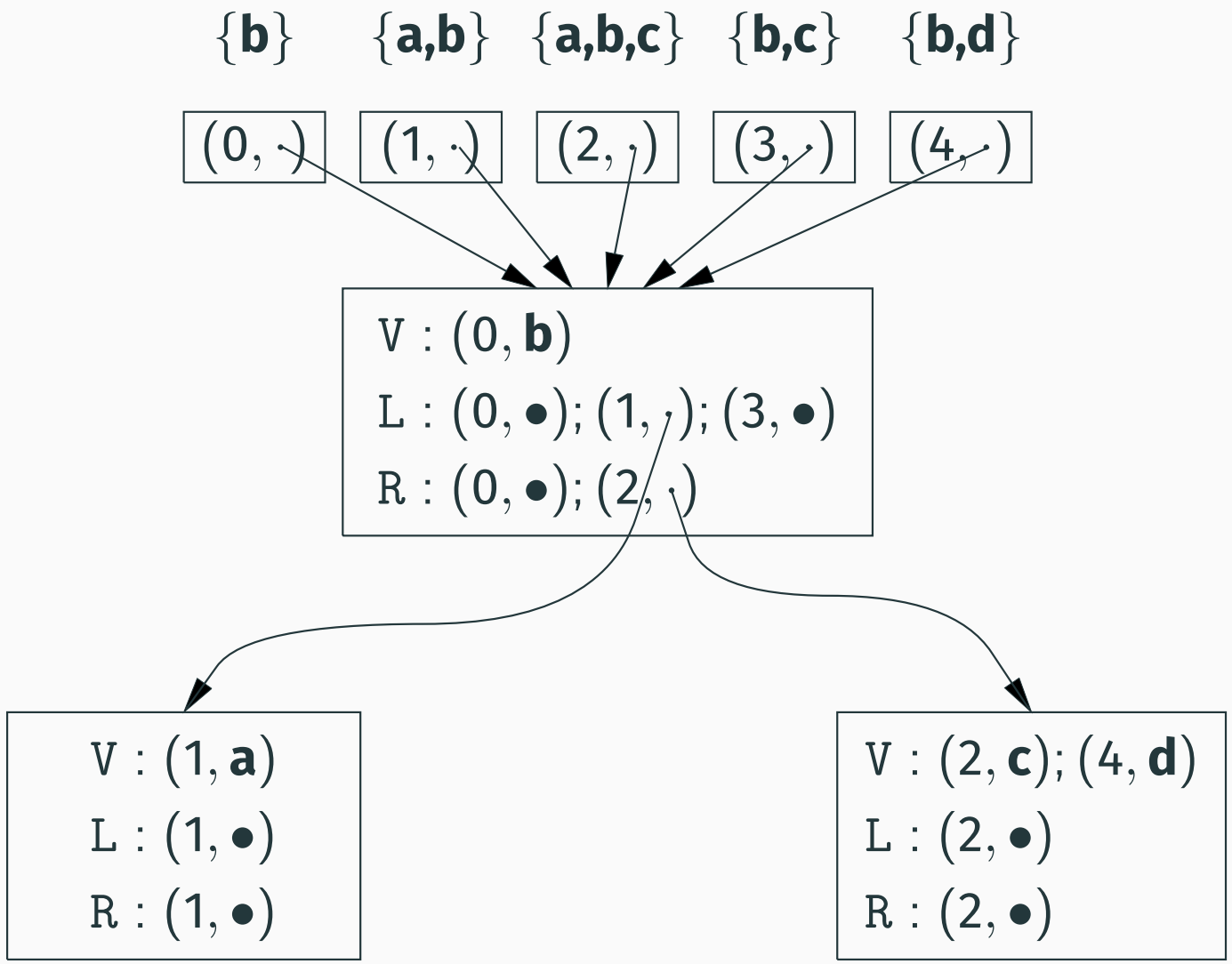
Un A.B.R. avec des gros nœuds (*fat nodes*)

Un gros nœud (*fat node*) = un enregistrement

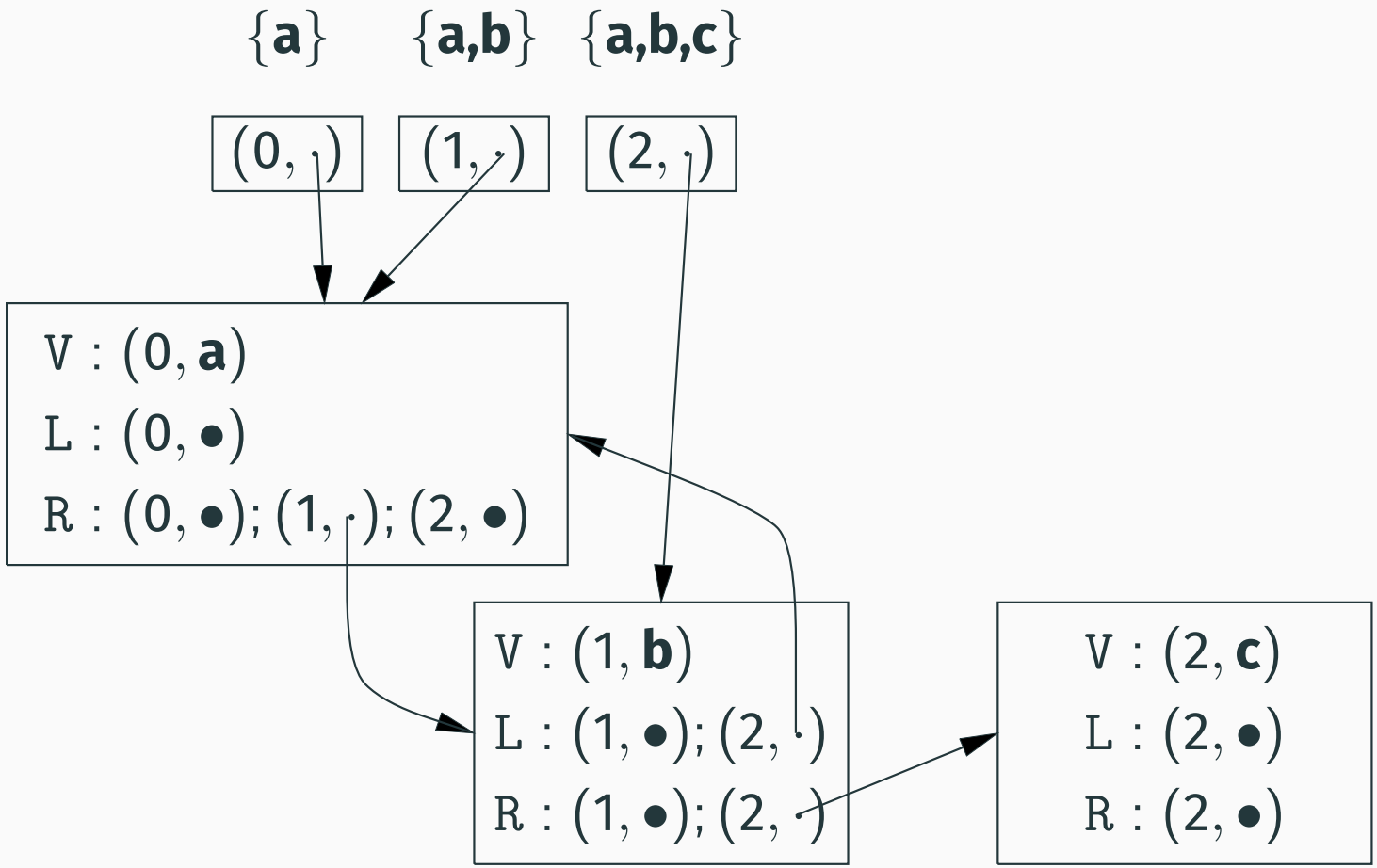
- champ `V` : historique (date, valeur)
- champ `L` : historique (date, sous-arbre gauche)
- champ `R` : historique (date, sous-arbre droit)
- informations d'équilibrage : hauteur, couleur rouge-noir, etc

