Rien ne se perd, tout se crée: introduction aux structures de données persistantes

Xavier Leroy

2023-03-09

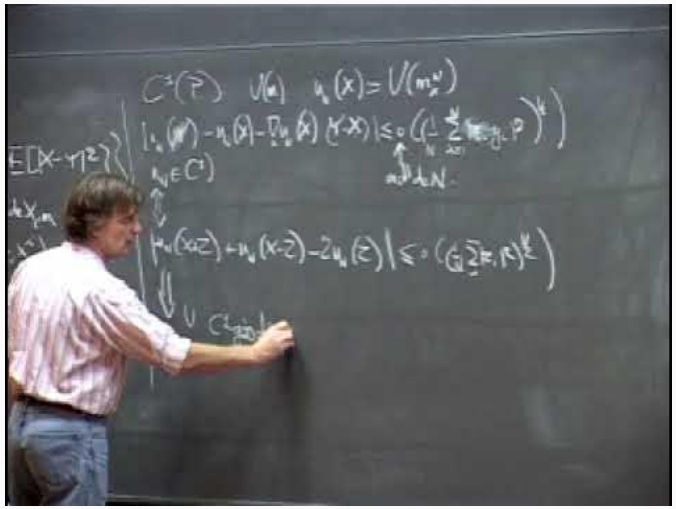
Collège de France, chaire de sciences du logiciel
xavier.leroy@college-de-france.fr

Un algorithme, c’est comme une recette de cuisine !

Conséquence naturelle : les données d’entrées et les résultats intermédiaires ne sont pas préservés pendant le calcul.

Formalisme fondamental : la machine de Turing.

Un algorithme, c’est une définition mathématique



Le calcul vu comme une suite d’étapes construisant le résultat final, sans jamais effacer les données d’entrées ni les résultats intermédiaires.

Formalismes fondamentaux : fonctions récursives générales, lambda-calcul, systèmes de réécriture, sémantiques formelles.

Programmation «déclarative» (par opposition à «impérative») : fonctionnelle, logique, par contraintes, …

Quelle algorithmique pour la programmation déclarative ?

La programmation déclarative est réputée inefficace car ne pouvant pas utiliser des structures de données éphémères (modifiée en place) (p.ex. les tableaux).

Les structures de données persistantes que nous allons voir dans ce cours répondent à cette critique :

- Leur interface n'expose que des opérations qui ne modifient pas la structure en place, mais renvoient de nouvelles structures modifiées.
- Leur implémentation atteint ou s'approche des complexités des meilleures structures éphémères connues.

L’intérêt de la persistance pour la programmation impérative

Même dans une programmation impérative et une algorithmique classique, les structures de données persistantes sont intéressantes :

- Le retour en arrière (*checkpointing*, *backtracking*) est trivial.
- On peut conserver l’historique complet de la structure.
- L’absence de modification en place autorise le partage en mémoire entre les versions successives de la structure, et donc une représentation compacte de son historique.

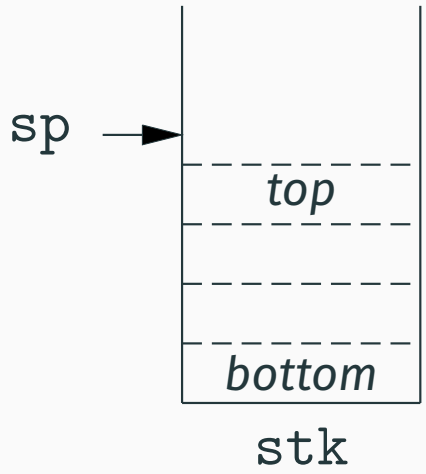
Exemple introductif : les piles

Une pile

Opérations :

`init`	vide la pile
`push`(v)	empile v au sommet
`top`	renvoie la valeur au sommet
`pop`	dépile la valeur au sommet.

Une pile éphémère implémentée par un tableau

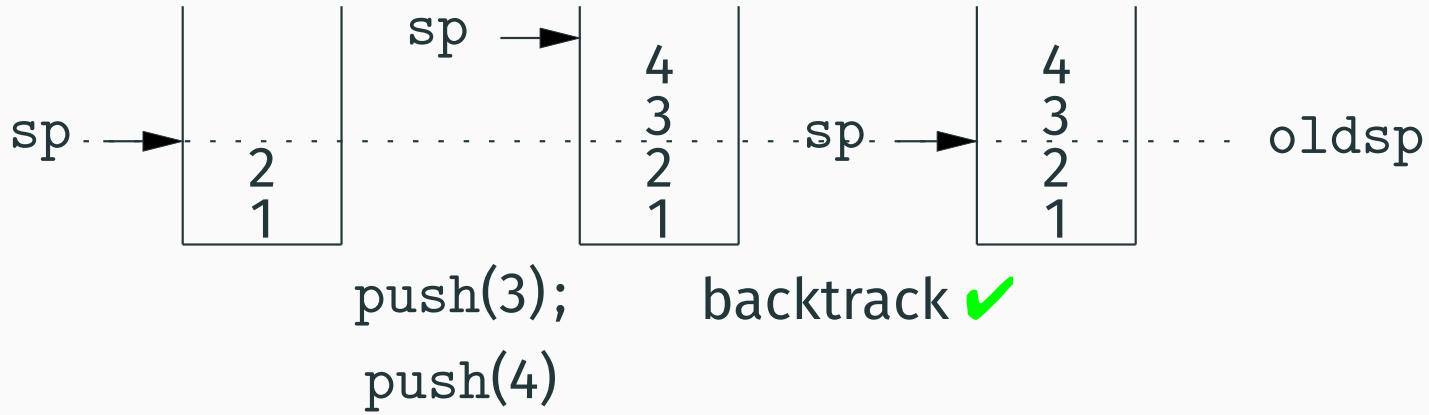



```
int stk[SIZE];
int sp;

void init(void) { sp = 0; }

void push(int v) {
  assert (sp < SIZE);
  stk[sp] = v; sp = sp + 1;
}
int top(void) {
  assert (sp > 0); return stk[sp - 1];
}
void pop(void) {
  assert (sp > 0); sp = sp - 1;
}
```

Retour en arrière (*checkpointing & backtracking*)

Approche efficace mais incomplète :

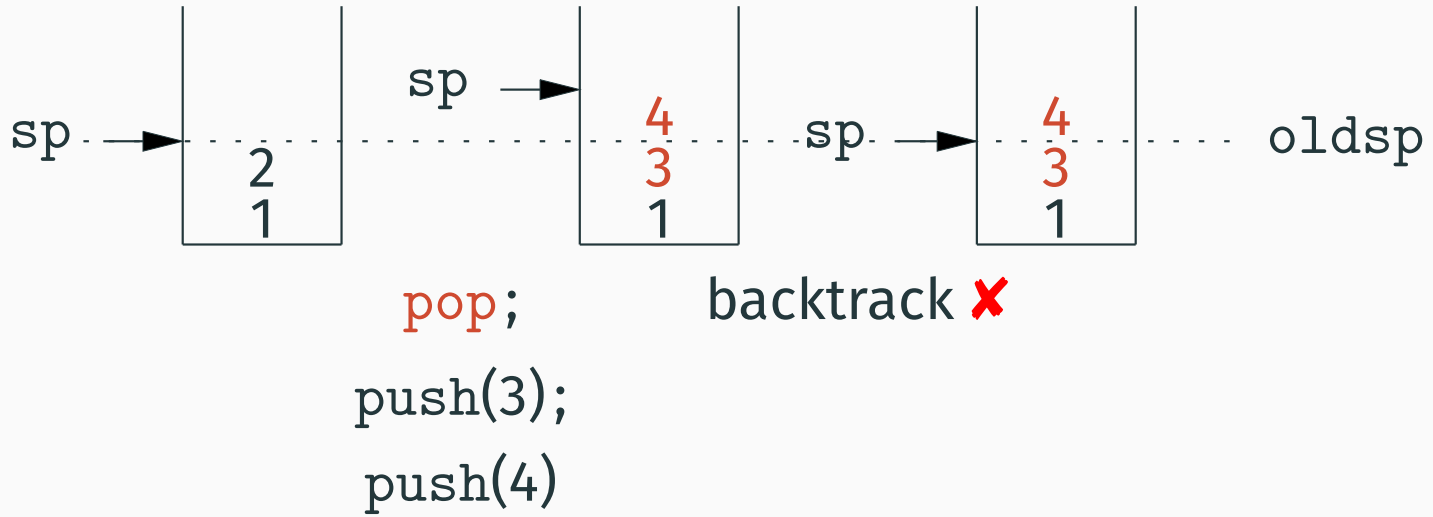
- Pose d'un *checkpoint* : `oldsp = sp;` (sauvegarder `sp`)
- Retour arrière (*backtracking*) : `sp = oldsp;` (restaurer `sp`)



Retour en arrière (*checkpointing & backtracking*)

Approche efficace mais incomplète :

