Gestion de la mémoire

Langages et mémoire

Différence principale entre

- langages de haut niveau :
  OCaml, Java, C#
- langages de bas niveau :
  C, C++ :

Gestion de la mémoire

Gestion manuelle de la mémoire

Le programmeur doit explicitement allouer et libérer des zones mémoire (`malloc`, `free`).

Avantages :

- contrôle fin de la mémoire
- possibilité de libérer une zone dès qu'elle n'est plus utile
- possibilités d'optimisation

Inconvénients :

- Source de nombreuses erreurs (`segfault` !)
  - accès à une zone non allouée ou déjà libérée ;
  - libération d'une zone déjà libérée
  - non-libération de zones non utilisées (fuite mémoire)

Gestion automatique de la mémoire

L'espace mémoire est automatiquement alloué par les constructeurs

Il est libéré via l'utilisation d'un *Garbage Collector* (ramasse-miette, glanneur de cellules)

- cherche quels objets ne peuvent plus être utilisés par le programme ;
- libère l'espace mémoire correspondant.

Première apparition : LISP (McCarthy, 1959)

Garbage Collector

Avantages :

- le programmeur peut se concentrer sur l'algorithmique
- code plus sûr (pas de segfault)
- pas (ou peu) de fuite de mémoire

Inconvénients :

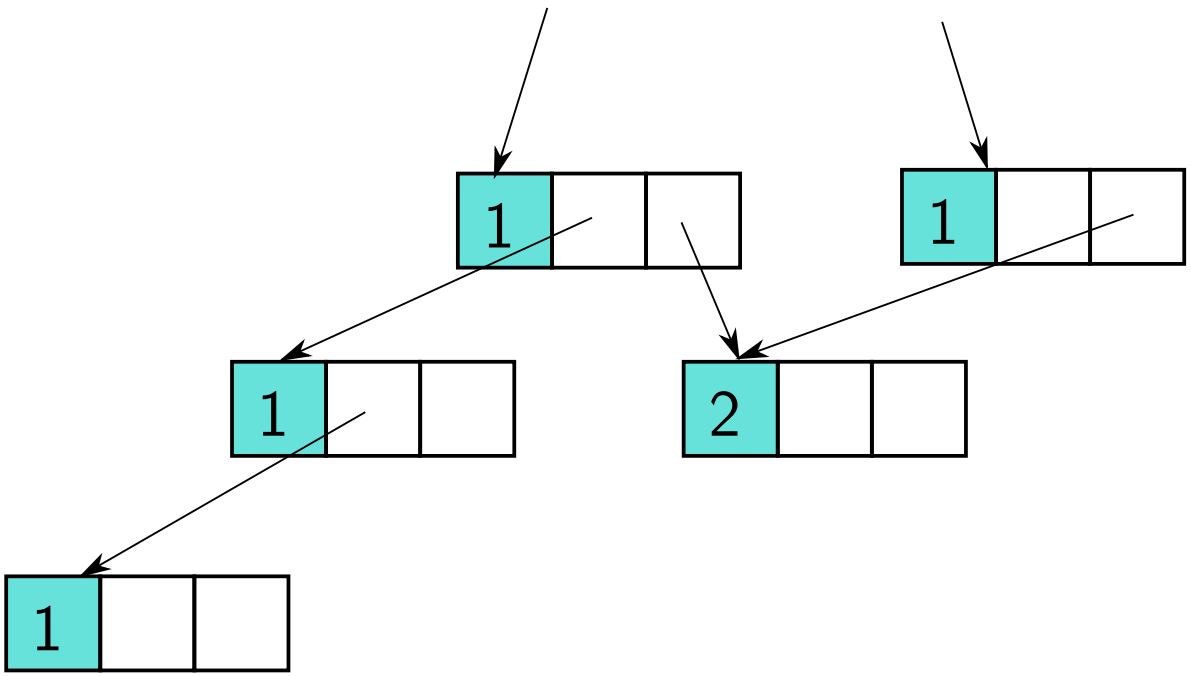
- Surcoût en temps et en mémoire
- Peut bloquer/ralentir le programme
- Moins de contrôle du moment où la donnée est libérée
- Le GC peut être amené à changer l'emplacement des données en mémoire

Deux grandes familles :

