Récupération automatique de la mémoire

Introduction

Les langages de haut niveau s'occupent, à la place du programmeur, de la récupération de la mémoire occupée par les objets abandonnés.

La récupération automatique de la mémoire offre plusieurs avantages:

- Simplification de la programmation dans le langage
  - Pas de code à écrire pour récupérer la mémoire
  - Pas besoin de "tordre" le style de programmation
- Moins de bogues dans les programmes
  - Plus de libération trop hâtive d'objets (*dangling pointers*)
  - Plus de fuites de mémoire involontaires (i.e. pas demandées expressément par le programme)

Introduction

Définition: *GC* (pour *Garbage Collector* ou *Glaneur de Cellules*): routine fournie par l'implantation du langage qui a pour tâche de libérer l'espace occupé par les objets abandonnés.

Le GC fait partie du *système de gestion de la mémoire*, lequel fournit les services:

- d'allocation des objets,
- d'accès en lecture et en écriture aux objets et
- de libération des objets.

Conséquence: on ne peut pas toujours dissocier le GC du reste de la gestion mémoire.

Introduction

Définition: *Vivant* (ou *accessible* ou *détenu par le programme*): objet qui est directement référencé par une racine du programme ou par un autre objet vivant.

Définition: *Mort* (ou *abandonné*): objet qui n'est plus vivant.

Définition: *Racine*: référence qui est directement accessible au programme.

Exemple: souvent, les racines d'un programme sont l'ensemble des variables globales et la pile d'exécution.

Introduction

Note: le fait qu'un objet soit vivant *ne* signifie *pas* qu'il sera nécessairement utilisé ultérieurement.

Exemple:

```
let x = /* grosse structure de donnees */ in
  if (/* test complexe */)
    then x
    else 5
```

Entre le '`in`' et le '`if`', le GC doit considérer la structure de données comme étant vivante même si le test risque de s'avérer faux par la suite.

Note:

peut-être utilisé plus tard $\Rightarrow$ vivant

mort $\Rightarrow$ certainement plus utilisé

GC par compteurs de référence

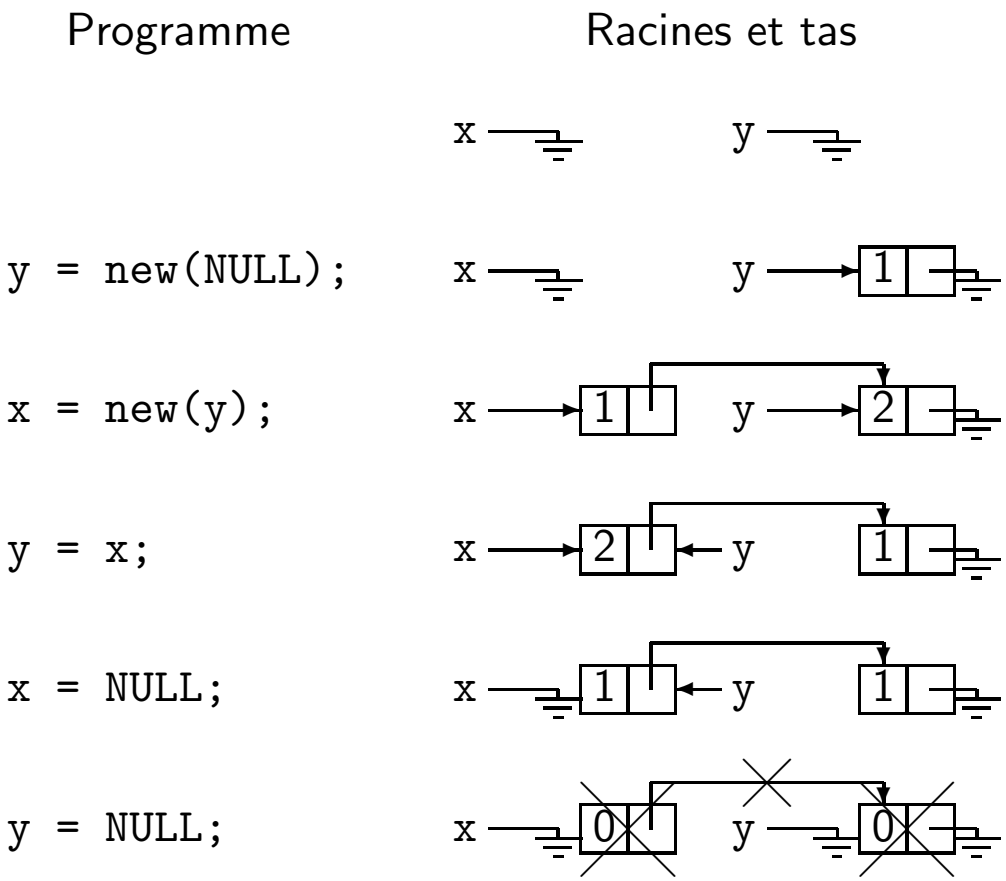
Description:

