

Dictionnaires randomisés à représentation unique

Implémentation en OCaml et utilisation dans l'analyseur statique Salto

Hadrien Ferlin (ENS Rennes)

Encadrant : Benoît Montagu (Inria)

Stage de recherche de L3 (avril–juillet 2026) - équipe Epicure (Inria, Irisa)

24 septembre 2026



Contexte : analyser un programme sans jamais l'exécuter

Salto : analyseur statique pour OCaml, par interprétation abstraite

→ calcule une **sur-approximation** de tous les états atteignables.

Le prix à payer : le temps et la mémoire.

Analyser `boyer.ml` aujourd'hui : **5 heures** et **2,3 Gio** de pic mémoire.

Contexte : analyser un programme sans jamais l'exécuter

Salto : analyseur statique pour OCaml, par interprétation abstraite

→ calcule une **sur-approximation** de tous les états atteignables.

Le prix à payer : le temps et la mémoire.

Analyser `boyer.ml` aujourd'hui : **5 heures** et **2,3 Gio** de pic mémoire.

Deux techniques classiques pour maîtriser ces coûts :

- **Hash-consing** : un seul exemplaire de chaque valeur en mémoire
- **Mémoïsation** : mettre en cache les résultats des fonctions

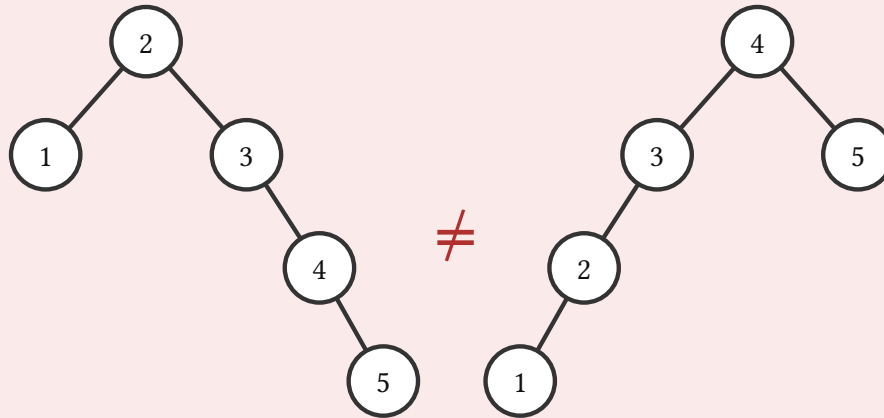
Les deux exigent que deux valeurs **égales** soient **la même valeur en mémoire**.

Le problème : les Maps OCaml n'ont pas de forme canonique

Le tas du programme analysé est stocké dans des **dictionnaires** (Map OCaml).

× Map OCaml : 2 formes pour le même dictionnaire

la forme dépend de l'ordre des insertions



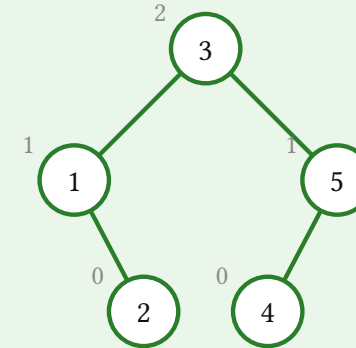
ajoutées dans l'ordre 1,2,3,4,5

ajoutées dans l'ordre 5,4,3,2,1

aucun sous-arbre partageable

✓ Zip-Tree : 1 seule forme

la forme ne dépend que des clés



rang = f (hash de la clé)

quel que soit l'ordre des ajouts

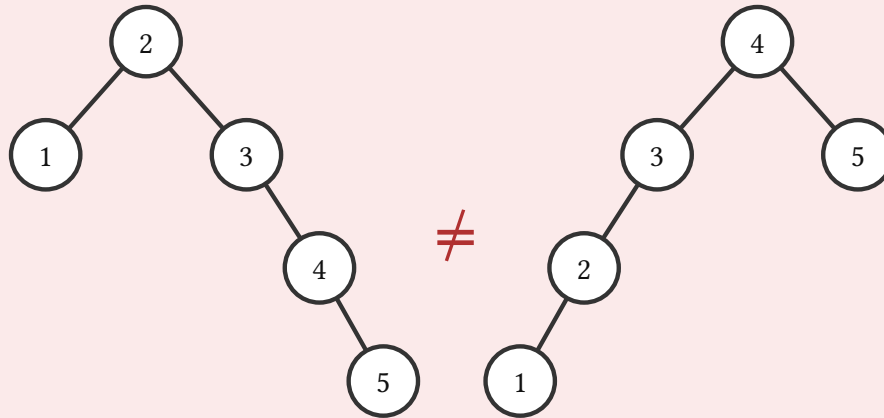
partage maximal possible

Le problème : les Maps OCaml n'ont pas de forme canonique

Le tas du programme analysé est stocké dans des **dictionnaires** (Map OCaml).