Initialement, on vise uniquement la persistance partielle, donc les informations d'équilibrage, qui ne servent que pour l'écriture, n'ont pas besoin d'être versionnées et s'appliquent à la dernière version uniquement.

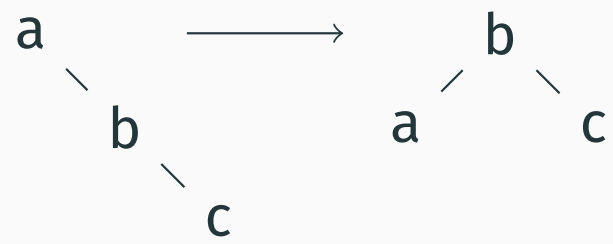
Exemple d'A.B.R. avec des gros nœuds



Rééquilibrage par rotations pendant l'insertion



Correspond à la rotation



Analyse

Après n insertions depuis l'arbre vide :

Recherche d'un élément : temps $\mathcal{O}(\log^2 n)$

(on parcours une branche de longueur $\log n$, et à chaque nœud on consulte un historique de taille au plus n).

Insertion d'un élément : même temps + espace $\mathcal{O}(1)$

(chaque insertion crée un nœud et modifie un champ,
puis effectue un petit nombre de rotations :
maximum 2 rotations pour un arbre rouge-noir).

Les n versions occupent donc un espace $\mathcal{O}(n)$.

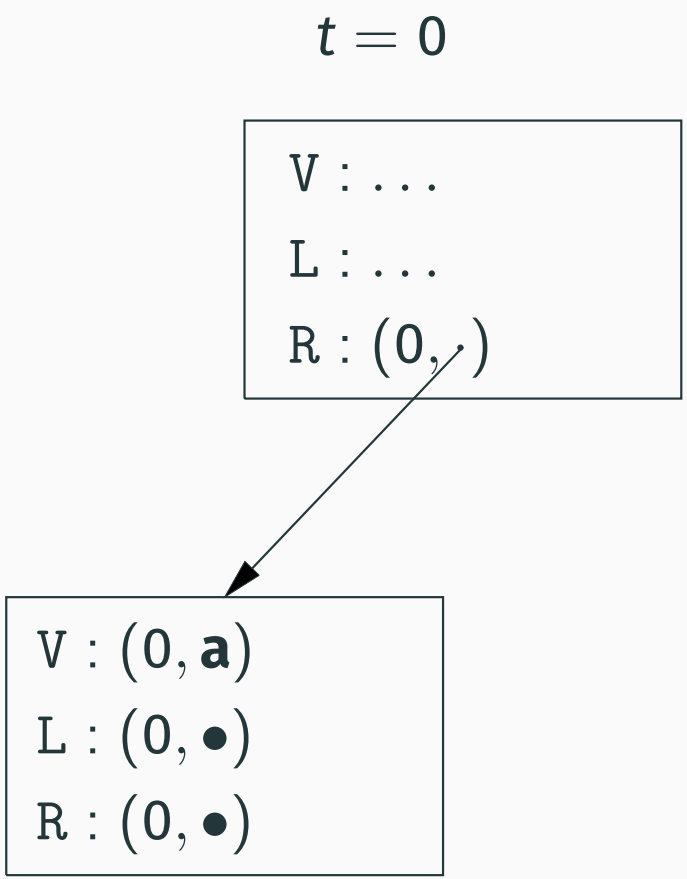
Limiter la taille des historiques

Supposons que chaque historique (pour `V`, `L`, `R`) contienne au plus k entrées. La recherche dans un historique est en temps constant, et la recherche dans l'arbre entier est en temps $\mathcal{O}(\log n)$.

Comment modifier un champ `V`/`L`/`R` d'un gros nœud ?

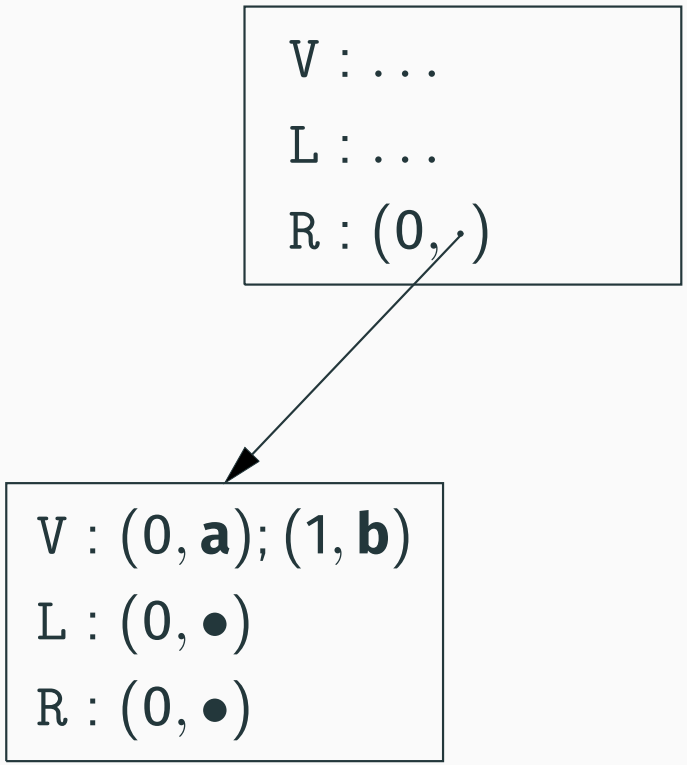
- Si l'historique correspondant n'est pas plein, on ajoute une entrée dans l'historique (temps constant).
- Si l'historique est plein, on crée une copie du nœud avec des historiques à une entrée (les nouvelles valeurs de `V`,`L`,`R`), et on modifie récursivement le champ `L` ou `R` du nœud parent.