- Chaque objet alloué dans le tas comporte un champ supplémentaire qui sert à compter le nombre de références qui pointent sur l'objet.
- Les opérations de modification des références doivent faire la mise à jour des compteurs.
- Quand le compteur d'un objet tombe à zéro, ce dernier est détruit.

GC par compteurs de référence

Exemple:

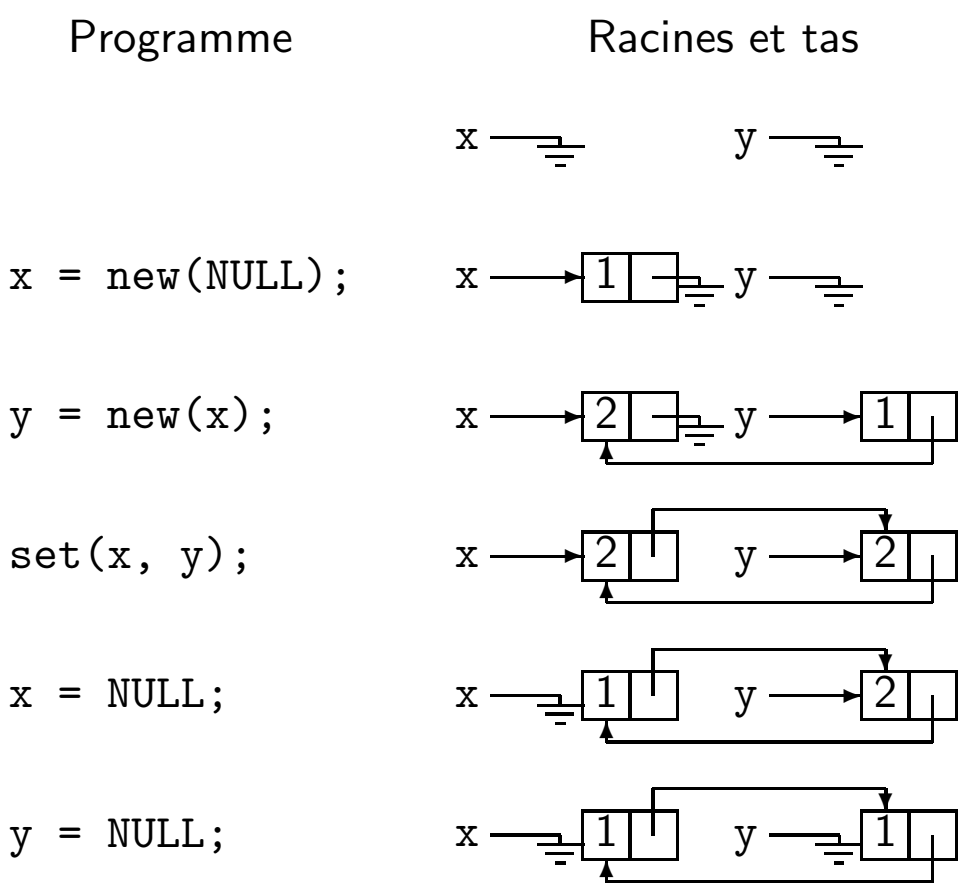
On suppose qu'on a un type '`cell`' qui ne contient qu'un champ. On dispose des opérations '`new`', '`get`' et '`set`' pour manipuler les cellules.