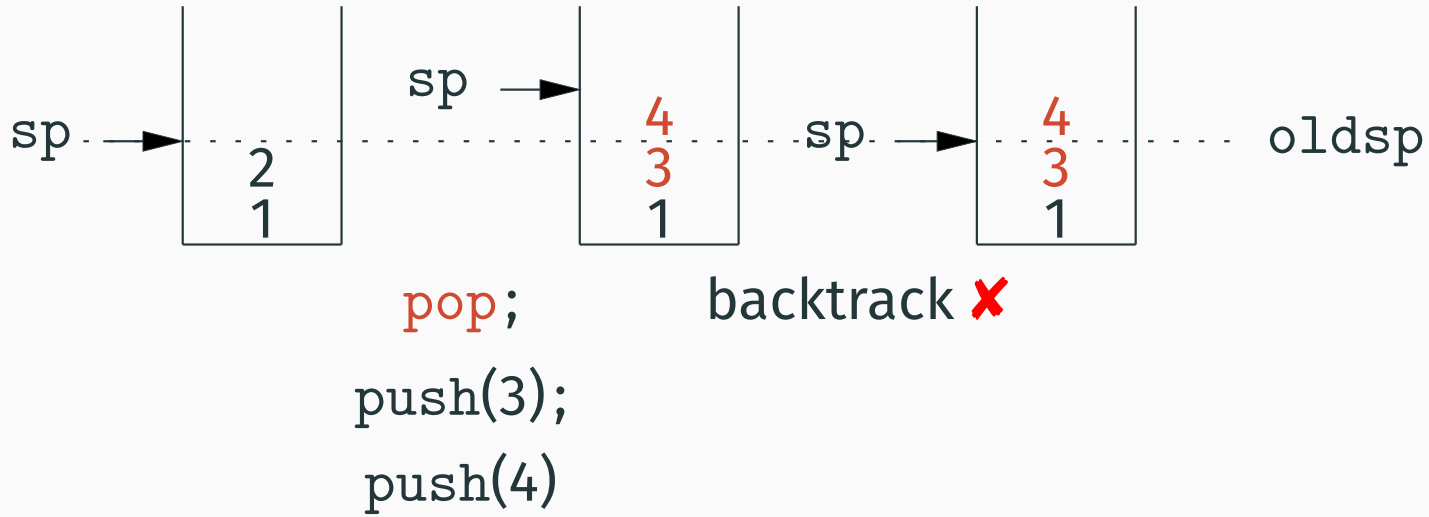
- Pose d'un *checkpoint* : `oldsp = sp;` (sauvegarder `sp`)
- Retour arrière (*backtracking*) : `sp = oldsp;` (restaurer `sp`)



Retour en arrière (*checkpointing & backtracking*)

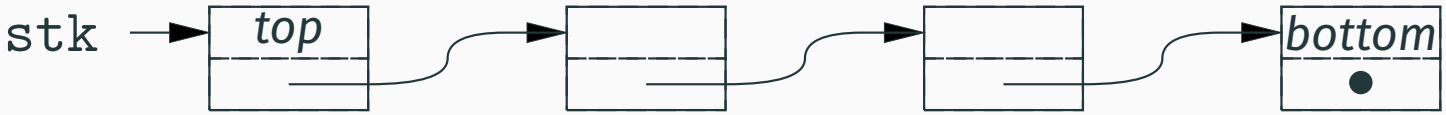
Approche efficace mais incomplète :

- Pose d'un *checkpoint* : `oldsp = sp;` (sauvegarder `sp`)
- Retour arrière (*backtracking*) : `sp = oldsp;` (restaurer `sp`)



Approche toujours correcte mais inefficace :
copier `stk`[0 . . . `sp`[dans un autre tableau.

Une pile persistante implémentée par une liste chaînée

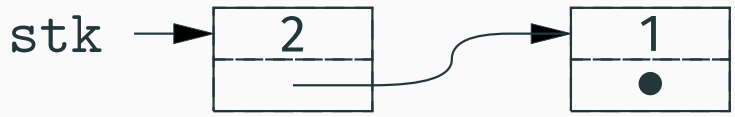



```
class Stack {
  private int hd; private Stack tl;
  static Stack empty = null;
  static Stack push(int v, Stack s)
  { Stack t = new Stack(); t.hd = v; t.tl = s; return t; }
  static int top(Stack s) { return s.hd; }
  static Stack pop(Stack s) { return s.tl; }
}
```

L'interface change : `push`, `pop` renvoient la nouvelle pile en résultat au lieu de modifier la pile passée en argument.

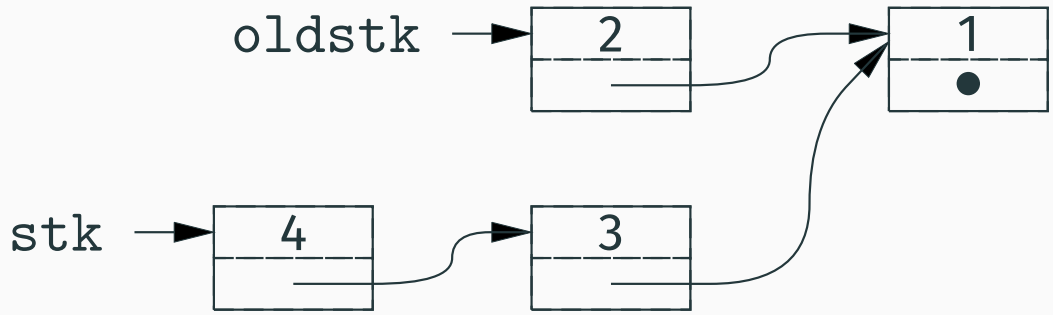
On s'appuie sur la gestion mémoire automatique du langage pour récupérer les cellules de listes après `pop`.

Piles persistantes et retour en arrière



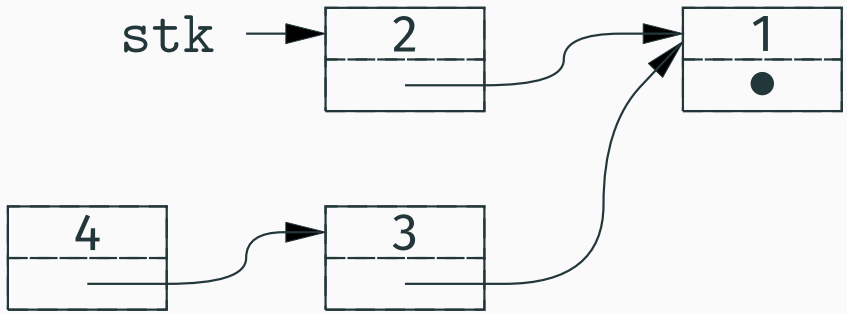

```
stk = Stack.push(2, Stack.push(1, Stack.empty));
```

Piles persistantes et retour en arrière




```
stk = Stack.push(2, Stack.push(1, Stack.empty));
// Checkpoint
oldstk = stk;
// Work
stk = Stack.push(4, Stack.push(3, Stack.pop(stk)));
```

Piles persistantes et retour en arrière




```
stk = Stack.push(2, Stack.push(1, Stack.empty));
// Checkpoint
oldstk = stk;
// Work
stk = Stack.push(4, Stack.push(3, Stack.pop(stk)));
// Backtrack
stk = oldstk;
```

Implémentation fonctionnelle pure des piles persistantes

En Lisp, Scheme, etc : avec les opérations primitives des listes

`empty` $\equiv$ `nil` `push` $\equiv$ `cons` `top` $\equiv$ `car` `pop` $\equiv$ `cdr`

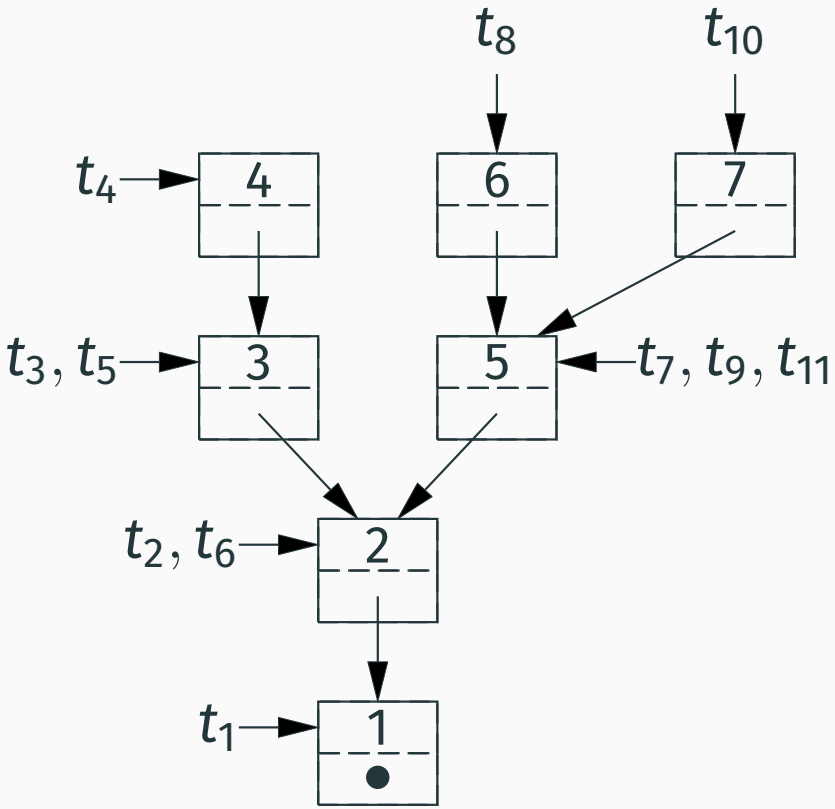
En OCaml, Haskell, etc : avec un type algébrique

```
type 'a stack = Empty | Stack of 'a * 'a stack
let empty = Empty
let push v s = Stack(v,s)
let top = function Stack(v,_) -> v | _ -> assert false
let pop = function Stack(_,s) -> s | _ -> assert false
```