- comptage de références
- GC traversants

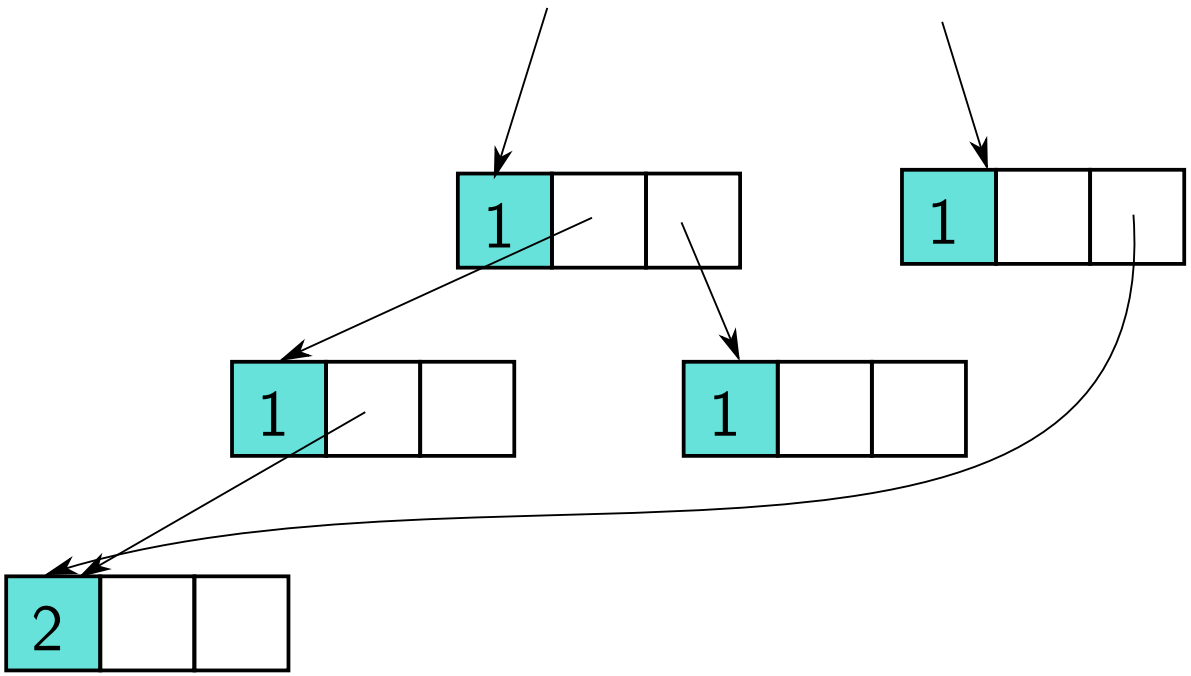
Comptage de références

Principe : on associe à chaque zone mémoire un compteur qui indique le nombre de références qui pointent sur elle
Incrémente/décrémente ce compteur / changement référence
Quand ce compteur atteint 0, la zone n'est plus accessible par personne, on peut la libérer



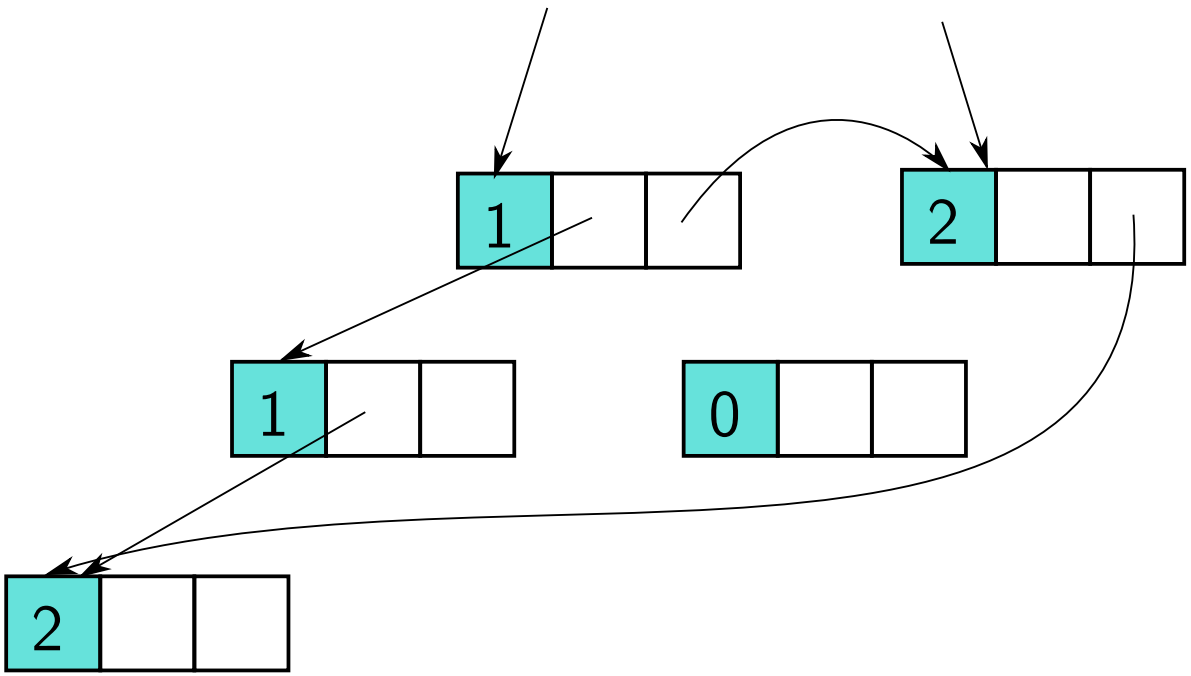
Comptage de références

Principe : on associe à chaque zone mémoire un compteur qui indique le nombre de références qui pointent sur elle
Incrémente/décrémente ce compteur / changement référence
Quand ce compteur atteint 0, la zone n'est plus accessible par personne, on peut la libérer