× Map OCaml : 2 formes pour le même dictionnaire

la forme dépend de l'ordre des insertions



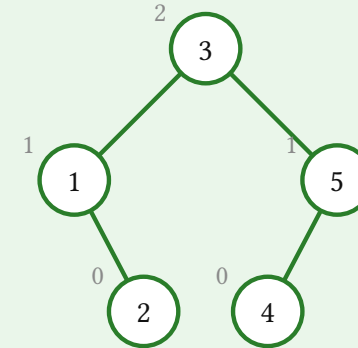
ajoutées dans l'ordre 1,2,3,4,5

ajoutées dans l'ordre 5,4,3,2,1

aucun sous-arbre partageable

✓ Zip-Tree : 1 seule forme

la forme ne dépend que des clés



rang = f (hash de la clé)

quel que soit l'ordre des ajouts

partage maximal possible

Question du stage : quelle structure de dictionnaire persistant à représentation unique est la plus performante.

Zip-Trees : le rang est dérivé du hash de la clé

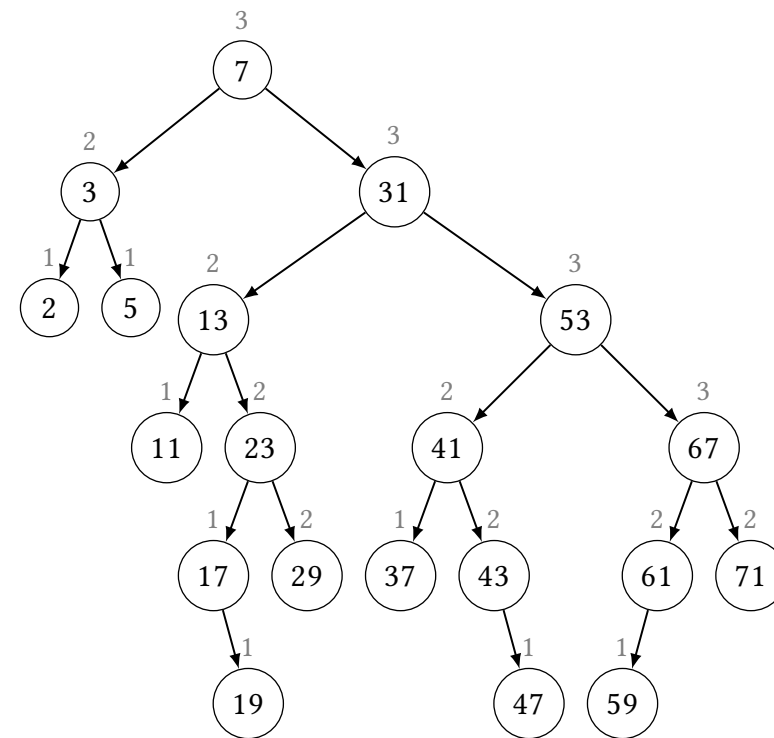
Chaque nœud porte une **clé** et un **rang** calculé de façon déterministe à partir du **hash de la clé**.

L'arbre est à la fois :

1. un arbre binaire de **recherche** selon les clés ;
2. un **tas max** selon les rangs.

Zip Tarjan+ 21	rang géométrique : zéros de poids faible du hash
Uniform Zip Gila+ 23	rang = hash directement (loi uniforme)

Profondeur moyenne : $\leq 1.5 \log_2 n$, et $1.39 \log_2 n$ pour les Uniform.



Zip-Trees : le rang est dérivé du hash de la clé

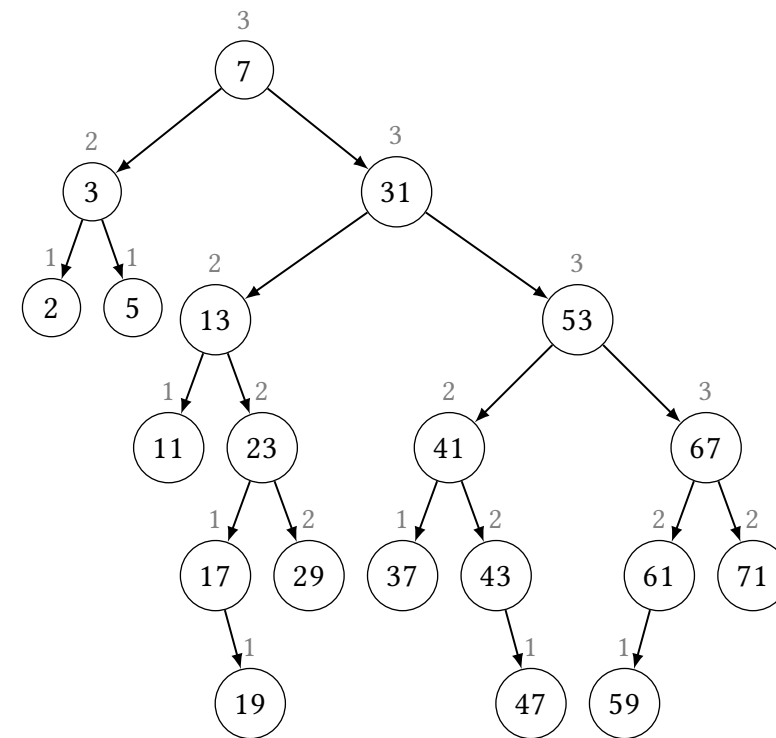
Chaque nœud porte une **clé** et un **rang** calculé de façon déterministe à partir du **hash de la clé**.