Exemple de modifications avec historiques de taille $k = 2$

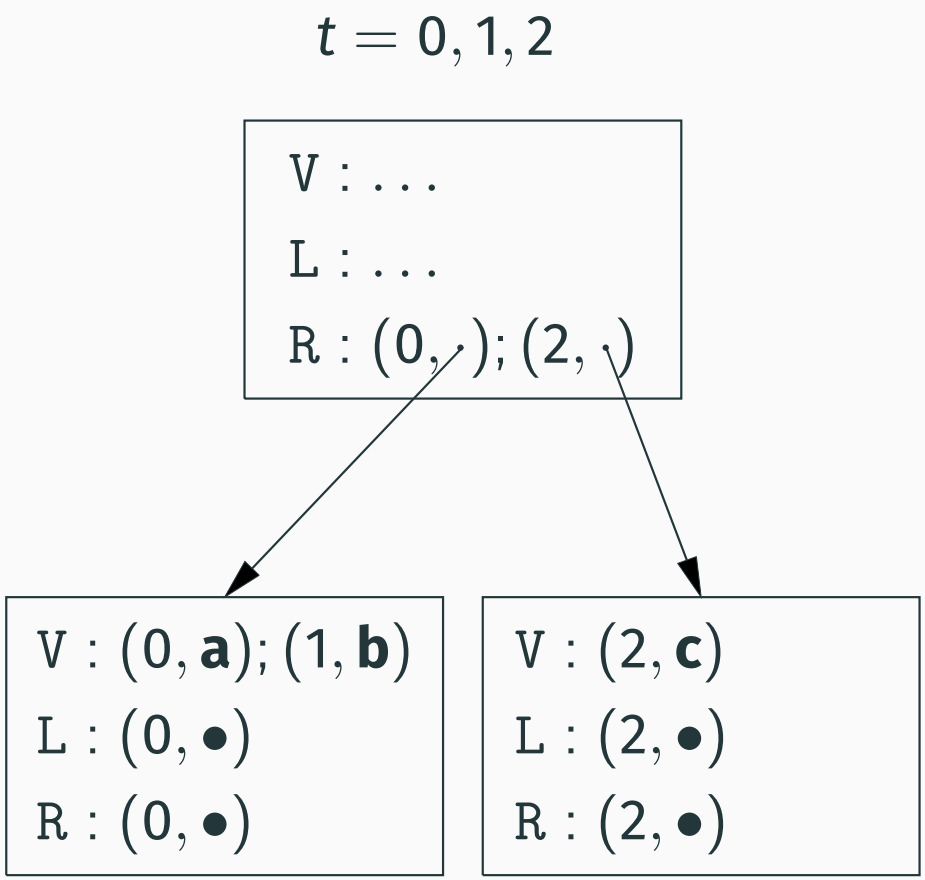


Exemple de modifications avec historiques de taille $k = 2$

$t = 0, 1$

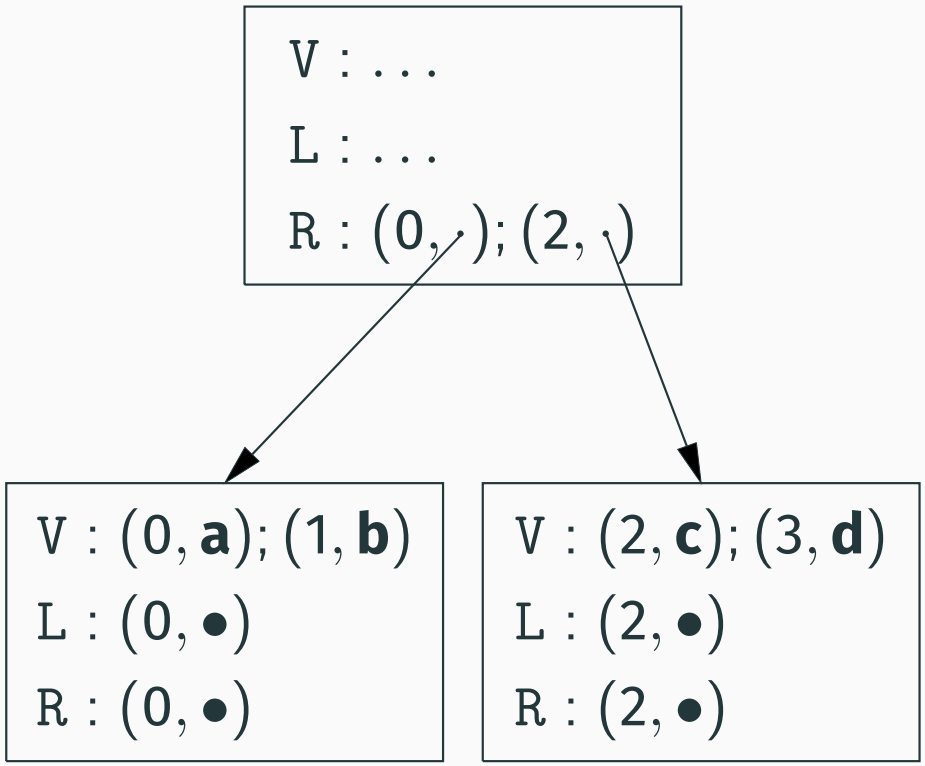


Exemple de modifications avec historiques de taille $k = 2$

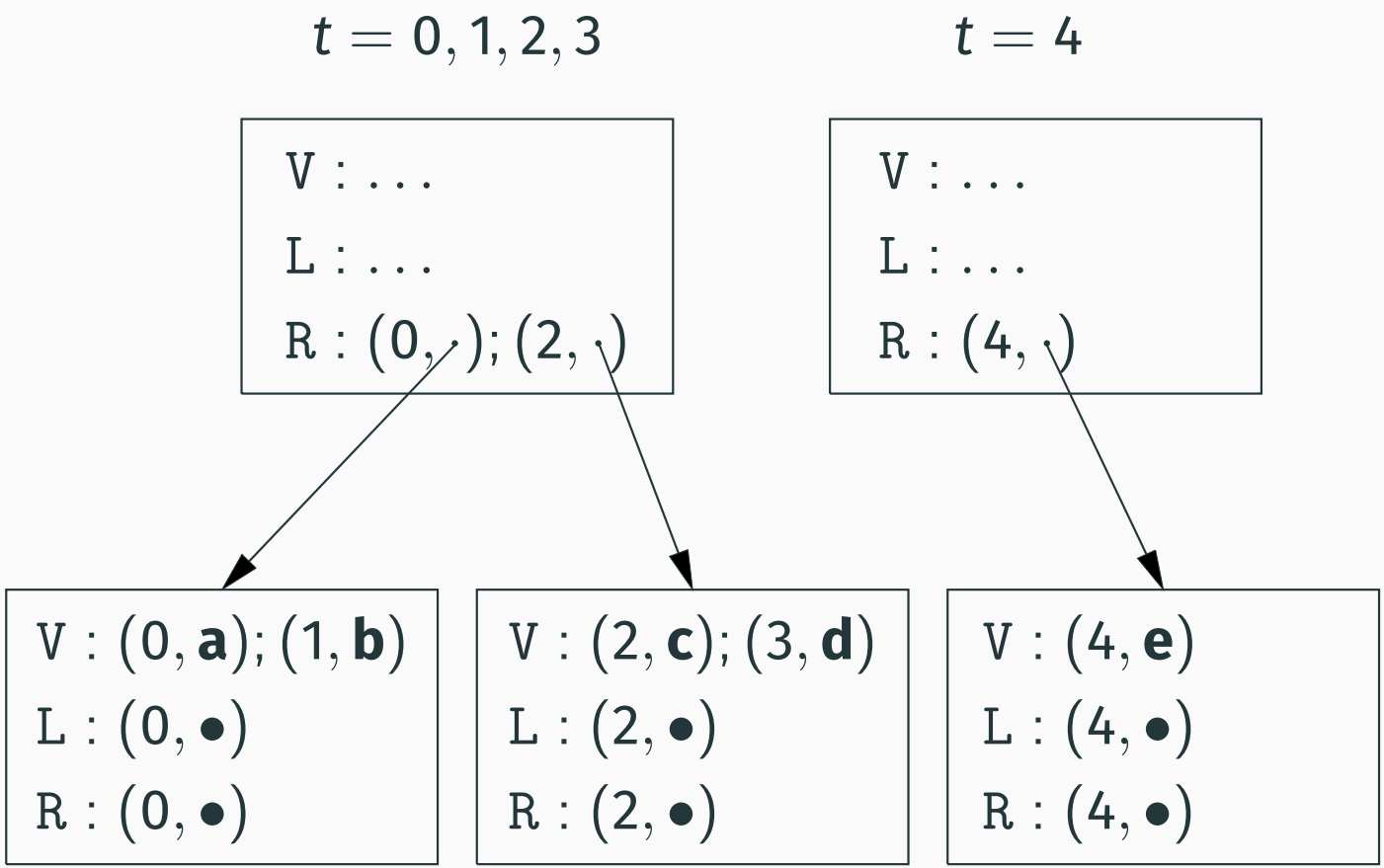


Exemple de modifications avec historiques de taille $k = 2$