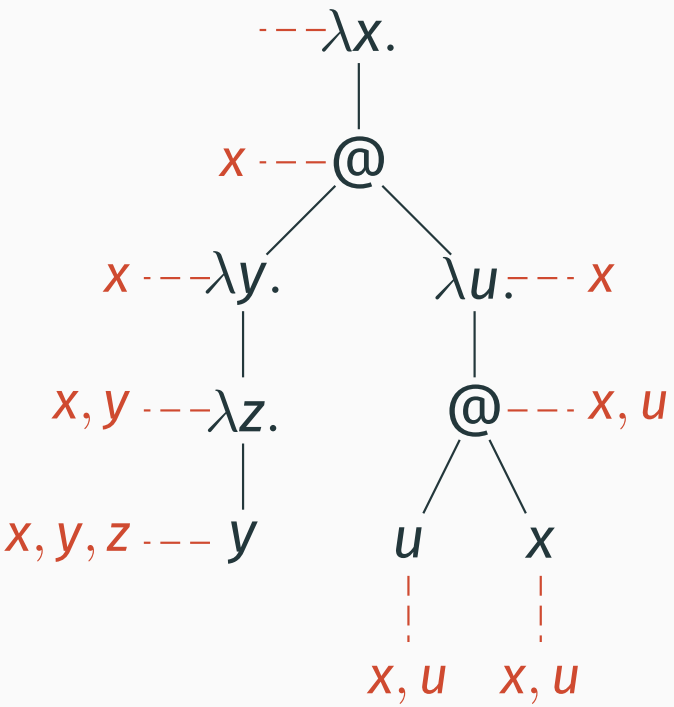
Piles persistantes et partage en mémoire

Une pile produite par `push` ou `pop` partage tous les blocs mémoire sauf un avec la pile précédente. Cela permet de conserver tous les états successifs de la pile de manière économe en mémoire : N blocs pour toute séquence de N `push` et M `pop`.

$$
\begin{aligned}
t_1 &= \mathtt{push}(1, \mathtt{empty}) \\
t_2 &= \mathtt{push}(2, t_1) \\
t_3 &= \mathtt{push}(3, t_2) \\
t_4 &= \mathtt{push}(4, t_3) \\
t_5 &= \mathtt{pop}(t_4) \\
t_6 &= \mathtt{pop}(t_5) \\
t_7 &= \mathtt{push}(5, t_6) \\
t_8 &= \mathtt{push}(6, t_7) \\
t_9 &= \mathtt{pop}(t_8) \\
t_{10} &= \mathtt{push}(7, t_9) \\
t_{11} &= \mathtt{pop}(t_{10})
\end{aligned}
$$



Application : annoter un ASA par des environnements



Annoter chaque nœud d'un arbre de syntaxe abstraite par son environnement, c.à.d. l'ensemble des variables libres en ce point.

Environnement ≈ pile
Entrée dans un lieur ≈ empiler
Fin de portée d'un lieur ≈ dépiler

	Tableaux	Listes	A.B.R. (→ 2e cours)
Partage	aucun	maximal	élevé
Espace total	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
Temps recherche	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$

Spécifications algébriques

En mathématiques, une structure algébrique est un ensemble muni d'opérations qui satisfont des identités (équations).

Exemple : un groupe est un ensemble G avec trois opérations : une constante 1, une opération binaire $\cdot$, une opération unaire $^{-1}$, satisfaisant les identités

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$
$$1 \cdot x = x = x \cdot 1$$
$$x \cdot x^{-1} = 1 = x^{-1} \cdot x$$

Spécifications algébriques

(Guttag and Horning, *The Algebraic Specification of Abstract Data Types*, 1978.)

En informatique, un type abstrait algébrique est un type abstrait (= nom de type + opérations) spécifié par des équations portant sur les opérations.

Exemple : les piles (opérations $\texttt{empty}$, $\texttt{push}$, $\texttt{pop}$, $\texttt{top}$)

$$\texttt{top}\,(\texttt{push}(v, s)) = v$$
$$\texttt{pop}\,(\texttt{push}(v, s)) = s$$

Si on ajoute l'opération $\texttt{enqueue}$ (insérer en bas de la pile) :

$$\texttt{enqueue}(v, \texttt{empty}) = \texttt{push}(v, \texttt{empty})$$
$$\texttt{enqueue}(v, \texttt{push}(v', s)) = \texttt{push}(v', \texttt{enqueue}(v, s))$$

Spécifications algébriques et persistance

$$\texttt{top}\ (\texttt{push}(v, s)) = v$$

$$\texttt{pop}\ (\texttt{push}(v, s)) = s$$

Ce style équationnel de spécification suppose que le type abstrait a une interface persistante : les opérations `push`, `pop` produisent de nouvelles piles, elles ne modifient pas (de manière observable) les piles existantes.

Spécifications algébriques et persistance

$$\texttt{top}\ (\texttt{push}(v, s)) = v$$

$$\texttt{pop}\ (\texttt{push}(v, s)) = s$$

Pour une structure éphémère (comme la pile à base de tableau), on n'a plus d'équations, plutôt des équivalences de programmes

$$\begin{aligned} \texttt{push}(v);\ \texttt{pop}() &\approx \texttt{skip} \\ \texttt{push}(v);\ x := \texttt{top}() &\approx x := v;\ \texttt{push}(v) \end{aligned}$$

plus des règles de commutation avec les commandes qui ne dépendent pas de l'état de la pile.

Spécifications algébriques et implémentations fonctionnelles

$$\texttt{top}\ (\texttt{push}(v, s)) = v$$
$$\texttt{pop}\ (\texttt{push}(v, s)) = s$$
$$\texttt{enqueue}(v, \texttt{empty}) = \texttt{push}(v, \texttt{empty})$$
$$\texttt{enqueue}(v, \texttt{push}(v', s)) = \texttt{push}(v', \texttt{enqueue}(v, s))$$

Il est facile de vérifier qu'une implémentation fonctionnelle pure vérifie ces équations. Pour l'implémentation en OCaml, une fois expansées les définitions, il reste à montrer

```
(match Stack(v,s) with Stack(v,_) -> v | _ -> assert false) = v
(match Stack(v,s) with Stack(_,s) -> s | _ -> assert false) = s
```

ce qui découle de la sémantique opérationnelle du filtrage `match...with`.

Spécifications algébriques et implémentations fonctionnelles

$$\texttt{top}\ (\texttt{push}(v, s)) = v$$
$$\texttt{pop}\ (\texttt{push}(v, s)) = s$$
$$\texttt{enqueue}(v, \texttt{empty}) = \texttt{push}(v, \texttt{empty})$$
$$\texttt{enqueue}(v, \texttt{push}(v', s)) = \texttt{push}(v', \texttt{enqueue}(v, s))$$

Symétriquement, on peut souvent dériver systématiquement une implémentation fonctionnelle pure à partir des équations.

P.ex. pour `enqueue` :

```
let rec enqueue v s =
  match s with
  | Empty -> Stack(v, Empty)
  | Stack(v', s) -> Stack(v', enqueue v s)
```

L’émergence des structures de données persistantes

Structure de donnée = données + relations structurelles

Computer programs usually operate on tables of information. In most cases these tables are not simply amorphous masses of numerical values; they involve important structural relationships between the data elements.

In order to use a computer properly, we need to understand the structural relationships present within data, as well as the basic techniques for representing and manipulating such structure within a computer.

D. E. Knuth, The Art of Computer Programming, vol 1, chap 2, "Information structures", 1968.

1945–1960 : La préhistoire des structures de données

Données stockées dans des tableaux, soit comme des ≪masses amorphes de valeurs numériques≫, soit avec un peu de structure :

- table triée + recherche dichotomique ; (Mauchly, 1946)
- table de hachage ; (A. Dumey et al, 1956–)
- ≪pointeurs≫ d'un tableau dans un autre
  (dans les premières bases de données ou de connaissances).

1960–1970 : émergence du concept, premiers grands progrès

Une structure de donnée = une interface (jeu d'opérations),
avec plusieurs implémentations possibles.

- Piles, files d'attente.
- Dictionnaires, avec des implémentations à base d'arbres :
  sans équilibrage, (Windley et al, 1960 ; bien d'autres)
  auto-équilibrés. (Adelson-Velskii et Landis, 1962 ; bien d'autres)
- Files de priorité : tas (*heaps*) (Williams, 1964)

Un programme =

des algorithmes abstraits

\+ des structures de données efficaces adaptées

(pouvant elles-même mettre en œuvre des algorithmes subtils)

Exemple d'algorithme abstrait : le plus court chemin de Dijkstra

> *In the course of the solution the nodes are subdivided into three sets [...] Consider all branches connecting the node just transferred to set A with nodes R in sets B or C [...] the node with minimum distance from P is transferred from set B to set A [...]*
>
> *(E.W. Dijkstra, A note on two problems in connexion with graphs, 1959.)*

L'implémentation concrète des ensembles est laissée au lecteur, ainsi que le moyen efficace de trouver ≪le nœud à distance minimale de *P*≫ dans l'ensemble *B*.