L'arbre est à la fois :

1. un arbre binaire de **recherche** selon les clés ;
2. un **tas max** selon les rangs.

Zip Tarjan+ 21	rang géométrique : zéros de poids faible du hash
Uniform Zip Gila+ 23	rang = hash directement (loi uniforme)

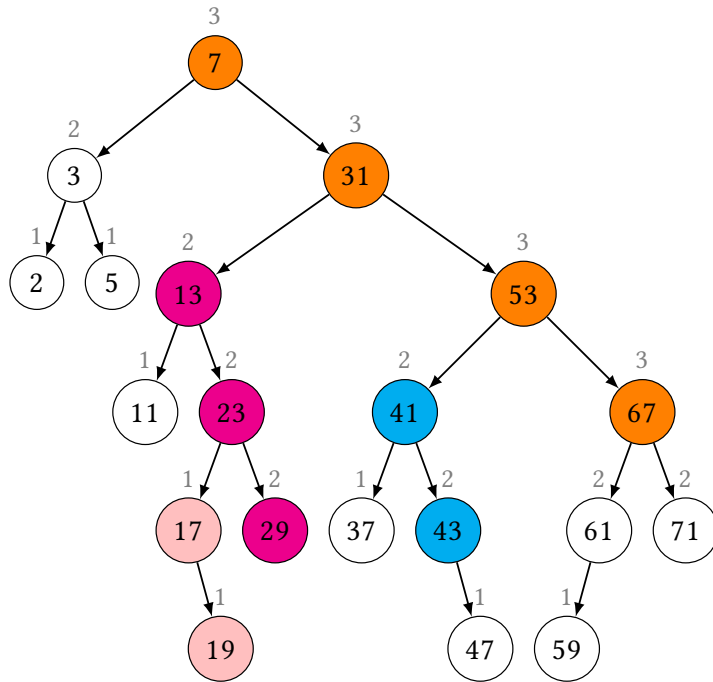
Profondeur moyenne : $\leq 1.5 \log_2 n$, et $1.39 \log_2 n$ pour les Uniform.



Théorème (unicité) : pour un ensemble de clés donné, il existe **un unique** Zip-Tree $\rightarrow$ la forme ne dépend plus de l'**historique des opérations**.

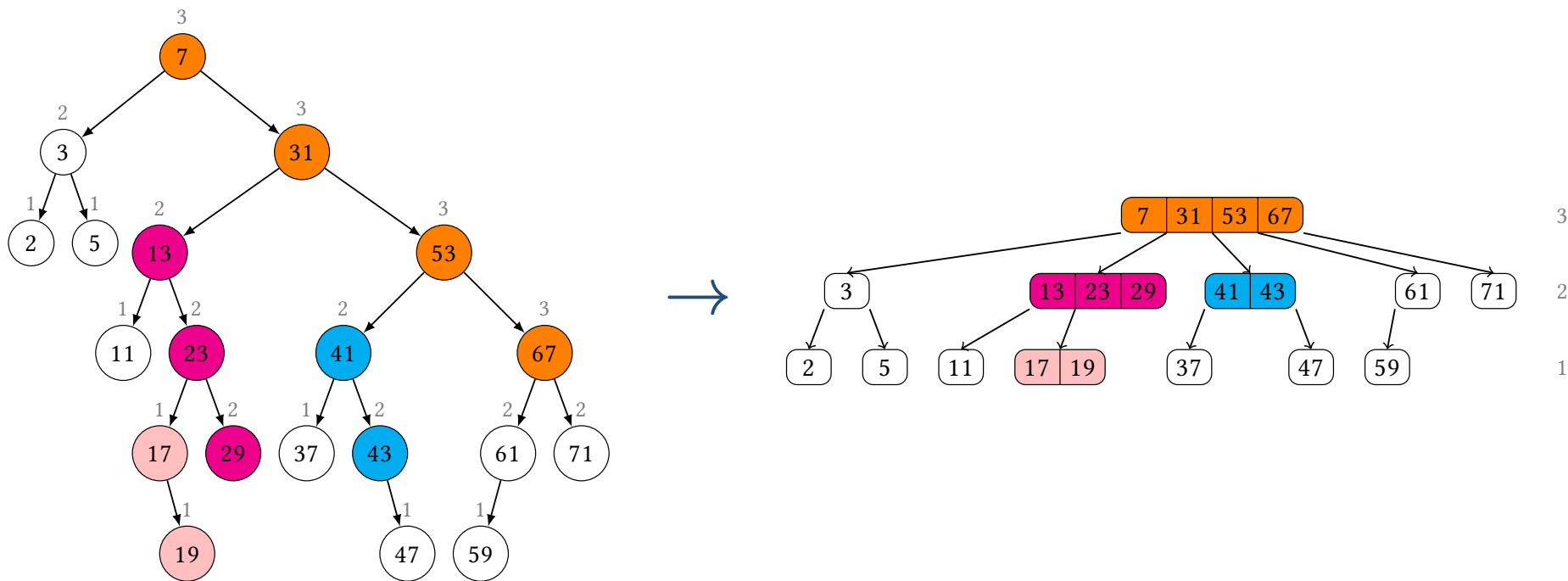
k-Zip-Trees : fusionner les chaînes de collisions