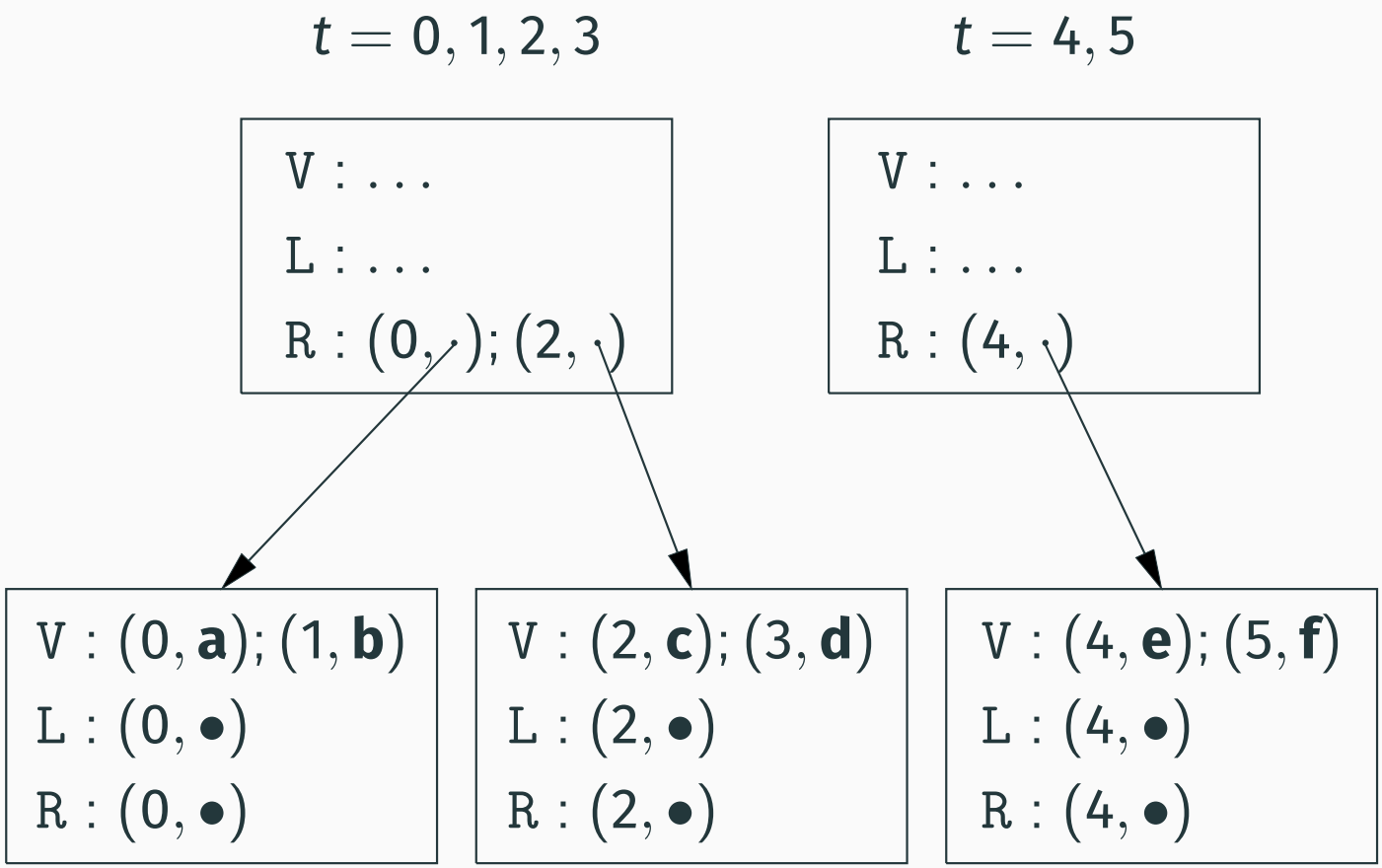
$t = 0, 1, 2, 3$



Exemple de modifications avec historiques de taille $k = 2$



Exemple de modifications avec historiques de taille $k = 2$



Chaque modification de V fait en moyenne 2 écritures ou créations de nouveau bloc $\rightarrow$ temps et espace $\mathcal{O}(1)$ amorti.