GC par compteurs de référence

Le GC par compteurs de références fonctionne incorrectement en présence de cycles dans les structures de données.

Exemple:



Fuite de mémoire!

GC par compteurs de référence

Avantages et inconvénients:

+ Technique simple.
+ Il est facile de faire générer le code de mise à jour des compteurs par le compilateur.
– Ce GC ne traite pas les cycles dans les structures de données.
– À cause de la mise à jour des compteurs, ce GC est plus coûteux que les autres technique de GC.

Les techniques de GC suivantes n'utilisent pas de compteurs pour déterminer la vivacité des objets mais plutôt une *traversée de graphe*.

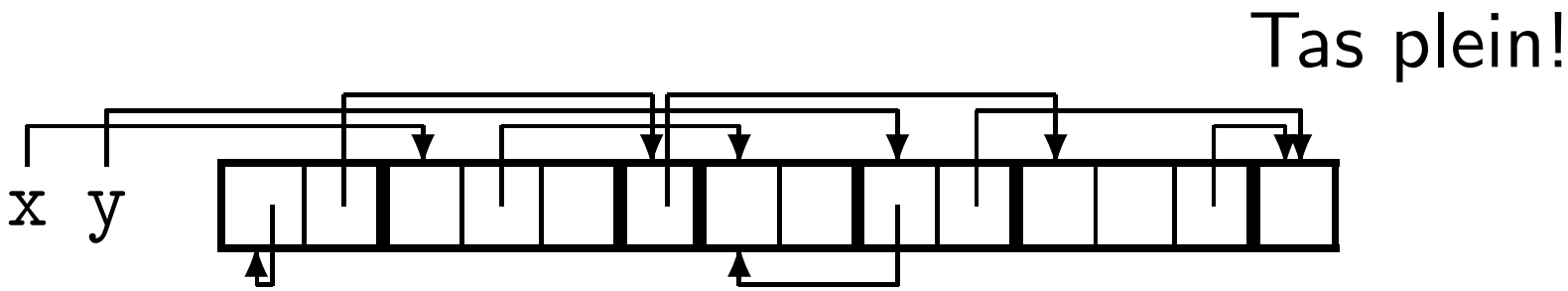
GC “Mark-and-sweep”

Description:

- Chaque objet contient un bit supplémentaire dit *bit de marquage*.
- Au début d'un cycle de GC, tous ces bits sont éteints.
- Par une traversée de graphe, le GC allume le bit de marquage de tous les objets vivants (accessibles). C'est la phase “mark”.
- La traversée considère les objets comme étant des noeuds et les références comme étant des arcs; elle débute à partir des racines; elle se termine lorsque tous les objets accessibles directement ou indirectement à partir des racines ont été visités.
- Les objets dont le bit de marquage est resté éteint sont morts.
- Par une traversée séquentielle du tas, le GC libère l'espace occupé par les objets morts et éteint le bit de marquage des objets vivants. C'est la phase “sweep”. Cette traversée clôt le cycle de GC.
- Un cycle de GC est déclenché chaque fois que le tas est trop plein; typiquement, c'est lorsque la prochaine allocation ne peut être effectuée par manque d'espace.