Dans un Zip-Tree, les nœuds de **même rang** forment une chaîne le long des fils droits : un **G-nœud** (Farmer & Meyer 2024) regroupe une chaîne dans un tableau trié immuable.



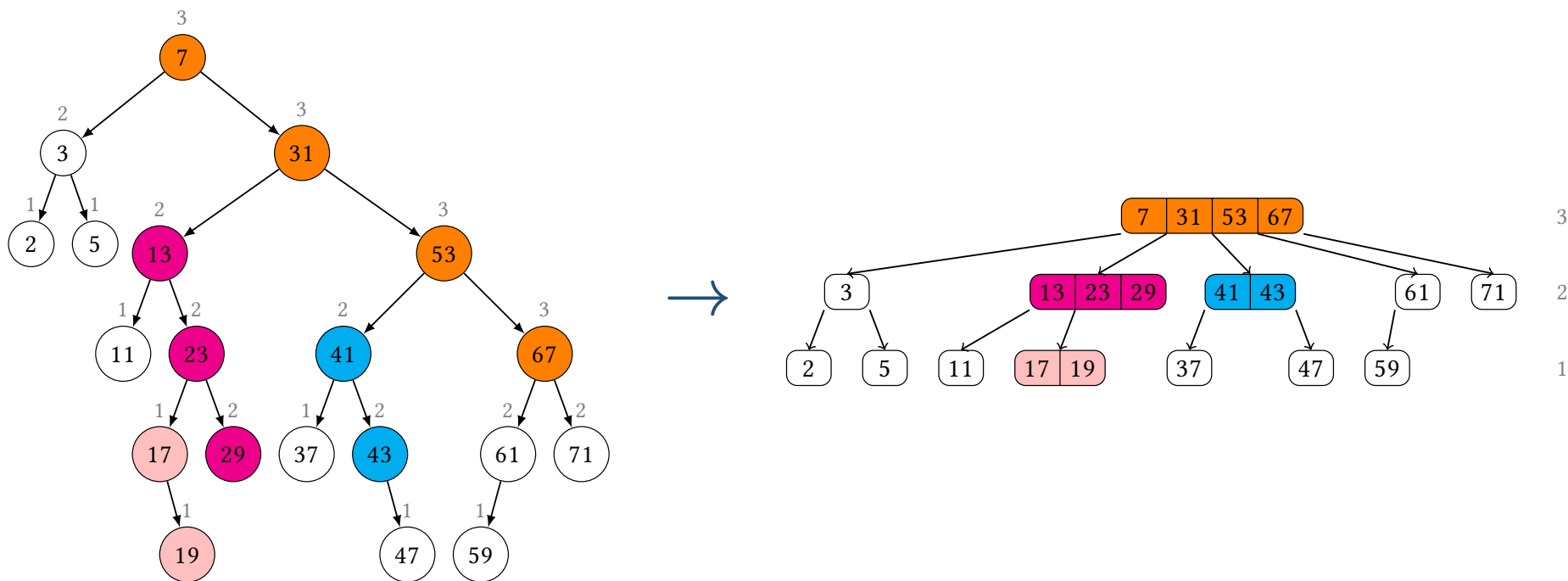
k-Zip-Trees : fusionner les chaînes de collisions

Dans un Zip-Tree, les nœuds de **même rang** forment une chaîne le long des fils droits : un **G-nœud** (Farmer & Meyer 2024) regroupe une chaîne dans un tableau trié immuable.



k-Zip-Trees : fusionner les chaînes de collisions

Dans un Zip-Tree, les nœuds de **même rang** forment une chaîne le long des fils droits : un **G-nœud** (Farmer & Meyer 2024) regroupe une chaîne dans un tableau trié immuable.