Analyse amortie (Sarnak et Tarjan, 1986)

Dans le modèle de Sarnak et Tarjan, chaque gros nœud a 3 champs `V`, `L`, `R` qui contiennent leur valeur la plus récente, et k cases d'historique «indifférenciées» contenant d'anciennes valeurs datées de `V`, `L` ou `R`.

Le potentiel de la structure de donnée est

$$\Phi = \text{nombre de nœuds} - \frac{\text{nombre de cases libres}}{k}$$

Allocation d'un nouveau nœud ou copie d'un nœud existant : nombre de nœuds +1, nombre de cases libres +k, donc $\Delta\Phi = 0$.

Écriture sans copie de nœud : $\Delta\Phi = 1/k$.

Pour chaque insertion dans l'arbre, $\Delta\Phi$ est borné par une constante, et l'espace amorti est $\mathcal{O}(1)$.

Extension à la persistance complète

Trois principaux changements :

1. Une écriture dans un champ peut nécessiter deux insertions dans son historique, comme on l'a vu pour les tableaux de O'Neill et Burton.
2. Lorsque l'historique déborde, au lieu de créer un nouveau nœud vide, il faut diviser le nœud en deux nœuds à moitié pleins.
3. Il faut ≪versionner≫ aussi les informations d'équilibrage (couleurs rouge/noir). Jusqu'à $\log n$ couleurs peuvent changer à chaque insertion.

   (Cela gêne Driscoll, Sarnak, Sleator et Tarjan (1989) au point qu'ils décrivent un algorithme de recoloriage paresseux des arbres rouge-noir qui permettrait de contourner ce problème.)

Rendre persistante une structure éphémère

Making Data Structures Persistent

Un célèbre article de Driscoll, Sarnak, Sleator et Tarjan (1989) qui montre comment partir de

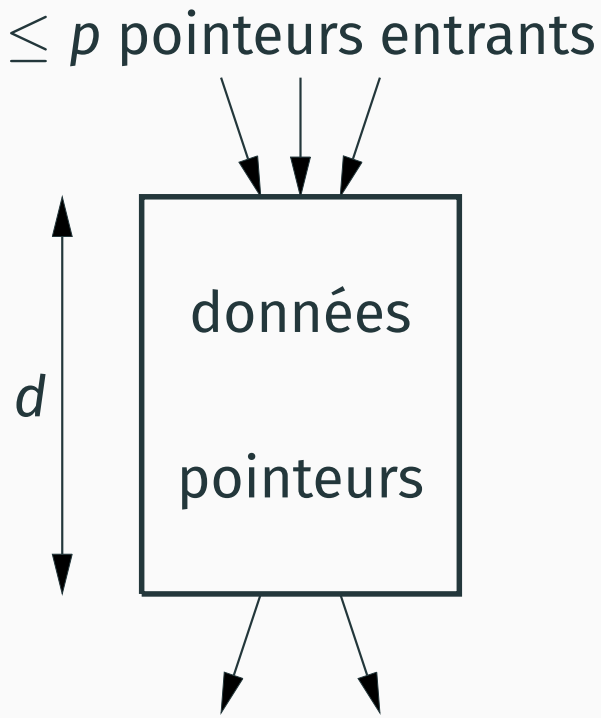
une structure éphémère presque arbitraire
(graphe orienté dont les sommets sont de degré borné)

et la transformer en

une structure ayant les mêmes opérations,
mais partiellement ou complètement persistante.

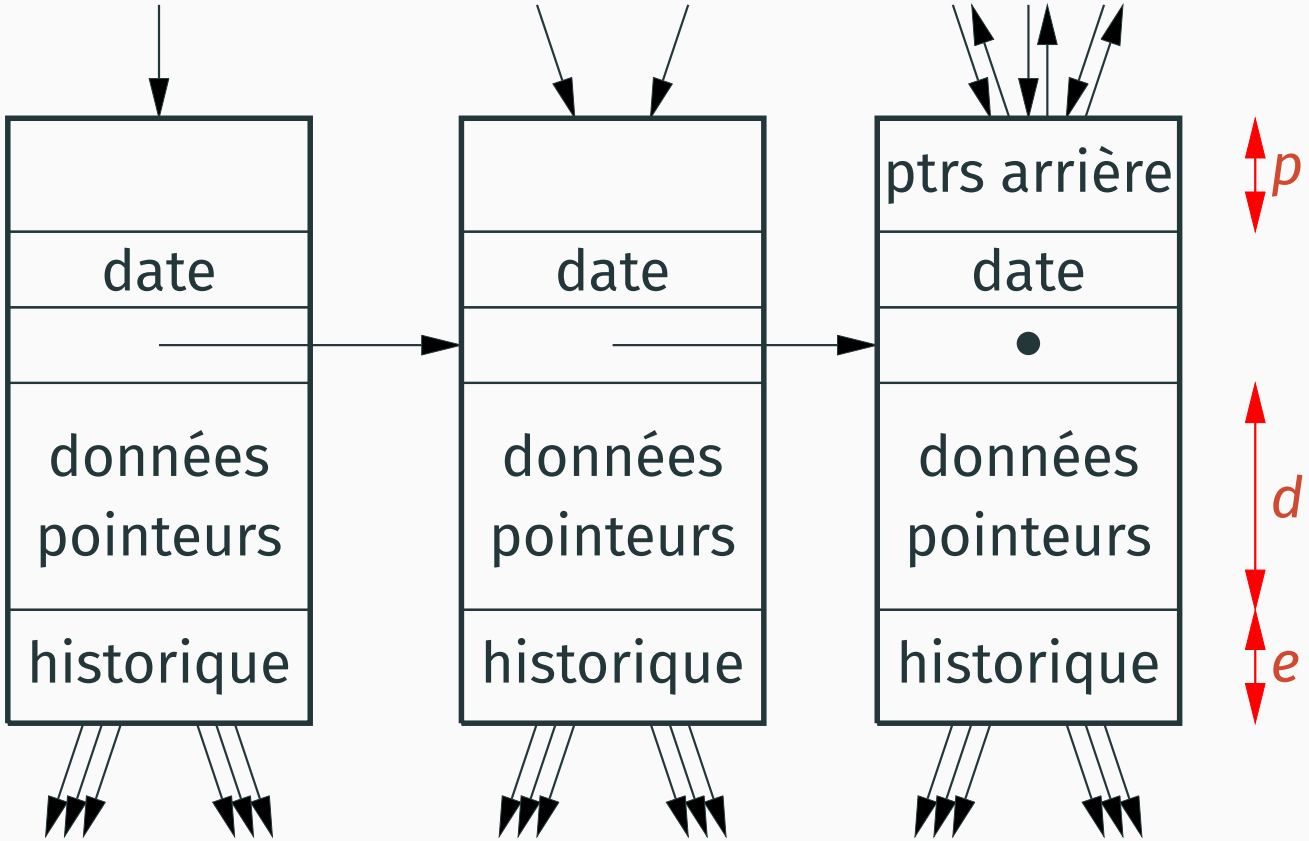
Mêmes ingrédients que pour les arbres rouge-noir :
≪gros nœuds≫ avec historiques de taille fixe
+ copie ou division de nœuds.