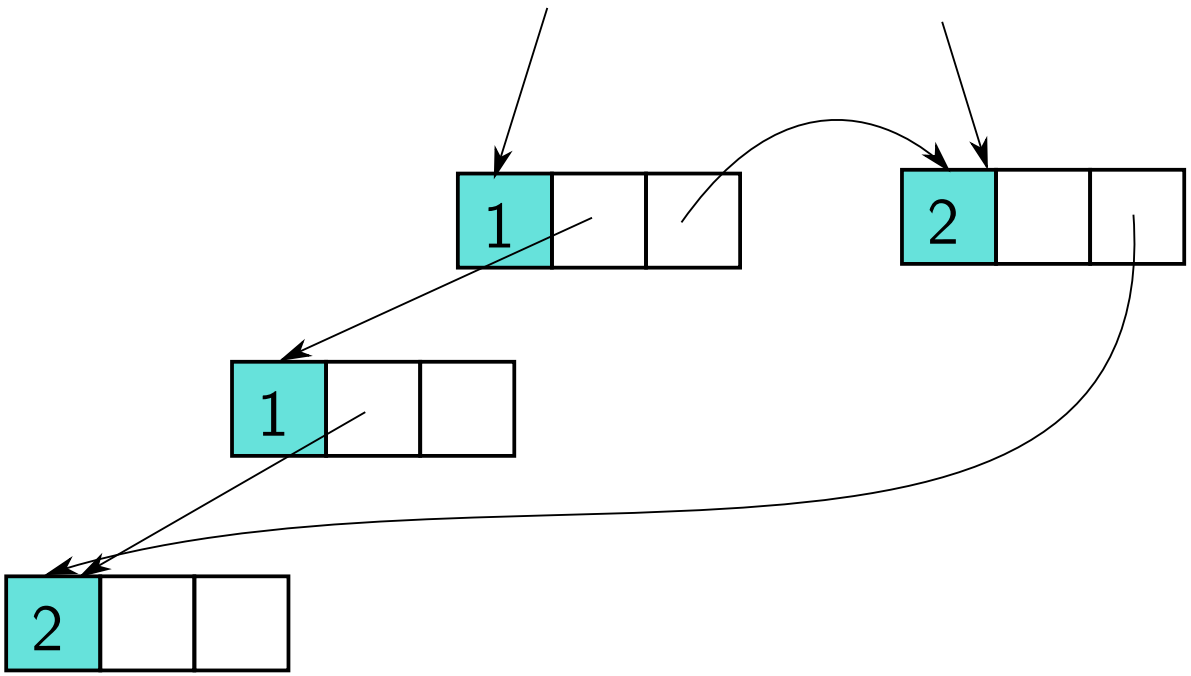
Comptage de références

Principe : on associe à chaque zone mémoire un compteur qui indique le nombre de références qui pointent sur elle
Incrémente/décrémente ce compteur / changement référence
Quand ce compteur atteint 0, la zone n'est plus accessible par personne, on peut la libérer



Comptage de références

Principe : on associe à chaque zone mémoire un compteur qui indique le nombre de références qui pointent sur elle
Incrémente/décrémente ce compteur / changement référence
Quand ce compteur atteint 0, la zone n'est plus accessible par personne, on peut la libérer



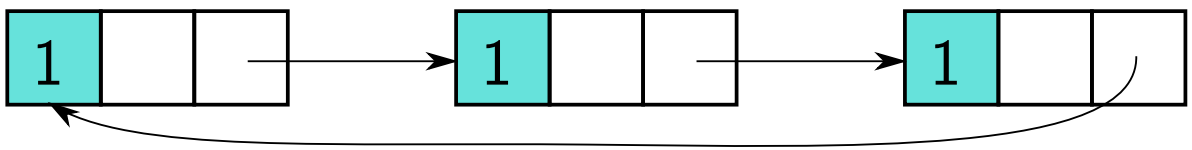
Comptage de références (suite)

Avantages :