$\approx k + 1$ clés par nœud $\rightarrow$ hauteur $O(\log_{k+1} n)$: **la meilleure borne du lot.**
On instancie avec $k = 32$ (nœuds d'arité moyenne 33).

Contribution (1/2) : trois implémentations OCaml

Interface **compatible Map** → substituables sans modifier le code client.

Un seul foncteur, paramétré par un type ordonné muni d'un rang :

```
module type RankedOrderedType = sig
  type t
  val compare : t -> t -> int
  val rank : t -> int          (* dérivé du hash de la clé *)
end
```

- **Zip / Uniform Zip** : même foncteur, deux fonctions de rang.
Rang géométrique = nombre de zéros de poids faible, en $O(1)$ par **séquence de De Bruijn**.
- **G-trees** : foncteur générique paramétré par la structure des G-nœuds

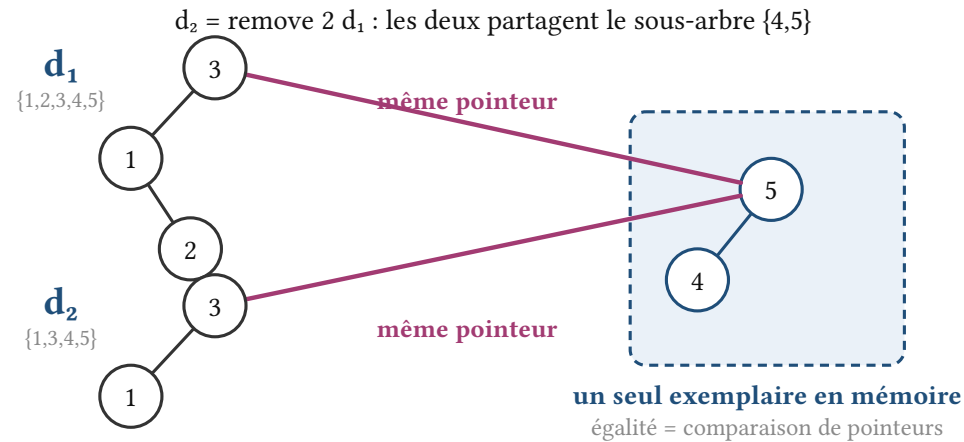
Contribution (2/2) : versions hash-consées

```

type raw_t =
  | Node of { l : t; v : key; d : value;
              r : t; rank : int }
  | Empty
and t = raw_t hash_consed

```

- Chaque nœud construit passe par une table de hash-consing.
- Hash d'un arbre en $O(1)$ à partir des **tags** des enfants.
- N'exige des clés et valeurs que tag, equal, compare, hash.



Deux dictionnaires égaux $\rightarrow$ **physiquement égaux** : égalité en $O(1)$, mémoïsation par *tag*.

Vérification : test différentiel avec Monolith

Monolith génère des séquences d'opérations aléatoires et compare, opération par opération, une **candidate** (nos arbres) à une **référence** (Map OCaml).

séquence d'opérations tirée par **AFL++** : add, remove, union, find_opt, ...

↓ même séquence des deux côtés ↓

Référence

Map OCaml

?

Candidate

nos arbres

- Les dictionnaires restent **abstraits** : jamais comparés entre eux, seuls les **résultats observables** le sont (find_opt, cardinal, parcours ordonné).
- Peu de clés distinctes → collisions et cas limites atteints rapidement.
- En cas de désaccord → **scénario minimal reproductible**.

Plusieurs bugs détectés très tôt. **Toutes les implémentations passent les tests.**

Résultats : les opérations de base

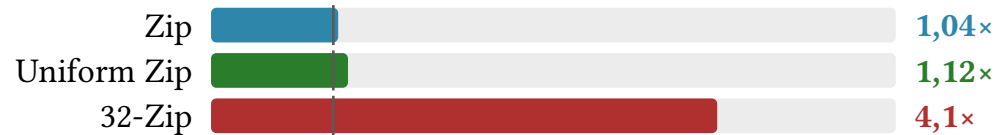
Profondeur moyenne mesurée ($n = 10^5$) : Zip $1.310 \log_2 n$, Uniform $1.215 \log_2 n$

- sous les bornes 1.5 et 1.39, et Uniform est bien le mieux équilibré.