Il faudra attendre 1962 et l'invention de la structure de tas (*heap*) par Williams pour avoir ce moyen.

Depuis 1970 : les temps modernes des structures de données

Exploration systématique tirée par les nouveaux besoins et les nouvelles approches en algorithmique :

- structures multidimensionnelles : algorithmes géométriques, bases de données, …
- chaînes et recherche de motifs ;
- structures sans verrouillage (*lock-free*) pour le parallélisme ;
- structures probabilistes (≪randomisées≫).

Énormes progrès de l'analyse d'algorithmes : cas le pire, analyse en moyenne, analyse amortie, analyse du temps attendu, …

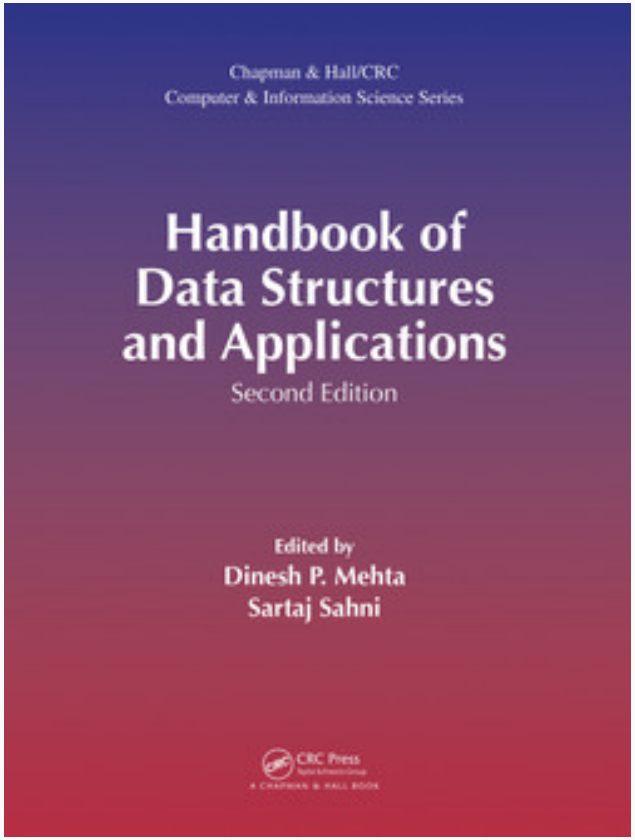
(Voir aussi : les cours de B. Chazelle, J.-D. Boissonnat, C. Mathieu, R. Guerraoui et F. Magniez sur la chaire annuelle d'informatique et de sciences numériques.)

Structures éphémères et programmation impérative

Tout comme la plupart des algorithmes sont de type ≪recette de cuisine≫ et la plupart des programmes sont écrits en style impératif, la plupart des structures de données sont éphémères :

> *Les opérations sur la structure (p.ex. insérer dans un dictionnaire) peuvent modifier ≪en place≫ l'état de la structure, rendant inaccessible l'état de la structure avant l'opération.*

Handbook of Data Structures and Applications (CRC, 2018)



Sur 64 chapitres, seulement 2 parlent de structures de données non éphémères :

31. *Persistent data structures*
(Haim Kaplan)

40. *Functional data structures*
(Chris Okasaki)

Structures persistantes

> *Les opérations préservent l'état courant de la structure. Si la structure doit être modifiée, une «nouvelle» structure est produite.*
> *Autrement dit : les opérations se présentent comme des fonctions pures, sans effets de bord.*

Note : l'implémentation peut utiliser des traits impératifs (p.ex. mutations sur des tableaux).

Apparues dans les années 1980, dans le contexte de l'algorithmique graphique, où il est parfois utile de disposer de l'historique complet de la structure.
(Souvent appelé *searching in the past* ou *in-the-past queries*.)

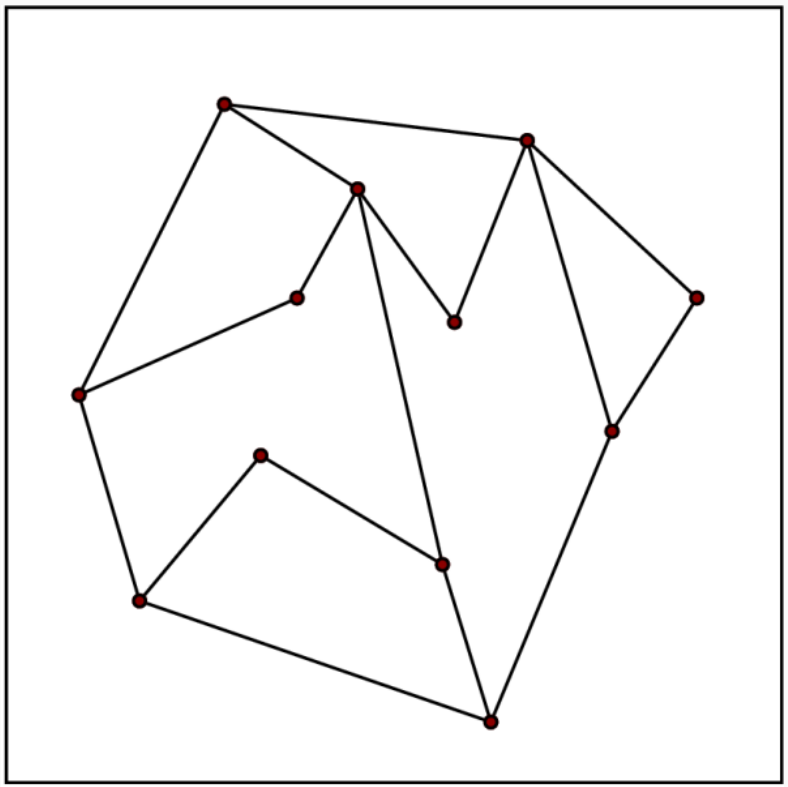
Structures purement fonctionnelles

Structures persistantes dont l'implémentation n'utilise aucun trait impératif (pas de mutations) et peut s'écrire dans un langage fonctionnel pur.

Apparues dans les années 1990 pour permettre une algorithmique efficace en programmation fonctionnelle pure.

Exemple avancé : localisation de points (*planar point location*)

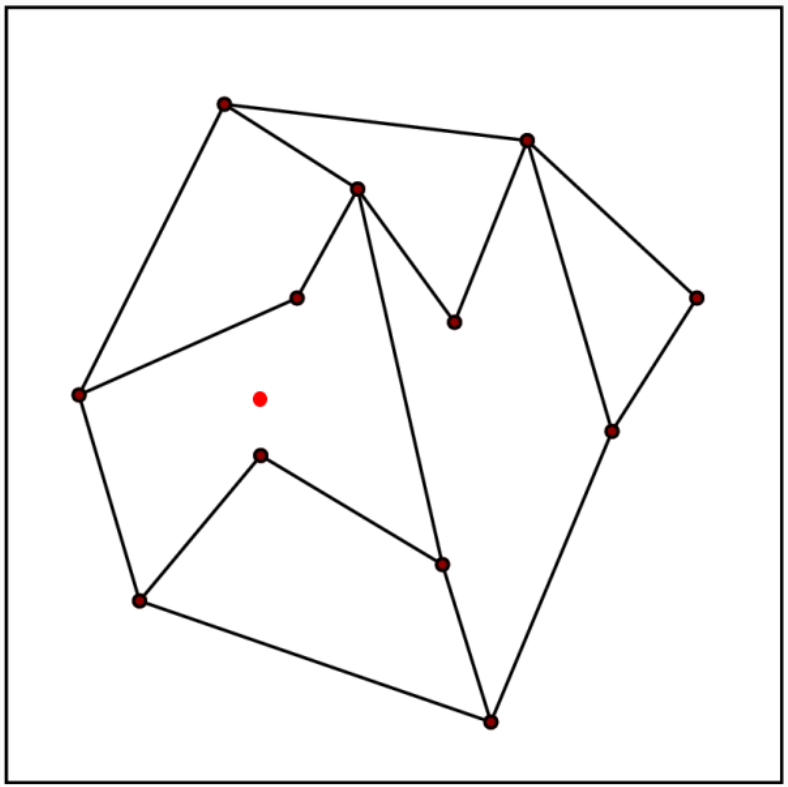
Le problème de localisation de points dans le plan (*planar point location*)



(Gfonsecabr, English Wikipedia)

Étant donnés des segments de droites définissant des polygones, et des points $P_1, \ldots, P_k$, trouver rapidement à quel polygone appartient chaque point.