- simple conceptuellement
- libération rapide

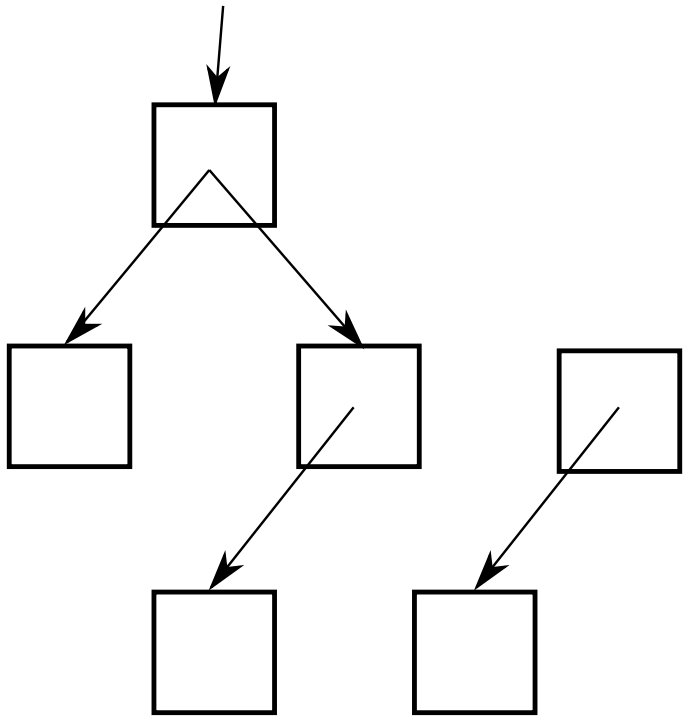
Inconvénients :

- surcoût en mémoire (taille du compteur)
- surcoût en temps (gestion des références)
- données cycliques jamais libérées



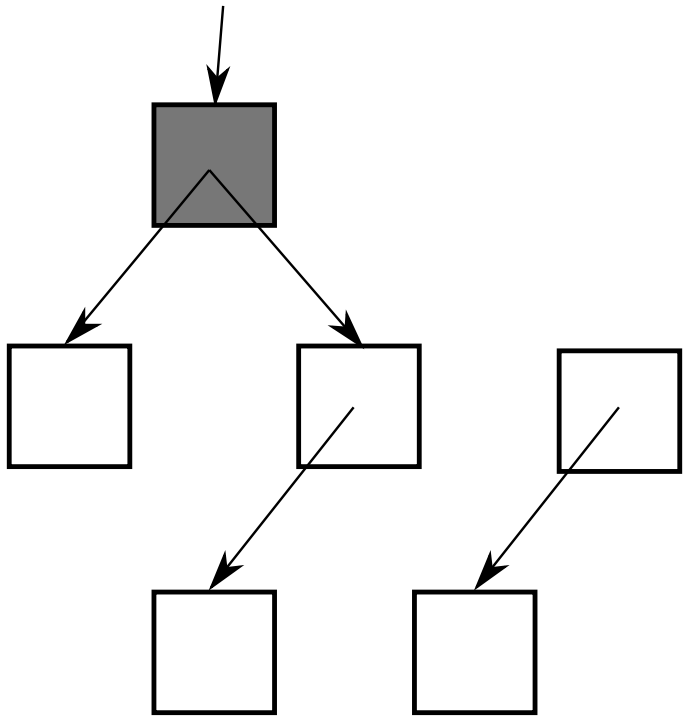
GC traversant

Principe : on parcours l'ensemble des données accessibles à partir des racines (variables du programme)
Les données accessibles sont conservées, les autres libérées



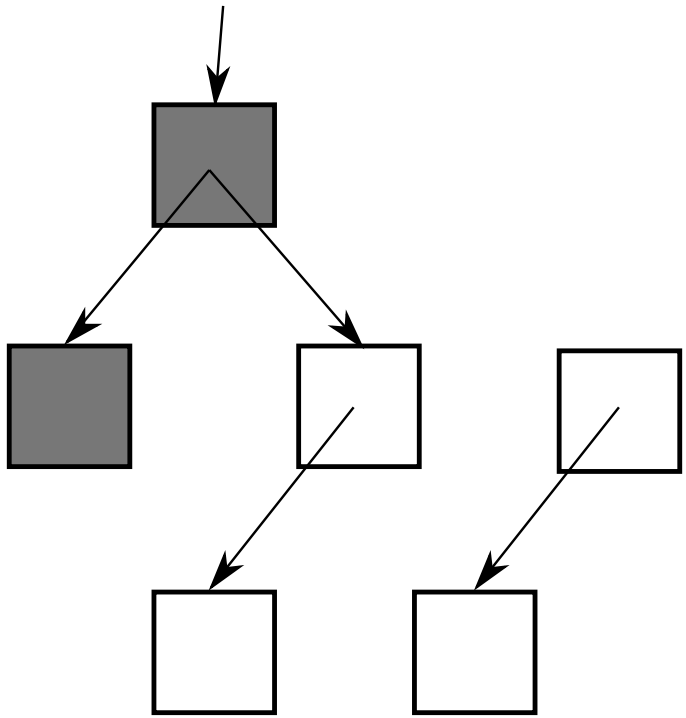
GC traversant

Principe : on parcours l'ensemble des données accessibles à partir des racines (variables du programme)
Les données accessibles sont conservées, les autres libérées