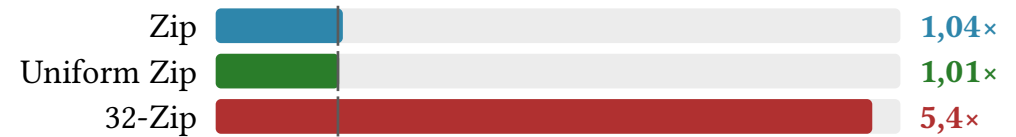
find



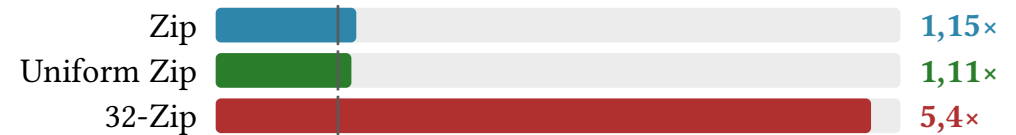
remove



add



update



Temps d'exécution rapporté aux Map OCaml (trait vertical = 1,00). **Plus court = mieux.**

Résultats : les opérations de base

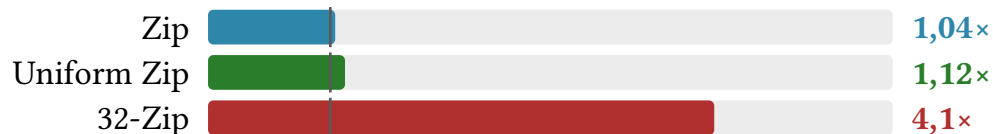
Profondeur moyenne mesurée ($n = 10^5$) : Zip $1.310 \log_2 n$, Uniform $1.215 \log_2 n$

- sous les bornes 1.5 et 1.39, et Uniform est bien le mieux équilibré.

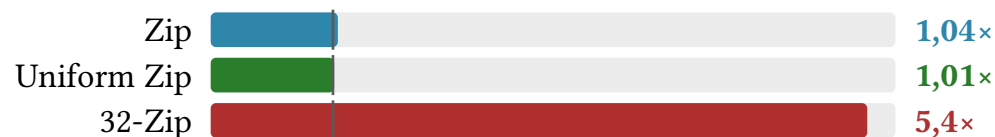
find



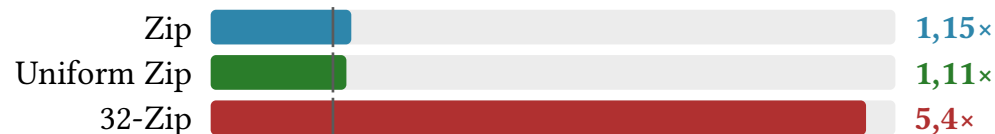
remove



add



update



Temps d'exécution rapporté aux Map OCaml (trait vertical = 1,00). **Plus court = mieux.**

32-Zip-Trees : chaque modification recopie un tableau (10–25× plus d'allocations).

Uniform Zip-Trees : à 1–12 % des Map OCaml → **structure retenue.**

Résultats : ce que rapporte le hash-consing

Benchmark : point fixe de Kleene sur une fonction 91 de McCarthy généralisée, puis unions et tests d'égalité des résultats - proche du travail d'un analyseur.

Points fixes (construction)



Unions



Égalité (× 5 000)



- Surcoût de **+55 %** à la construction, et le pic du tas passe de 10,6 à **62,8 Mo**
- Remboursé dès qu'on **compare** ou **combine** : égalité physique en $O(1)$ (**264× plus rapide**), unions mémorisées qui exploitent le partage (**6×**).

Résultats : ce que rapporte le hash-consing

Benchmark : point fixe de Kleene sur une fonction 91 de McCarthy généralisée, puis unions et tests d'égalité des résultats - proche du travail d'un analyseur.

Points fixes (construction)



Unions



Égalité (× 5 000)



- Surcoût de **+55 %** à la construction, et le pic du tas passe de 10,6 à **62,8 Mo**
- Remboursé dès qu'on **compare** ou **combine** : égalité physique en $O(1)$ (**264× plus rapide**), unions mémorisées qui exploitent le partage (**6×**).

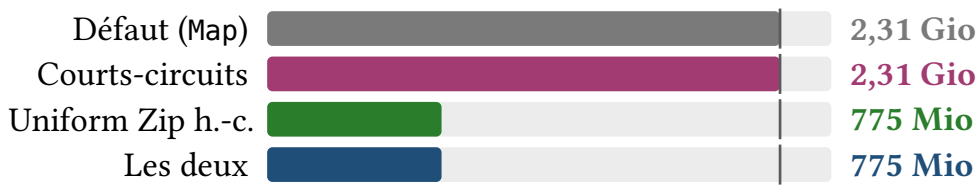
Un analyseur statique passe son temps à **comparer** des itérés et à **combinaer** des états : c'est exactement là que le hash-consing paie.