Un nœud de la structure éphémère



Chaque nœud contient d champs, qui peuvent être soit des données brutes soit des pointeurs vers d'autres nœuds.

Les gros nœuds correspondants (cas de la persistance partielle)



Les historiques contiennent jusqu'à e entrées de la forme (nom du champ, date, nouvelle valeur).

Seul le dernier gros nœud de la chaîne peut être modifié et contient des pointeurs arrières.

Écriture dans un champ

Même approche que dans l'exemple des arbres rouge-noir partiellement persistants :

- S'il reste de la place dans l'historique, écrire dedans.
- Sinon créer un nouveau nœud et écrire un nouveau pointeur dans les nœuds parents.

Principale différence : il faut utiliser les pointeurs arrière (pour localiser les nœuds parents) et les mettre à jour.

Écriture dans un champ

Écrire la valeur v dans le champ f du gros nœud x à la date i :

- Si x contient déjà une valeur pour f à la date i, la mettre à jour.
- Sinon, s'il reste de la place dans l'historique, ajouter une entrée (f, i, v).
- Sinon, créer un nouveau nœud y, avec un historique vide, le champ f initialisé à v, et les autres champs aux valeurs qu'ils ont dans x à la date $i - 1$.
- Dans le dernier cas, utiliser les pointeurs arrière de x pour mettre à jour les pointeurs entrants : si le champ g de z pointe vers x à la date $i - 1$, écrire dans $z.g$ un pointeur vers y à la date i.
- Dans tous les cas, si f est de type pointeur, mettre à jour les pointeurs arrière des blocs pointés par f aux dates $i - 1$ et i.

Analyse amortie

Chaque écriture est en temps amorti $\mathcal{O}(1)$, et donc en espace amorti $\mathcal{O}(1)$ aussi, à condition que l'historique soit assez grand : $e \geq p$.

Intuition : à chaque fois que l'on copie un nœud, on gagne e entrées d'historique libres, qui vont compenser les p mises à jour de pointeurs entrants requises.

Extension à la persistance complète

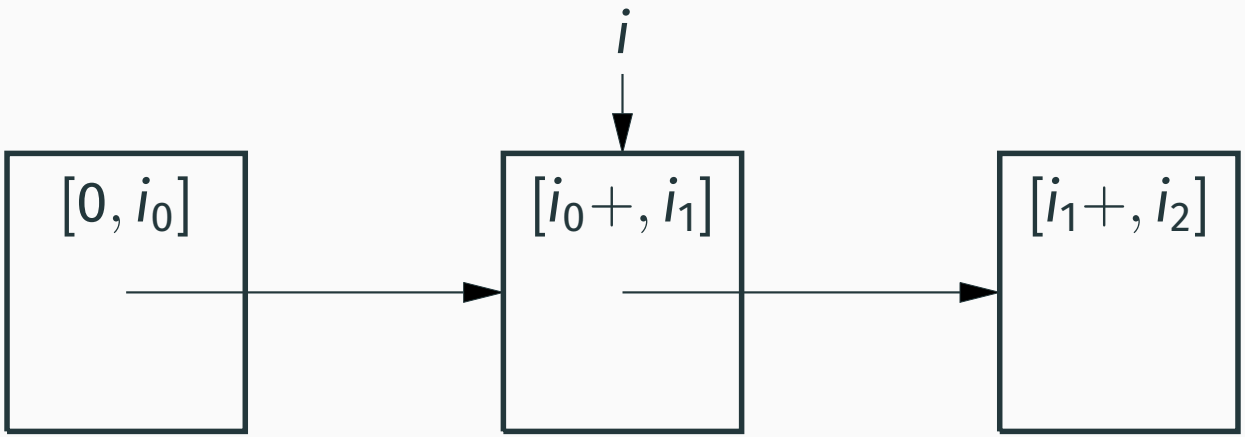
Trois principaux changements :

1) Si on peut écrire dans n'importe quelle version, il faut des pointeurs inverses pour toutes les versions, pas juste la dernière
$\rightarrow$ on ≪versionne≫ les ptrs inverses comme les autres ptrs.

2) Une écriture dans un champ peut nécessiter deux insertions dans son historique.

3) Au lieu de couper un nœud plein en un nœud plein + un nœud vide, on le divise en deux nœuds à moitié pleins.

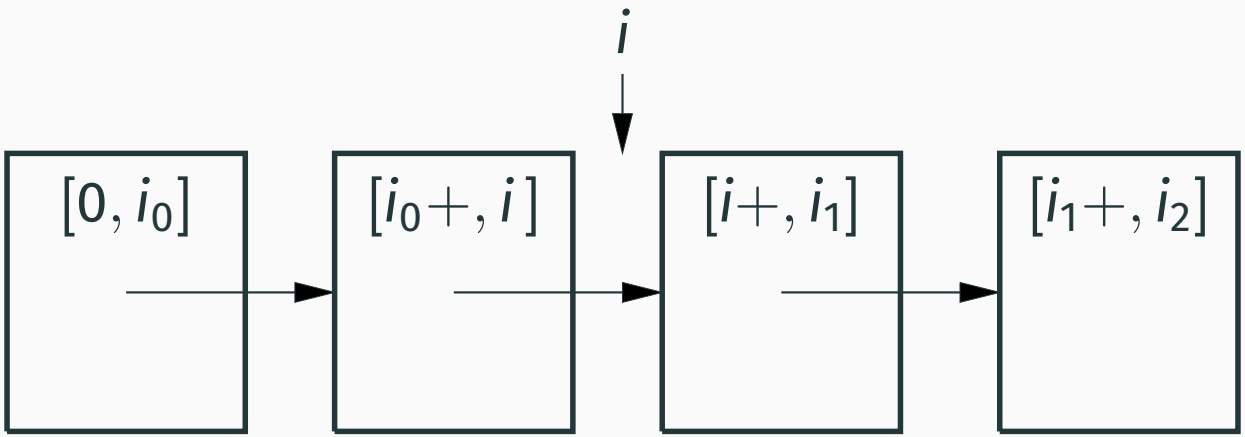
Chaînage et division des nœuds



À chaque nœud de la structure éphémère correspond une liste chaînée de gros nœuds, chacun valide pour un intervalle de dates.

La division d'un nœud sur une date i insère un nouveau nœud contenant l'historique depuis la date $i+$, laissant dans l'ancien nœud l'historique jusqu'à la date i.