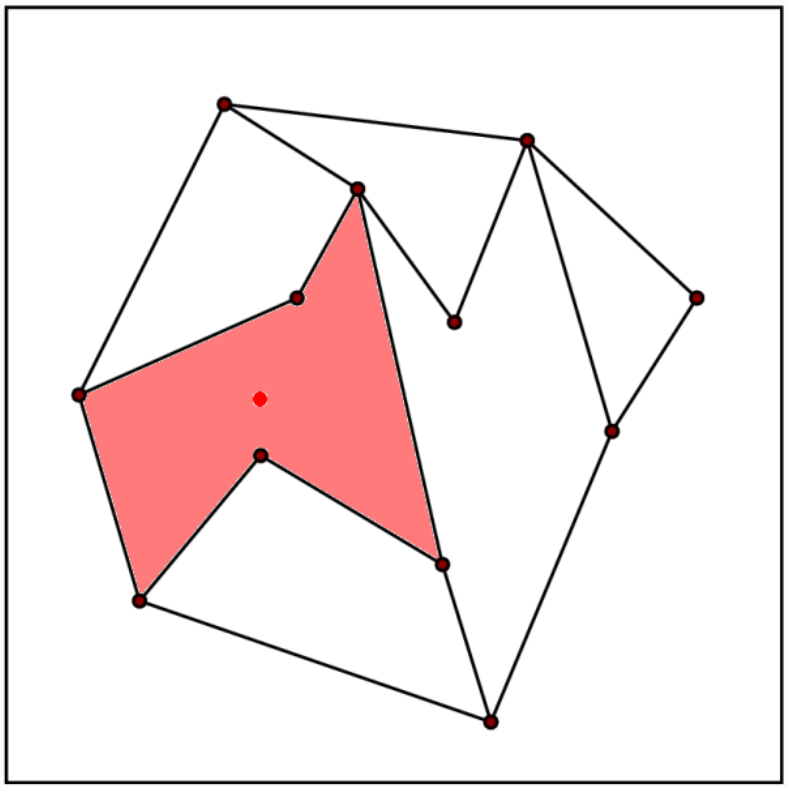
Le problème de localisation de points dans le plan (*planar point location*)



(Gfonsecabr, English Wikipedia)

Étant donnés des segments de droites définissant des polygones, et des points $P_1, \ldots, P_k$, trouver rapidement à quel polygone appartient chaque point.

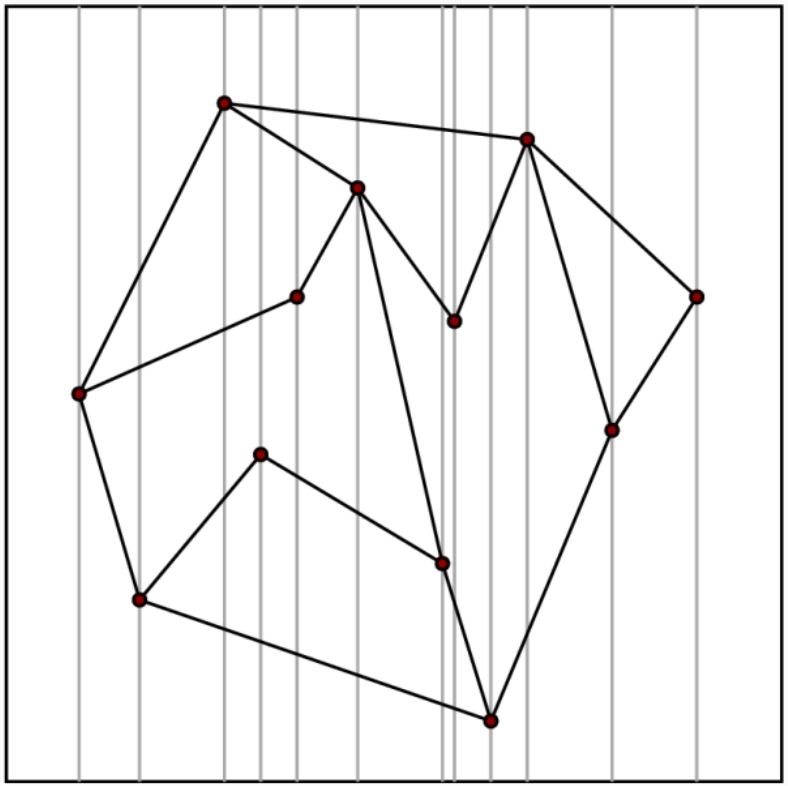
Le problème de localisation de points dans le plan (*planar point location*)



(Gfonsecabr, English Wikipedia)

Étant donnés des segments de droites définissant des polygones, et des points $P_1, \ldots, P_k$, trouver rapidement à quel polygone appartient chaque point.

Découpage en tranches (*slabs*) (Dobkin et Lipton, 1976)



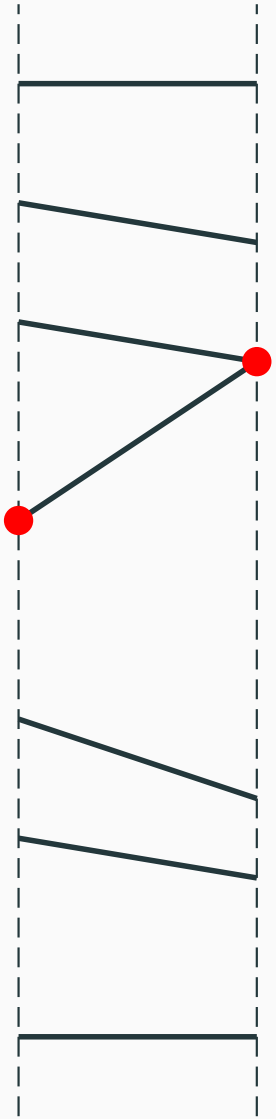
On trie les coordonnées x des extrémités des segments.

Cela découpe le plan en tranches verticales (les *slabs*).

À l'intérieur d'une tranche, le nombre de segments est constant.

Les segments ≪entrent≫ et ≪sortent≫ aux frontières entre deux tranches.

Recherche dichotomique à l'intérieur d'une tranche



À l'intérieur d'une tranche, les segments ne se croisent pas.
On peut donc les trier par position verticale, du plus bas au plus haut.

Étant donné un point P, une recherche dichotomique détermine les deux segments S_i, S_j immédiatement ≪au dessus≫ et ≪en dessous≫.

Cela suffit à identifier le polygone qui contient P.

Algorithme de Dobkin et Lipton

Pré-traitement des n segments :

1. Trier les x des extrémités des segments $\rightarrow \mathcal{O}(n)$ tranches
2. Pour chaque tranche, construire un tableau des $\mathcal{O}(n)$ segments dans la tranche, et les trier par position verticale.

Espace : $\mathcal{O}(n^2)$.

Pour chaque point $P = (x, y)$:

1. Trouver la tranche qui le contient par recherche dichotomique sur x.
2. Trouver les deux segments qui l'encadrent par recherche dichotomique dans la tranche.

Temps : $\mathcal{O}(\log n)$.

Réduire l'espace et le temps de prétraitement

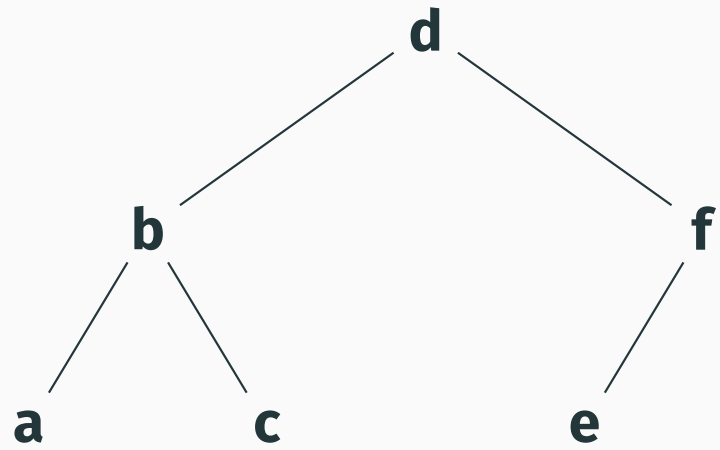
Chacun des n segments «entre» et «sort» d'une tranche exactement une fois.

Deux tranches successives partagent donc la plupart de leurs segments, avec les mêmes positions verticales relatives.

Idée : représenter chaque tranche non plus par un tableau trié, mais par une structure persistante avec recherche en $\mathcal{O}(\log n)$ mais qui permet le partage entre tranches.

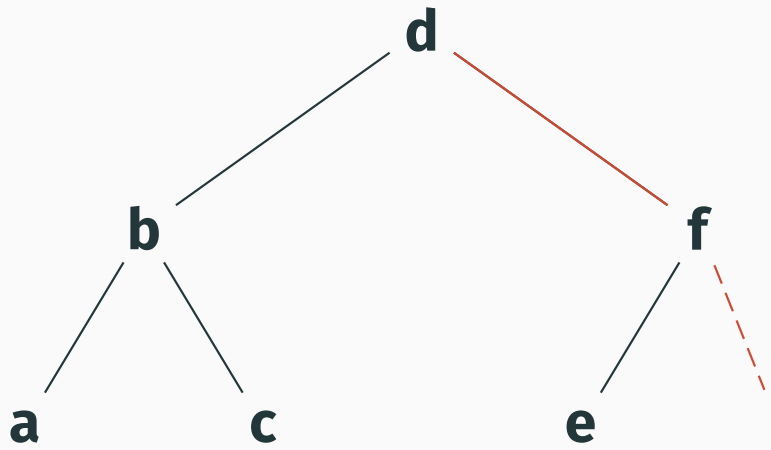
Une structure appropriée : un arbre binaire de recherche équilibré persistant (AVL, rouge-noir, etc) ($\rightarrow$ 2e cours)

Insertion persistante dans un A.B.R.