On utilise les Uniform Zip-Trees hash-consés pour le tas, fonctions sur le tas mémorisées.
Indépendamment, on ajoute des **courts-circuits** dans les fonctions qui manipulent le tas.

Temps d'analyse



Pic du tas



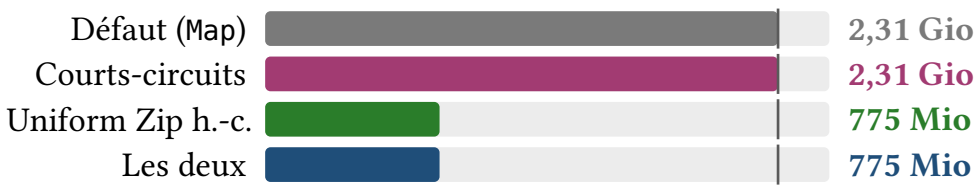
Rapporté à la configuration par défaut. **Plus court = mieux.**

On utilise les Uniform Zip-Trees hash-consés pour le tas, fonctions sur le tas mémorisées.
Indépendamment, on ajoute des **courts-circuits** dans les fonctions qui manipulent le tas.

Temps d'analyse



Pic du tas



Rapporté à la configuration par défaut. **Plus court = mieux.**

Deux gains **indépendants et cumulables** :
nos dictionnaires → **pic mémoire ÷ 3**, temps neutre courts-circuits → **-10 % de temps.**
Des analyses jusque-là **bloquées par la RAM** deviennent réalisables.

Contributions

- 3 familles d'arbres randomisés à représentation unique en OCaml, avec leurs versions hash-consées, interface compatible Map
- correction testée avec Monolith + AFL++
- évaluation expérimentale complète

Ce que ça ouvre

- hash-conserver les autres domaines abstraits de Salto
- publier la bibliothèque

Finale

Une meilleure borne asymptotique ne fait pas une meilleure structure : les **Uniform Zip-Trees hash-consés** gagnent, et divisent par 3 le pic mémoire de Salto.

Le rang en $O(1)$: les séquences de De Bruijn

Le rang d'un Zip-Tree est le nombre de **zéros de poids faible** du hash de la clé : à calculer sans boucle ni branchement. (Leiserson, Prokop & Randall 1998)

Hash de la clé 01101000 → on cherche à lire 3

Multiplier par 2^i , c'est décaler de i crans

x	01101000	le hash
-x	10011000	complément à deux
x land (-x)	00001000	isole le 1 de poids faible : 2^3
* 00011101	11101000	décale la séquence de 3 crans
>> 5	111	les 3 bits de tête = 7
table.(7)	3	le rang

rang i	2^i * 00011101	tête
0	00011101	000
1	00111010	001
2	01110100	011
3	11101000	111
4	11010000	110
5	10100000	101
6	01000000	010
7	10000000	100

00011101 est une séquence de De Bruijn : chacun des 8 motifs de 3 bits y est facteur **exactement une fois**.

Les 8 têtes sont **deux à deux distinctes** : c'est un **hash parfait** des puissances de deux.
En 64 bits : pareil avec une constante de 64 bits, décalage de 58, table de 64 entrées
Que 4 étapes en $O(1)$: inversion de signe, land, multiplication, décalage, accès.

Pourquoi nos mesures diffèrent de Farmer & Meyer

Dans (Farmer & Meyer 2024), les 32-Zip-Trees font des recherches $\approx 2\times$ **plus rapides** que les Zip-Trees. Nous mesurons l'inverse ($1,6\times$ plus **lentes**).

Leurs Zip-Trees de comparaison sont eux-mêmes **instanciés comme des G-trees** : ils paient le même surcoût de structure intermédiaire.
Nous comparons à une **implémentation directe** des Zip-Trees, bien plus légère.

Autres écarts de contexte :

- notre implémentation est **persistante et immuable** $\rightarrow$ les tableaux des G-nœuds sont recopiés à chaque modification ;
- nous mesurons aussi add, remove, update, pas seulement la recherche.

Pourquoi pas des arbres de Patricia ?

Les arbres de Patricia (Okasaki–Gill) sont eux aussi **canoniques** : la forme ne dépend que de l'ensemble des clés. C'est la solution employée en pratique dans les analyseurs statiques (les Hptmap hash-consées de Frama-C).

Leurs limites

- exigent des clés **entières** (ou une injection vers les entiers)
- branchement sur les **bits**, pas sur compare → pas d'ordre utilisateur
- `min_binding`, parcours ordonné, `split` : non gratuits

Nos structures

- tout type ordonné **hachable**
- interface Map complète, ordre de l'utilisateur préservé
- substituables sans toucher au code client de Salto

Comparaison expérimentale directe avec les Patricia tries : **travail futur.**

Preuve de l'unicité de représentation

Théorème. Pour tout ensemble fini de clés munies de leurs rangs, il existe un unique Zip-Tree contenant exactement ces clés.