Chaînage et division des nœuds



À chaque nœud de la structure éphémère correspond une liste chaînée de gros nœuds, chacun valide pour un intervalle de dates.

La division d'un nœud sur une date i insère un nouveau nœud contenant l'historique depuis la date $i+$, laissant dans l'ancien nœud l'historique jusqu'à la date i.

Tenir à jour les pointeurs

La division d'un nœud peut invalider à la fois des pointeurs avant et des pointeurs arrière ! D'où un algorithme itératif compliqué :

- Identifier les pointeurs qui sont devenus incorrects
  ($\approx$ pointeur dont l'intervalle de validité n'est plus inclus dans l'intervalle de validité du nœud pointé)
- Réécrire ces pointeurs pour les faire pointer vers la bonne version du nœud.
- Ces écritures peuvent provoquer de nouvelles divisions de nœuds.

L'écriture reste en temps $\mathcal{O}(1)$ amorti pourvu que $e \geq d + p$.

Le problème de la liste ordonnée

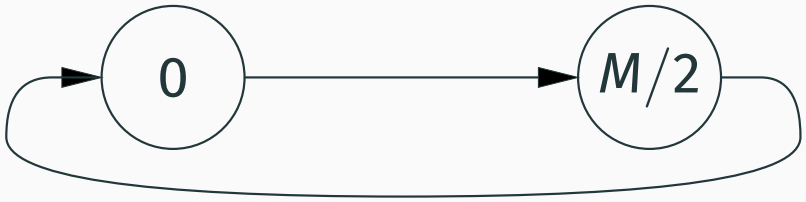
Le problème de la liste ordonnée (*order maintenance problem*)

Une liste ordonnée et deux opérations :

- Étant donnés deux éléments de la liste, déterminer leur position relative (lequel apparaît avant l'autre dans la liste).
- Insérer un nouvel élément juste après un élément donné.

On cherche des implémentations en temps amorti constant pour ces deux opérations.

Une implémentation naïve



Une liste circulaire mutable de cellules c portant 2 champs :

- $L(c)$: une étiquette entière dans $[0, M[$ avec $M = 2^{128}$ p.ex.
- $S(c)$: pointeur vers la prochaine cellule.

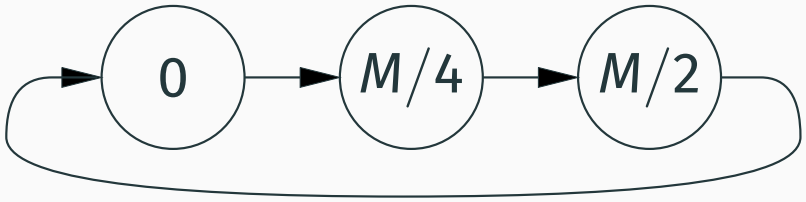
Déterminer la position relative des cellules c_1 et c_2 :

Comparer leurs étiquettes entières $L(c_1), L(c_2)$.

Insérer un nouvel élément juste après la cellule c :

Si $L(S(c)) \geq L(c) + 2$, lui donner une étiquette intermédiaire.
Sinon, renuméroter des cellules au voisinage de c
et recommencer.

Une implémentation naïve



Une liste circulaire mutable de cellules c portant 2 champs :

- $L(c)$: une étiquette entière dans $[0, M[$ avec $M = 2^{128}$ p.ex.
- $S(c)$: pointeur vers la prochaine cellule.

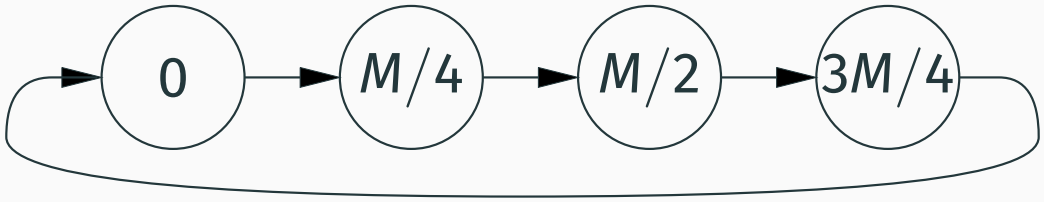
Déterminer la position relative des cellules c_1 et c_2 :

Comparer leurs étiquettes entières $L(c_1), L(c_2)$.

Insérer un nouvel élément juste après la cellule c :

Si $L(S(c)) \geq L(c) + 2$, lui donner une étiquette intermédiaire.
Sinon, renuméroter des cellules au voisinage de c
et recommencer.

Une implémentation naïve



Une liste circulaire mutable de cellules c portant 2 champs :

- $L(c)$: une étiquette entière dans $[0, M[$ avec $M = 2^{128}$ p.ex.
- $S(c)$: pointeur vers la prochaine cellule.

Déterminer la position relative des cellules c_1 et c_2 :

Comparer leurs étiquettes entières $L(c_1), L(c_2)$.

Insérer un nouvel élément juste après la cellule c :

Si $L(S(c)) \geq L(c) + 2$, lui donner une étiquette intermédiaire.

Sinon, renuméroter des cellules au voisinage de c

et recommencer.

Le critère de renumérotation de Dietz et Sleator (1987)

On définit l'écart (*gap*) entre deux cellules

$$g(c_1, c_2) \stackrel{def}{=} (L(c_2) - L(c_1)) \bmod M$$

Si on cherche à insérer après c et que $g(c, S(c)) = 1$:

On parcours la liste en avant de c pour trouver un intervalle de largeur j (minimale) tel que $g(c, S^j(c)) > j^2$.

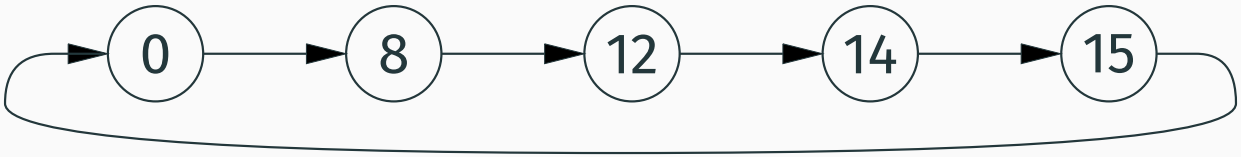
(C'est toujours possible si la liste contient au plus $\sqrt{M} = 2^{64}$ éléments.)