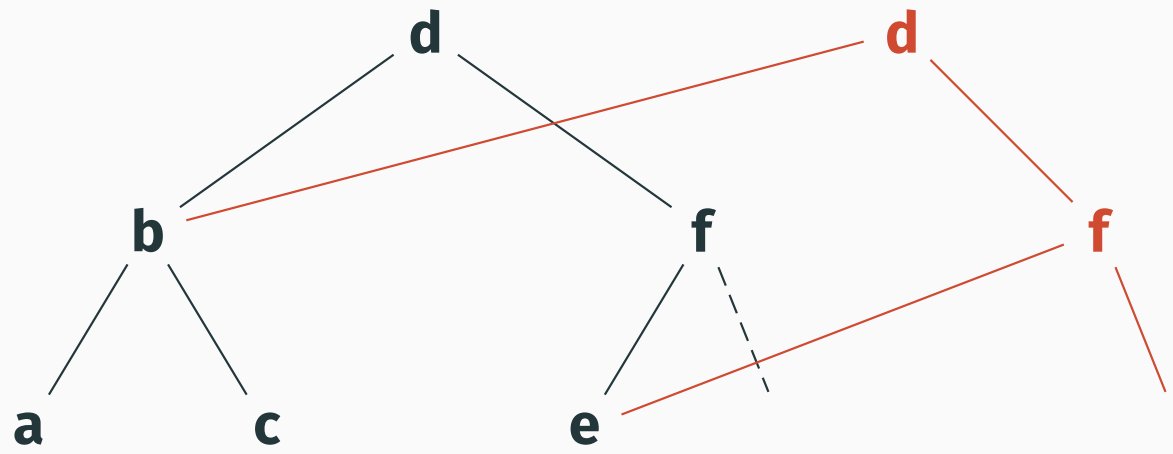
1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.

Insertion persistante dans un A.B.R.



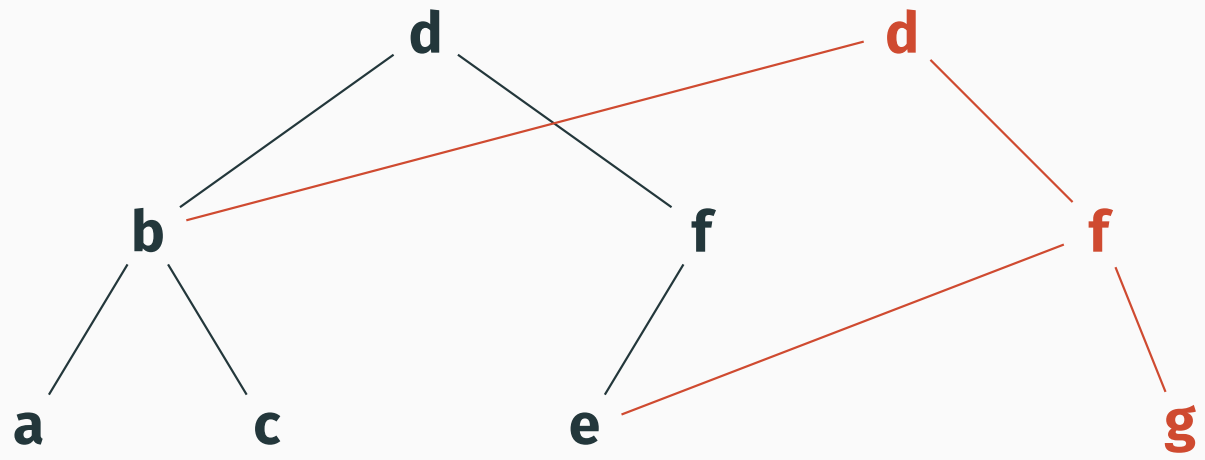
1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.

Insertion persistante dans un A.B.R.



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 persistante dans un A.B.R.



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. A la fin du chemin copié, ajouter le nœud $\langle \bullet, \mathbf{g}, \bullet \rangle$.
4. Si nécessaire : rééquilibrer l'arbre, en préservant le partage.

Temps : $\mathcal{O}(\log n)$, espace : $\mathcal{O}(\log n)$.

Algorithme de localisation amélioré

Pré-traitement des n segments :

1. Balayer les extrémités des segments par x croissant.
2. Les faire ≪entrer≫ et ≪sortir≫ dans un ABR persistant équilibré, trié par position verticale croissante.
3. Garder les états intermédiaires de l'ABR (= les tranches) dans un tableau.

Temps et espace : $\mathcal{O}(n \log n)$.

(Chacun des n segments ≪entre≫ et ≪sort≫ une fois de l'ABR, temps $\mathcal{O}(\log n)$.)

Pour chaque point $P = (x, y)$: recherche dichotomique dans le tableau des tranches, puis dans l'ABR correspondant à la tranche.
Temps : $\mathcal{O}(\log n)$.

L'algorithme de Sarnak et Tarjan

Sarnak et Tarjan (*Planar Point Location using Persistent Search Trees*, CACM, 1986) montrent comment réduire l'espace de prétraitement de $\mathcal{O}(n \log n)$ à $\mathcal{O}(n)$ en utilisant une autre implémentation des A.B.R. persistants.

Cette implémentation utilise de la mutation en place et s'appuie sur la technique des *fat nodes* de Driscoll, Sarnak, Sleator, et Tarjan (1989).

C'est une technique générale pour transformer une structure éphémère en structure persistante : remplacer chaque champ de chaque nœud par un historique des modifications de ce champ, i.e. un ensemble de paires (date de modification, nouvelle valeur).

$\rightarrow 4^{e}$ cours

Exemple d'ABR avec des *fat nodes*

ABR éphémère :

t_0

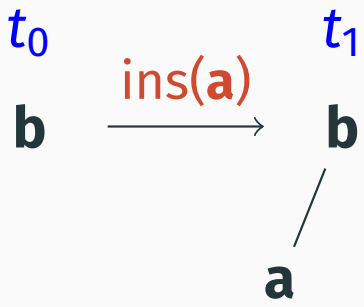
b

ABR persistant :

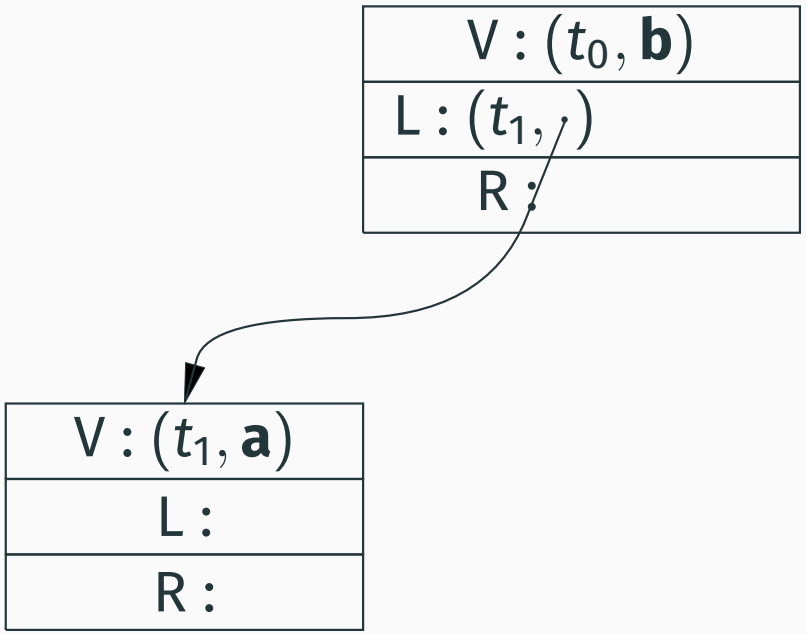
V : $(t_0, \mathbf{b})$
L :
R :

Exemple d'ABR avec des *fat nodes*

ABR éphémère :

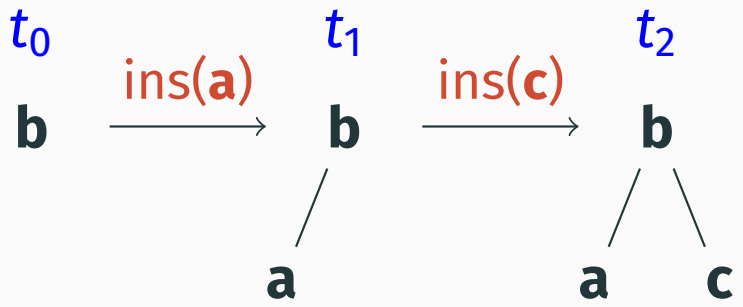


ABR persistant :