Preuve, par induction sur le nombre de clés :

- **Cas vide** : l'unique représentation est l'arbre vide.
- **Cas non vide** : les deux invariants forcent la racine à être le nœud de **plus petite clé parmi les nœuds de rang maximal** (tas max selon les rangs, puis départage en faveur de la plus petite clé).
- Les clés restantes se répartissent alors de façon forcée : les plus petites à gauche, les plus grandes à droite (arbre de recherche), et par hypothèse d'induction chacun de ces deux sous-arbres est unique.

Le rang étant une fonction déterministe de la clé, deux dictionnaires ayant les mêmes associations ont **exactement le même arbre**.

Et si la fonction de hachage est mauvaise ?

Toutes les garanties probabilistes supposent que le hash se comporte comme une **fonction aléatoire uniforme** sur son domaine.

- **Hash de mauvaise qualité** → rangs corrélés aux clés → l'arbre peut dégénérer. L'unicité de représentation, elle, est **toujours préservée**
- **Clés adversariales** : un attaquant qui contrôle les clés et connaît la fonction de hachage peut forcer une hauteur linéaire - même situation que pour une table de hachage. Non pertinent dans Salto, dont les clés viennent du programme analysé.
- Dans nos implémentations, les clés sont déjà hash-consées : le hash est **déjà calculé**, donc le rang est gratuit.

	$n = 100$	$n = 10^3$	$n = 10^4$	$n = 10^5$
Hauteur / $\log_2 n$ (écart type)				
Map OCaml	1.190 (0.054)	1.220 (0.038)	1.240 (0.038)	1.254 (0.024)
Zip-Tree	2.177 (0.364)	2.566 (0.300)	2.817 (0.248)	2.984 (0.207)
Uniform Zip-Tree	1.852 (0.236)	2.111 (0.194)	2.271 (0.158)	2.388 (0.131)
Profondeur moyenne / $\log_2 n$ (écart type)				
Map OCaml	0.760 (0.010)	0.843 (0.005)	0.888 (0.003)	0.915 (0.002)
Zip-Tree	1.045 (0.114)	1.184 (0.080)	1.262 (0.063)	1.310 (0.050)
Uniform Zip-Tree	0.977 (0.089)	1.103 (0.063)	1.171 (0.049)	1.215 (0.039)
32-Zip-Tree, normalisé par $\log_{32} n$				
hauteur	1.539 (0.262)	1.330 (0.260)	1.229 (0.168)	1.217 (0.112)
profondeur moyenne	0.754 (0.237)	0.808 (0.252)	0.837 (0.163)	0.902 (0.103)

Moyennes sur 5 000 arbres. L'écart Zip / Uniform ($\approx 8\%$) correspond au rapport théorique 1.5/1.39.

	find	add	remove	update
Allocations mineures par exécution (mots)				
Map OCaml	0	3,76 Mw	1,97 Mw	4,23 Mw
Zip-Tree	0	5,35 Mw	2,73 Mw	6,27 Mw
Uniform Zip-Tree	0	5,07 Mw	2,60 Mw	5,95 Mw
32-Zip-Tree	924 kw	61,7 Mw	50,2 Mw	73,0 Mw
Allocations majeures par exécution (mots)				
Map OCaml	0	849 kw	356	1,35 Mw
Zip-Tree	0	1 004 kw	663	1,82 Mw
Uniform Zip-Tree	0	921 kw	612	1,67 Mw
32-Zip-Tree	11	4 447 kw	401 173	7,66 Mw

Les 32-Zip-Trees allouent sur find, là où toutes les autres implémentations n'allouent **rien** : toute modification d'un G-nœud recopie son tableau.

	Map OCaml	Uniform Zip	Uniform Zip hash-consé
Temps (s)			
Points fixes	2.199	2.449	3.788
Unions	0.019	0.020	0.003
Égalité (× 5 000)	1.321	1.451	0.005
Allocations			
Points fixes	2,94 Go	340,5 Mo	590,4 Mo
Unions	25,4 Mo	20,3 Mo	1,2 Mo
Égalité (× 5 000)	6,37 Go	2,1 Mo	2,1 Mo
Mémoire			
Pic du tas	10,61 Mo	10,61 Mo	62,83 Mo
Résident (après GC)	8,30 Mo	8,23 Mo	18,76 Mo

Le hash-consing coûte de la mémoire **ici** : 10 dictionnaires seulement, donc peu de partage à amortir, et la table de hash-consing retient tout. Dans Salto, où des milliers de tas se recouvrent largement, le bilan s’inverse (pic ÷ 3).