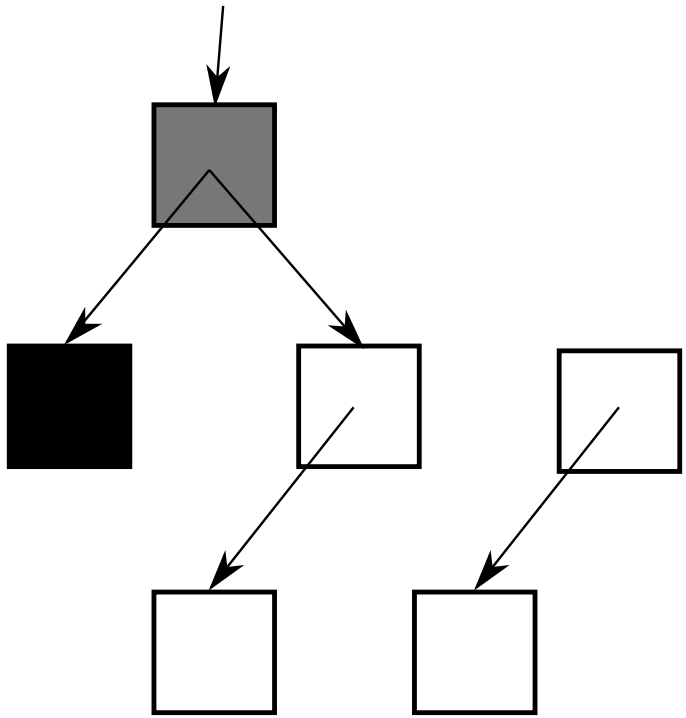
GC traversant

Principe : on parcours l'ensemble des données accessibles à partir des racines (variables du programme)
Les données accessibles sont conservées, les autres libérées



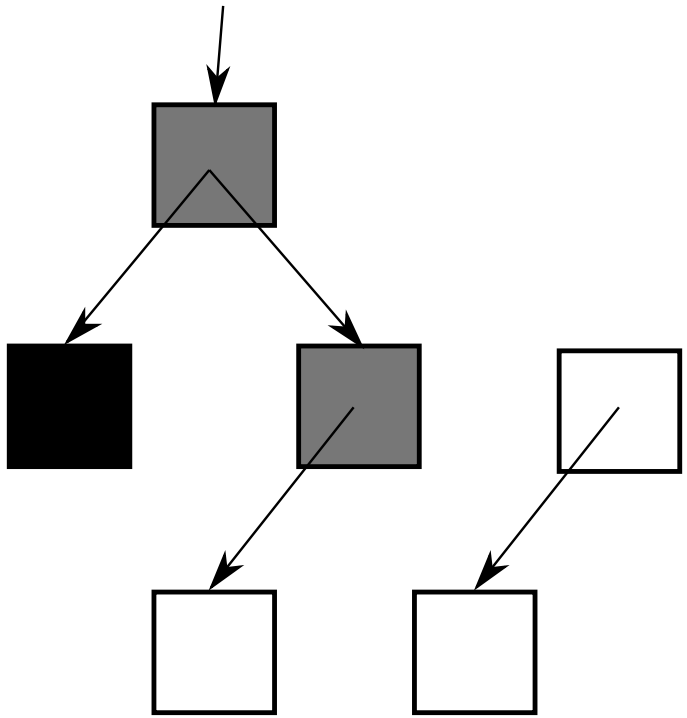
GC traversant

Principe : on parcours l'ensemble des données accessibles à partir des racines (variables du programme)
Les données accessibles sont conservées, les autres libérées