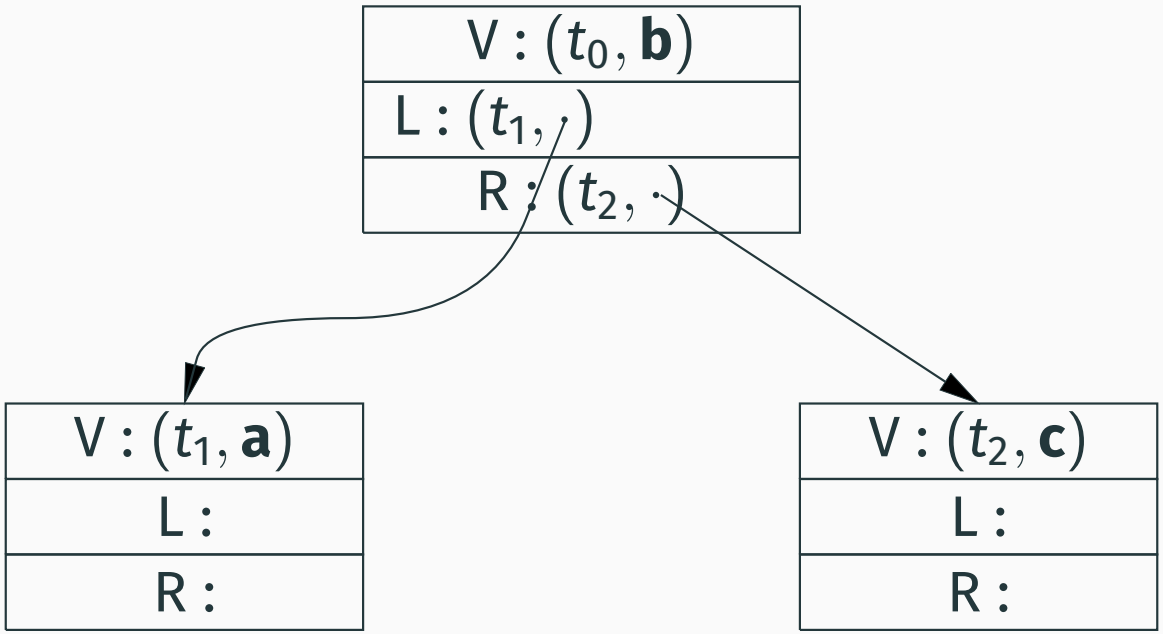


Exemple d'ABR avec des *fat nodes*

ABR éphémère :

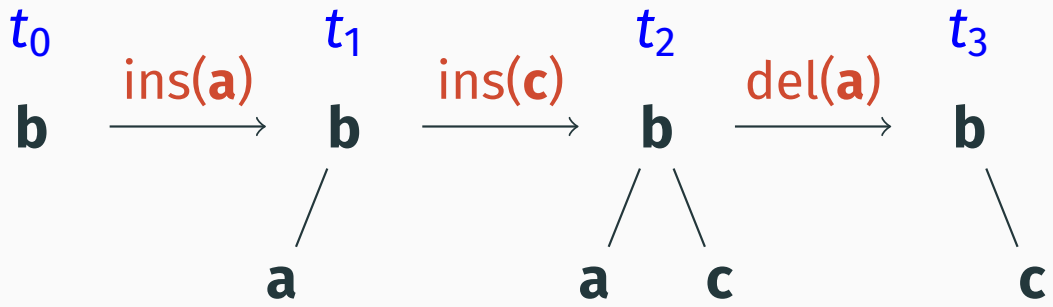


ABR persistant :

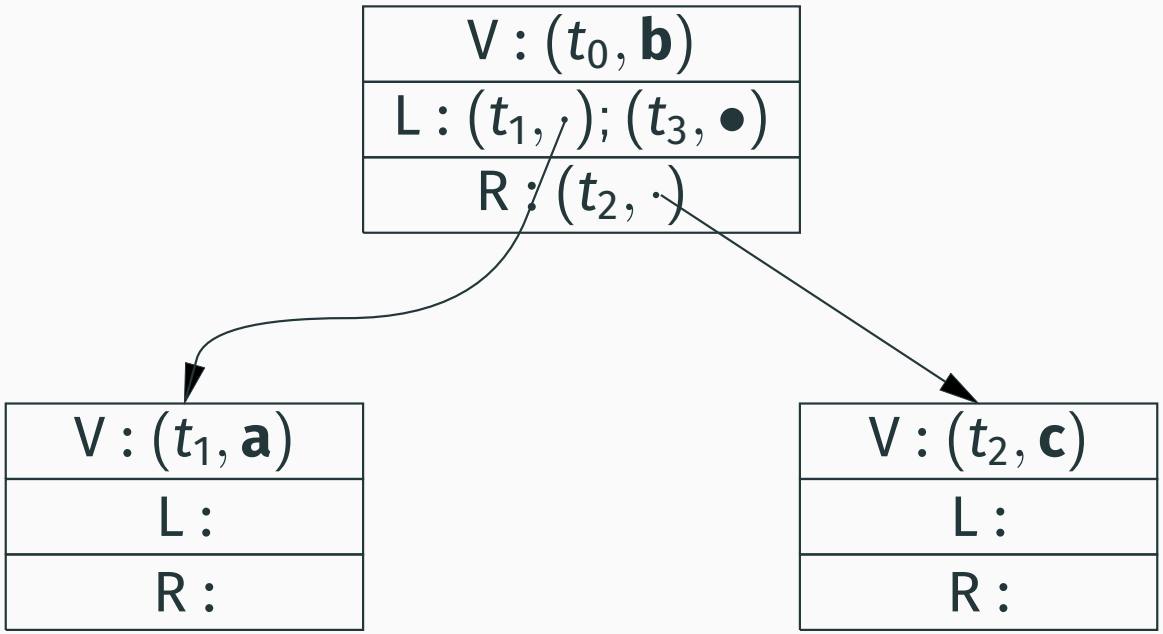


Exemple d'ABR avec des *fat nodes*

ABR éphémère :



ABR persistant :



Taille mémoire utilisée

Chaque modification d'un champ coûte $O(1)$ en espace (pour ajouter une entrée à l'historique correspondant).

Après c création de nœuds et m modifications de champ, la structure est donc de taille $\mathcal{O}(c + m)$.

Pour une insertion «en place» dans un A.B.R. équilibré, on a :

- une création de nœud et une MAJ de champ du parent ;
- plus un petit nombre de MAJ pour le rééquilibrage : $\mathcal{O}(1)$ amorti. ($\rightarrow$ 3e cours)

Même analyse pour la suppression.

Après n insertions et n suppressions, on a donc une structure de taille $\mathcal{O}(n)$ au lieu de $\mathcal{O}(n \log n)$ dans l'approche précédente.

Temps d'accès dans la structure

Dans un *fat node*, accéder à la valeur d'un champ à la date t est en temps $\mathcal{O}(\log m)$ où m est le nombre de modifications du champ.

Sans autres précautions, la recherche dans l'A.B.R. persistant prend donc un temps $\mathcal{O}(\log^2 n)$.

Sarnak et Tarjan montrent comment redescendre à $\mathcal{O}(\log n)$ par une combinaison de MAJ en place de *fat nodes* et de copie de nœuds : il suffit de borner la taille des historiques, et de copier le *fat node* lorsque l'historique déborde. ($\rightarrow$ 4e cours)

Le résultat est une structure de données optimale pour le problème *planar point location* : taille $\mathcal{O}(n)$ et temps d'interrogation $\mathcal{O}(\log n)$.

L’émergence de la programmation fonctionnelle pure

Lambda-calcul et fonctions récursives générales

Lambda-calcul pur : (Church, 1935)

$$M, N ::= x \mid \lambda x.\, M \mid M\ N$$

Tout est codé en termes de fonctions : les structures de données (entiers, booléens, listes, …) et les structures de contrôle (conditionnelle, opérateurs de point fixe pour la récursion).

Fonctions récursives générales : (Kleene, 1936)

Un seul type de données (tuples d'entiers) + des opérateurs pour construire des fonctions $\mathbb{N}^p \to \mathbb{N}^q$ (succ, pred, projections, composition, récursion primitive, minimisation).

Initialement étudiés comme formalismes de calculabilité (équivalents aux machines de Turing) et non comme langages de programmation.

Lisp (J. McCarthy et al, 1960)

Initialement : une bibliothèque FORTRAN pour le calcul symbolique, manipulant des S-expressions :

$$\begin{aligned} \mathit{sexp} &::= \mathit{atome} \mid (\mathit{sexp}\ .\ \mathit{sexp}) \\ \mathit{atome} &::= \mathit{nombre} \mid \mathit{symbole} \mid \texttt{nil} \end{aligned}$$

Évolue rapidement vers un langage applicatif pour définir des fonctions récursives opérant sur les S-expressions, et représentées elles-même par des S-expressions.

Lisp : le premier langage applicatif

```
(define mapcar (fun lst)
  (if (null lst)
      nil
      (cons (fun (car lst)) (mapcar fun (cdr lst)))))
```

Pas de distinction expression/commande ; tout est expression.

Exécuter une expression = calculer sa valeur.

Récursion de préférence à itération.

Pas (peu) de modifications en place : `cons` renvoie toujours une nouvelle cellule de liste ; récupération mémoire par GC.

Le début d'une longue lignée de langages, dont Common Lisp, Scheme, Racket, Clojure, …

Quelle programmation et quelle algorithmique en Lisp classique ?

Naturellement : une programmation fonctionnelle pure utilisant les listes comme principale structure de données.

- Lisibilité, réutilisabilité du code.
- Complexité non optimale (listes $\rightarrow \mathcal{O}(n)$).
- Mais n souvent petit, surtout dans les années 1960 !
- Algorithmes de calcul symbolique coûteux de toute façon.