On renumérote les j éléments de l'intervalle en les espaçant régulièrement :

$$L(S^k(c)) := \left(L(c) + \left\lfloor \frac{k}{j} \times g(c, S^j(c)) \right\rfloor \right) \bmod M$$

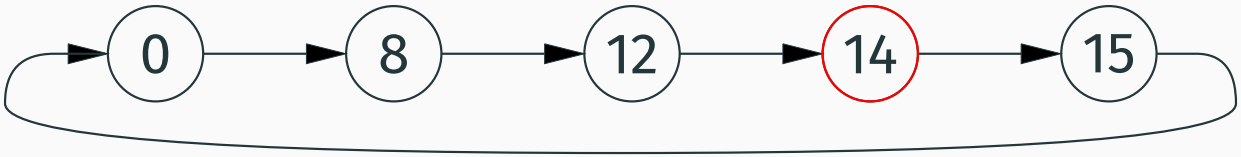
Exemple de renumérotation

Avec $M = 16$. On veut insérer après l'élément 14.



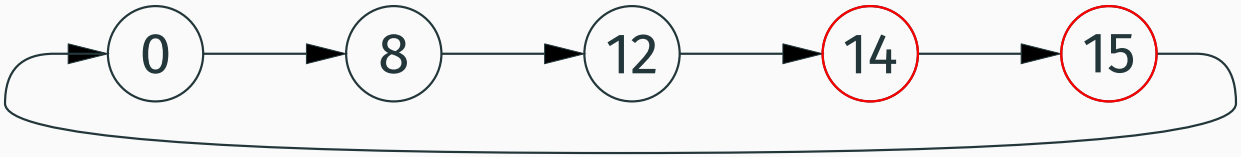
Exemple de renumérotation

Avec $M = 16$. On veut insérer après l'élément 14.



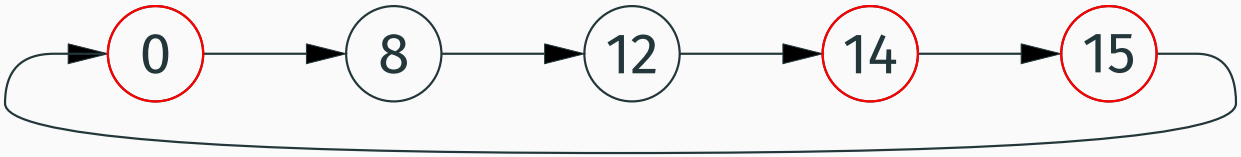
Exemple de renumérotation

Avec $M = 16$. On veut insérer après l'élément 14.



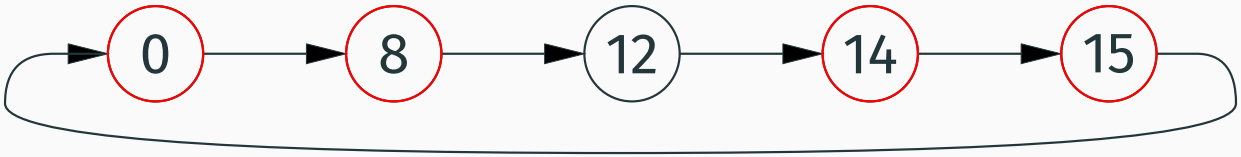
Exemple de renumérotation

Avec $M = 16$. On veut insérer après l'élément 14.



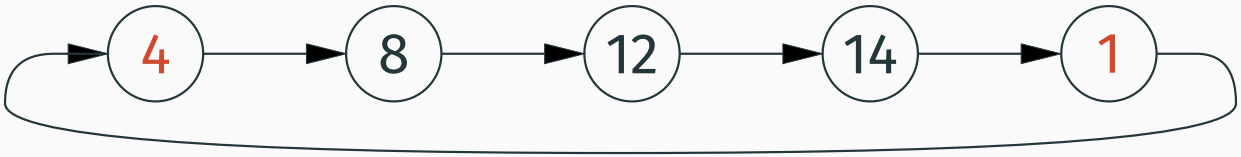
Exemple de renumérotation

Avec $M = 16$. On veut insérer après l'élément 14.



Exemple de renumérotation

Avec $M = 16$. On veut insérer après l'élément 14.



Renumérotation et positions relatives des cellules

La renumérotation peut ≪faire le tour≫ de la liste et affecter les premiers éléments. Il faut donc définir autrement l'ordre entre cellules :

Déterminer la position relative des cellules c_1 et c_2 :

Comparer les écarts à la base $g(base, c_1)$ et $g(base, c_2)$.

La renumérotation préserve les positions relatives !

	Étiquettes					Écarts à la base				
Avant renumérotation	0	8	12	14	15	0	8	12	14	15
Après renumérotation	4	8	12	14	1	0	4	8	10	13

Analyse amortie et ajout d'un niveau de liste

(P. F. Dietz, D. Sleator, *Two Algorithms for Maintaining Order in a List*, STOC 1987).

Une analyse particulièrement compliquée conclut que l'insertion prend un temps amorti $\mathcal{O}(\log n)$, où n est la taille de la liste.

On peut descendre à $\mathcal{O}(1)$ amorti avec une représentation à deux niveaux de listes :

- Chaque élément de la liste circulaire porte une sous-liste d'entiers avec au moins $\lceil \log n \rceil$ bits, p.ex. 64 bits, et de longueur maximale de l'ordre de $\log n$.
- On insère dans les sous-listes par l'algorithme naïf ≪la moyenne avec l'élément suivant≫.
- Quand on ne peut plus insérer dans une sous-liste, on la coupe en deux moitiés, on les renumérote, et on insère la 2^{e} moitié dans la liste principale.

Point d’étape

Rendre persistante une structure impérative éphémère

Une approche initiée par de beaux travaux des années 1980.

Elle n'a pas l'élégance algébrique de l'approche purement fonctionnelle, mais peut être plus performante dans des scénarios d'utilisation particuliers :

- persistance partielle ;
- semi-persistance ; ($\rightarrow$ séminaire J. C. Filliâtre)
- utilisation quasi linéaire.

La persistance complète est (trop ?) compliquée à atteindre.

L'approche permet surtout des gains en espace ($\mathcal{O}(n \log n) \rightarrow \mathcal{O}(n)$), alors qu'on s'attendait à des gains en temps de la part de la programmation impérative.

Bibliographie

Bibliographie

Une synthèse (dense mais très complète) de l'approche :

- H. Kaplan, *Persistent Data Structures*,
  chap. 31 du *Handbook of data structures and applications*,
  Chapman&Hall / CRC Press, 2005.

Les principaux articles discutés dans ce cours :

- M. E. O'Neill et F. W. Burton, *A new method for functional arrays*,
  J. Func. Prog. 7(5), 1997.
- N. Sarnak et R. E. Tarjan, *Planar Point Location using Persistent Search Trees*, Commun. ACM 29(7), 1986.
- J. Driscoll, N. Sarnak, D. Sleator et R. E. Tarjan, *Making Data Structures Persistent*, J. Comput. Syst. Sci. 38(1), 1989.
- P. Dietz et D. Sleator, *Two Algorithms for Maintaining Order in a List*, proceedings of STOC 1987, ACM.