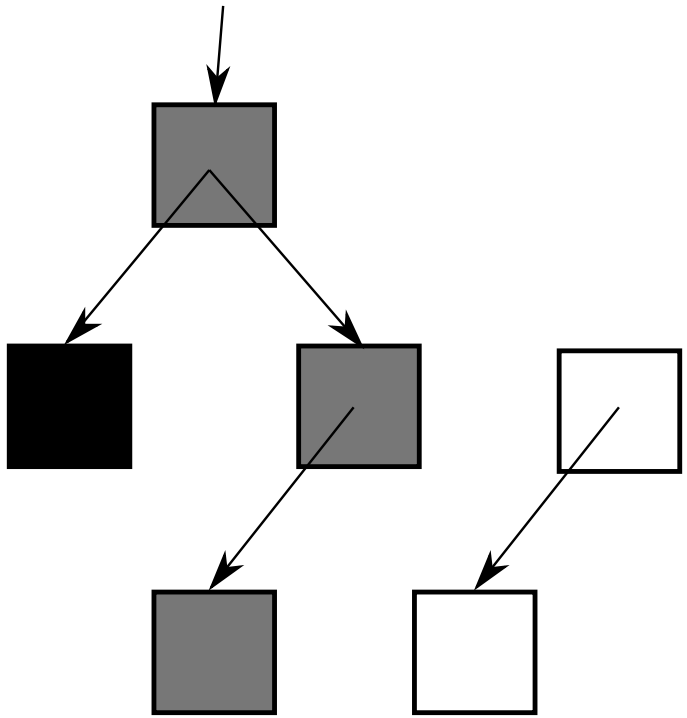
GC traversant

Principe : on parcours l'ensemble des données accessibles à partir des racines (variables du programme)
Les données accessibles sont conservées, les autres libérées



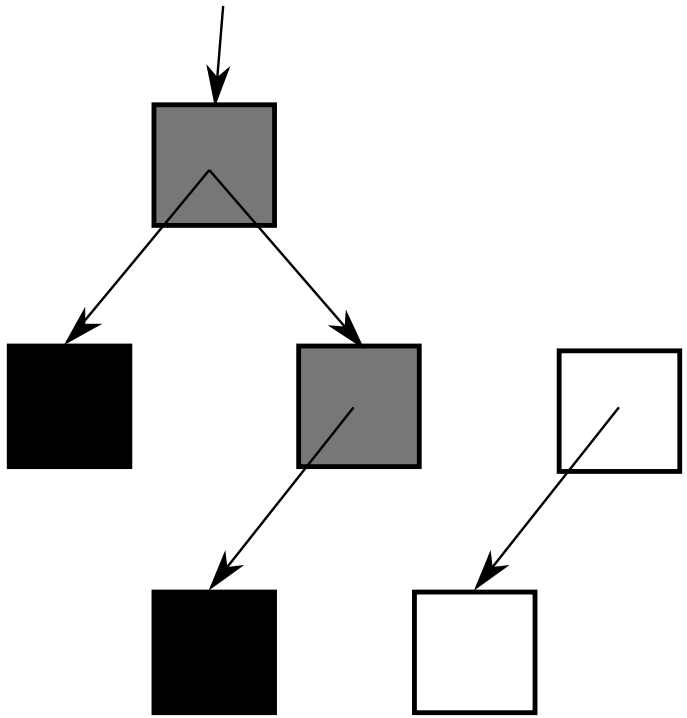
GC traversant

Principe : on parcours l'ensemble des données accessibles à partir des racines (variables du programme)
Les données accessibles sont conservées, les autres libérées



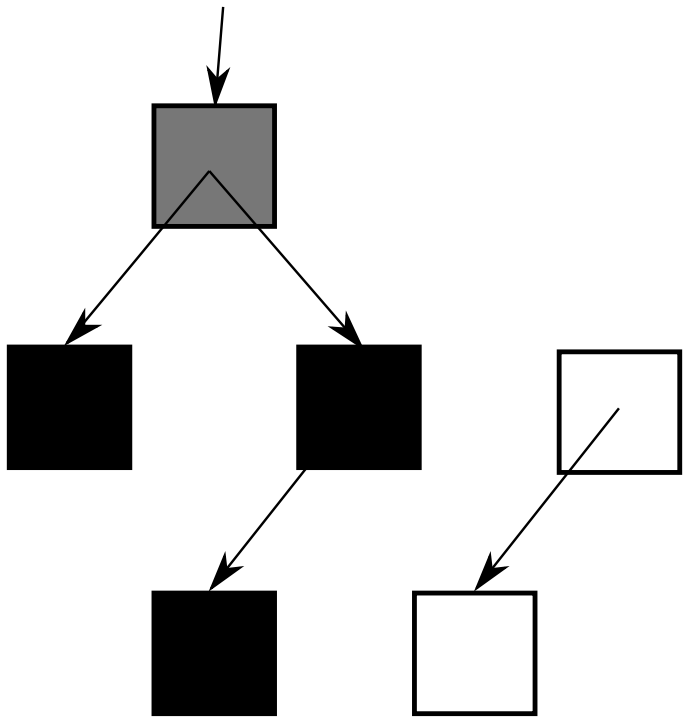
GC traversant

Principe : on parcours l'ensemble des données accessibles à partir des racines (variables du programme)
Les données accessibles sont conservées, les autres libérées