Comme concession à l'efficacité : quelques traits impératifs.

- `setq` pour changer la valeur d'un symbole (affectation) ;
- `rplaca`, `rplacd` pour modifier une cellule de liste en place ;
- tableaux (impératifs).

ML et descendants (Milner et al, 1978–)

Initialement : Lisp + typage statique et abstraction de types

```
absrectype * tree = * + * tree # * tree
  with leaf n = abstree(inl n)
   and node (t1, t2) = abstree(inr(t1, t2))
   and isleaf t = isl(reptree t)
   and leafval t = outl(reptree t) ? failwith ‘leafval‘
   and leftchild t = fst(outr(reptree t) ? failwith ‘leftchild‘
   and rightchild t = snd(outr(reptree t) ? failwith ‘leftchild‘
```

Puis étendu avec des types algébriques et du filtrage de motifs (HOPE, Prolog).

```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
let rec sum = function Leaf n -> n | Node(l, r) -> sum l + sum r
```

Riche descendance : Standard ML, Caml, OCaml, F#, etc.

Traits impératifs en ML

Même compromis qu'en Lisp : programmation naturellement fonctionnelle pure + de l'impératif si besoin.

> *At the same time, imperative features were important for practical reasons; no-one had experience of large useful programs written in a pure functional style. In particular, an exception-raising mechanism was highly desirable for the natural presentation of tactics.*
>
> *(Milner et al, The Definition of Standard ML)*

Non seulement des exceptions, mais aussi de l'état mutable sous forme de références (cellules d'indirection modifiables en place) et de tableaux.

```
let x = ref 0 in x := !x + 2
```

Haskell $\approx$ ML + évaluation paresseuse + bien d'autres nouveautés.

Une synthèse de travaux des années 1980 sur
l'évaluation paresseuse (à la demande) des calculs,
par opposition à l'évaluation stricte de Lisp et ML :

- Pour pouvoir définir davantage de «bouts de calculs» sous forme de fonctions. P.ex. la fonction `ifthenelse` :

```
ifthenelse True  a b = a
ifthenelse False a b = b
```

- Pour faciliter la manipulation de structures de données infinies, p.ex les flux (*streams*) (= listes infinies).

Haskell : le triomphe de la programmation fonctionnelle pure ?

> *Lazyness prevented us from sinning.*
>
> *(attribué à S. Peyton Jones)*

L'évaluation paresseuse (non stricte) rend quasi-impossible de savoir quand une expression est évaluée, rendant inutilisables les effets de bords (E/S, affectations).

D'où un retour aux sources de Lisp 1960 :

- Programmation purement fonctionnelle.
- Raisonnement équationnel.
- Dérivation de programmes par *calculation*.

Haskell : le retour de la programmation impérative

> *In short, Haskell is the world's finest imperative programming language*
>
> *(S. Peyton Jones, 2000)*

Besoins irrésistibles : entrées/sorties, interfaces autres langages, tableaux mutables (calcul numérique), références (algorithmes d'unification en place), …

$\rightarrow$ Retour des traits impératifs sous des formes contrôlées compatibles avec l'évaluation non stricte :

- Haskell : les monades
- Clean, Linear Haskell : les types linéaires.

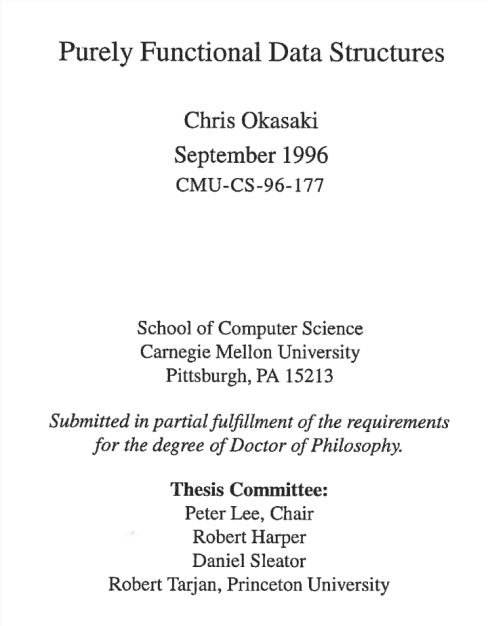
Émergence d'une algorithmique purement fonctionnelle

Dans les années 1990 s'impose (enfin) l'idée qu'un programme purement fonctionnel peut être algorithmiquement efficace à condition d'utiliser des structures de données (fonctionnelles ou persistantes) adaptées.

Premiers ≪clients≫ : les assistants à la démonstration et autres outils de vérification,

- pour plus de garanties de correction (pas d'interférences);
- pour économiser la mémoire (partage des données).

Les travaux de thèse de Chris Okasaki



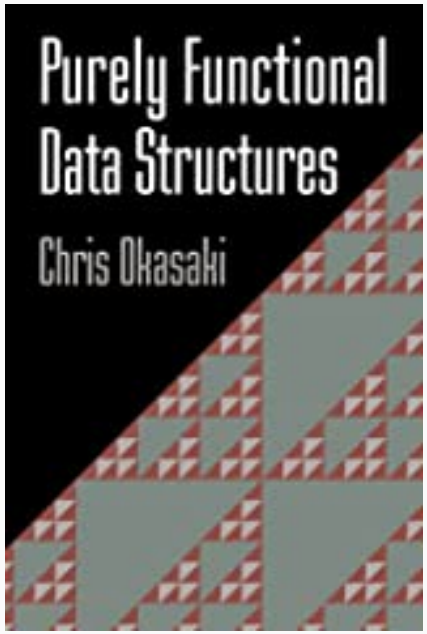
Purely Functional Data Structures

Chris Okasaki
September 1996
CMU-CS-96-177

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Peter Lee, Chair
Robert Harper
Daniel Sleator
Robert Tarjan, Princeton University



Reformulation systématique de structures et d'algorithmes avancés en style fonctionnel pur.

Nouvelles utilisations de l'évaluation paresseuse et nouvelles techniques d'analyse amortie correspondantes.

Le *Journal of Functional Programming*, 1993-2008

FUNCTIONAL PEARLS

Efficient sets—a balancing act

STEPHEN ADAMS
Electronics and Computer Science Department, University of Southampton, UK

Simple and efficient purely functional queues and deques

CHRIS OKASAKI
School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA
(e-mail: cokasaki@cs.cmu.edu*)*

FUNCTIONAL PEARL

The Zipper

GÉRARD HUET
INRIA Rocquencourt, France

FUNCTIONAL PEARL

Three algorithms on Braun trees

CHRIS OKASAKI*
School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA
(*e-mail:* cokasaki@cs.cmu.edu)

FUNCTIONAL PEARLS

Diets for fat sets

MARTIN ERWIG
FernUniversität Hagen, Praktische Informatik IV. 58084 Hagen, Germany
(*e-mail:* erwig@fernuni-hagen.de)

FUNCTIONAL PEARL

Explaining binomial heaps

RALF HINZE
Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* ralf@informatik.uni-bonn.de)

FUNCTIONAL PEARL

Red-black trees in a functional setting

CHRIS OKASAKI*
School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213, USA

Red-black trees with types

STEFAN KAHRS
University of Kent at Canterbury, Canterbury, Kent, UK

FUNCTIONAL PEARL

A fresh look at binary search trees

RALF HINZE
Institute of Information and Computing Sciences, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
(*e-mail:* ralf@cs.uu.nl)

Finger trees: a simple general-purpose data structure

RALF HINZE
Institut für Informatik III, Universität Bonn, Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* ralf@informatik.uni-bonn.de)

ROSS PATERSON
Department of Computing, City University, London EC1V OHB, UK
(*e-mail:* ross@soi.city.ac.uk)

Plan du cours

La suite du cours

16 mars	Arbres équilibrés + copie de branches = dictionnaires persistants
23 mars	Concilier amortissement et persistance : de l'importance de la paresse
30 mars	Comment rendre persistante une structure impérative ?
6 avril	Systèmes de numération, types non réguliers, et *bootstrapping* structurel
13 avril	De la dérivation formelle à la navigation dans une structure : contextes, *zippers*, index.
20 avril	À la recherche du vecteur perdu : limites théoriques et conclusions.

Le séminaire

23 mars Tobias Nipkow (T.U. München)
Verification of functional data structures : Correctness and complexity

30 mars Jean-Christophe Filliâtre (CNRS)
Structures de données semi-persistantes

6 avril KC Sivaramakrishnan (IIT Madras et Tarides)
Mergeable replicated data types

13 avril Arthur Charguéraud (Inria)
Transience : comment allier persistance et performance

20 avril Pierre-Etienne Meunier (Coturnix)
Une algèbre de modifications, ou : le contrôle de versions pour tous.

Bibliographie

Bibliographie

Deux références utiles pour tout le cours :

- Chris Okasaki, *Purely Functional Data Structures*,
  Cambridge University Press, 1998, 2009.

- Haim Kaplan, *Persistent Data Structures*,
  chap. 31 du *Handbook of data structures and applications*,
  Chapman&Hall / CRC Press, 2005,
  `http://www.cs.tau.ac.il/~haimk/papers/persistent-survey.ps`