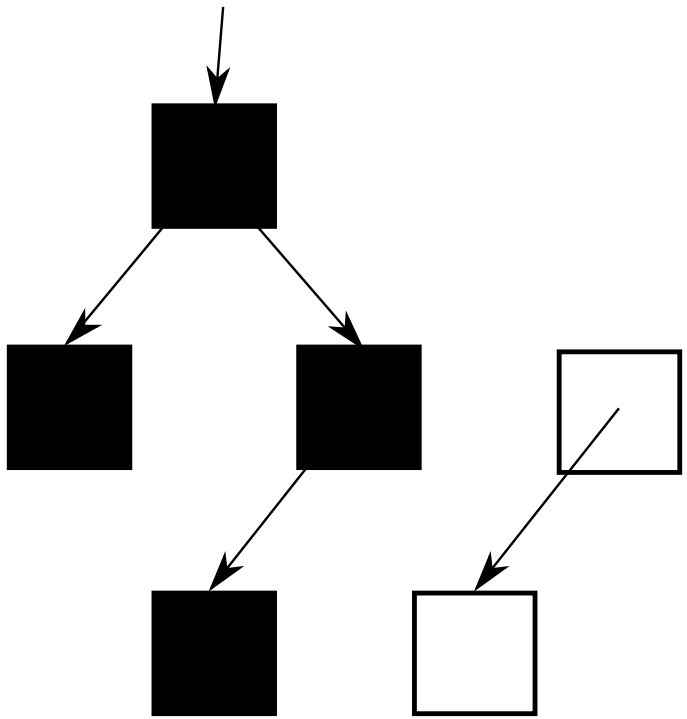
GC traversant

Principe : on parcours l'ensemble des données accessibles à partir des racines (variables du programme)
Les données accessibles sont conservées, les autres libérées



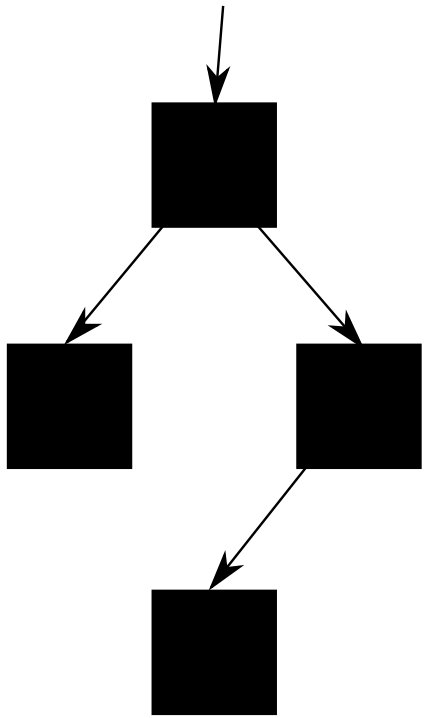
GC traversant

Principe : on parcours l'ensemble des données accessibles à partir des racines (variables du programme)
Les données accessibles sont conservées, les autres libérées



GC traversant

Principe : on parcours l'ensemble des données accessibles à partir des racines (variables du programme)
Les données accessibles sont conservées, les autres libérées



Marquer et nettoyer

On ajoute un bit aux zones mémoire pour indiquer leur accessibilité (blanc/noir)
On parcours ensuite la mémoire pour supprimer les zones restées blanches

Inconvénients :

- Surcoût en mémoire
- Balayage de la mémoire pour la libération

GC copiant

On sépare la mémoire en deux, on n'en utilise que la moitié
On recopie les zones accessibles dans l'autre moitié
On libère toute la première moitié, et on inverse les rôles

Avantages :

- Libération de la mémoire instantanée
- Après recopie, la mémoire est compacte : une zone mémoire est proche des zones vers lesquelles elle pointe (cache)

Inconvénients :

- Seule la moitié de la mémoire est disponible
- Les données changent de place dans la mémoire

En pratique

Combinaison de plusieurs techniques à différents niveaux

Utilisation de zones mémoires différentes suivant la "génération" des données, avec des techniques et des fréquences différentes

En OCaml :

- tas mineur / tas majeur
- GC mineur : copie des accessibles du tas mineur vers le tas majeur
- GC majeur : marquer et nettoyer incrémental sur le tas majeur

Partage maximal

Minimisation de la mémoire

En C, gestion fine de la mémoire :

- version destructive/non destructive des algorithmes
- changements sur place/recopie

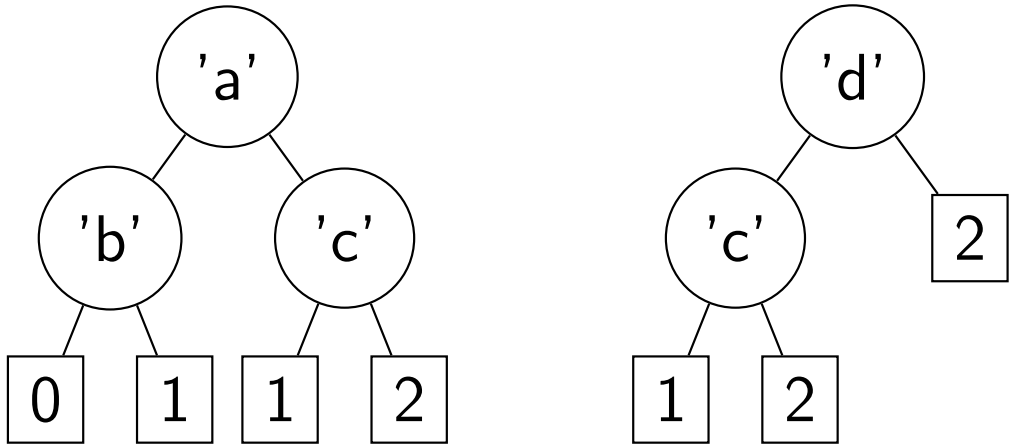
En OCaml, uniquement version non destructive

Comment minimiser l'espace mémoire ?

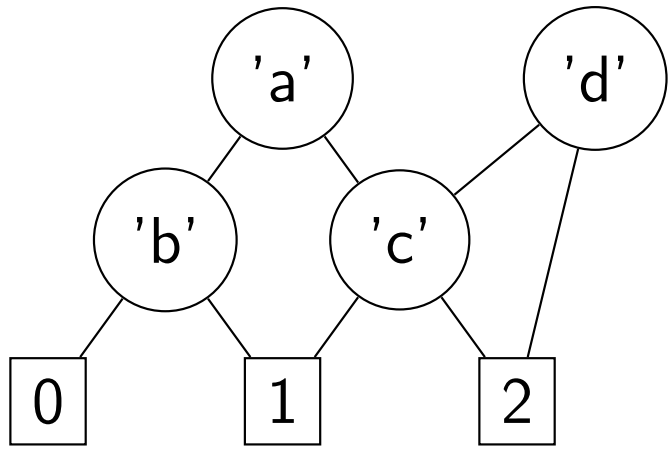
Partage maximal

Arbre binaires stricts :

Sans partage :



Avec partage maximal :



Égalité en OCaml

- = : égalité structurelle
  on parcours la structure pour vérifier que les sous-structures sont égales
- == : égalité physique
  on compare l'adresse en mémoire

Égalité structurelle coûteuse (parcours)

`a == b` implique `a = b`
mais la réciproque est fausse en général

- vraie avec partage maximal

Hash consing

Implémentation du partage maximal :

On stocke les valeurs déjà existantes dans une table de hachage

Quand on cherche à en créer une nouvelle (utilisation d'un constructeur), on vérifie d'abord si une valeur structurellement égale n'est pas déjà présente dans la table

- si oui, on retourne cette valeur
- sinon, on insère la nouvelle valeur dans la table, associée à elle-même

Sur les arbres binaires stricts

On crée une table de hachage avec

- clefs : arbres binaires stricts
- valeurs : arbres binaires stricts

Quand on veut créer un nouvel arbre, on vérifie qu'il n'est pas déjà présent à l'aide d'une fonction `hashcons` :

```
let table = creer 251

let hashcons x =
  try chercher table x
  with Not_found ->
    inserer table x x; x
```

Constructeurs

```
type ('a,'b) abs =
    Feuille of 'a
  | Noeud of ('a,'b) abs * 'b * ('a,'b) abs
```

Pour être certains d'avoir un partage maximal, on introduit des constructeurs intelligents :

```
let feuille i = hashcons (Feuille i)
let noeud (fg, n, fd) = hashcons (Noeud(fg,n,fd))
```

(on peut utiliser les types privés pour être certains de n'utiliser que ces constructeurs)

Raffinement de l'égalité

Si on suppose qu'on a le partage maximal, on peut utiliser l'égalité physique pour les sous-noeuds quand on cherche dans la table de hachage :

```
let egal x y = match x,y with
    Feuille i, Feuille j -> i == j
  | Noeud(gx,nx,dx), Noeud(gy,ny,dy) ->
      gx == gy && nx == ny && dx == dy

let rec rechercher_assoc l k = match l with
  | [] -> raise Not_found
  | (k',v)::q -> if egal k k' then v
                 else rechercher_assoc q k

let rechercher d k =
  let h = hash k in rechercher_assoc d.(h)
```

Hash consing et GC

Puisque les valeurs sont stockées dans la table de hachage, elles sont toujours accessibles

Même si rien d'autre ne pointe dessus

On utilise des **pointeurs faibles** dans la table de hachage : ceux-ci ne sont pas pris en compte par le GC