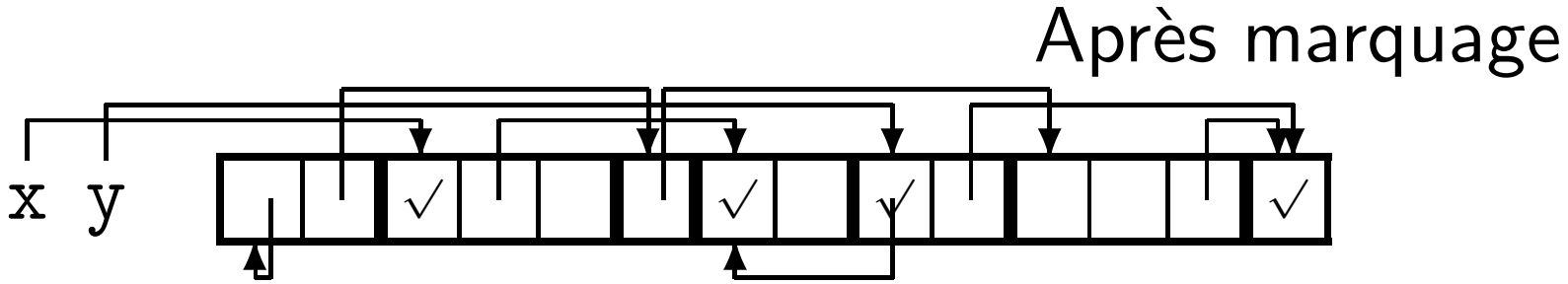
GC “Mark-and-sweep”

Exemple:



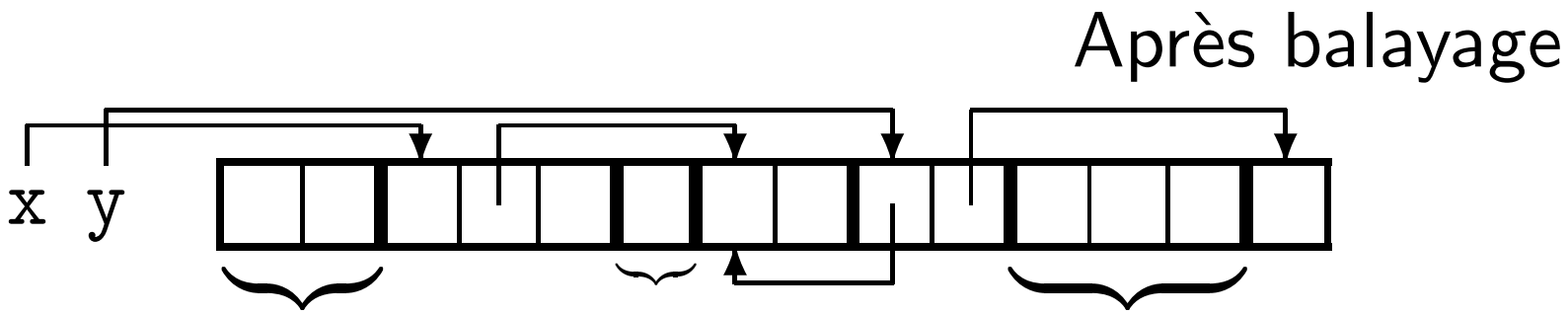
GC “Mark-and-sweep”

Exemple:



GC “Mark-and-sweep”

Exemple:



GC “Mark-and-sweep”

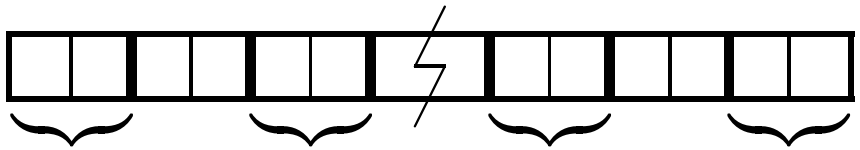
Avantages et inconvénients:

+ Ce GC ramasse tous les objets morts.

– De la fragmentation s'installe à la longue si les objets n'ont pas tous la même longueur.

? Chaque cycle de GC prend un temps proportionnel à la taille du tas.

Exemple de fragmentation de l'espace libre:

tas: 

Il y a beaucoup d'espace libre au total mais, à cause de la fragmentation, il est impossible d'allouer un objet de longueur 3.

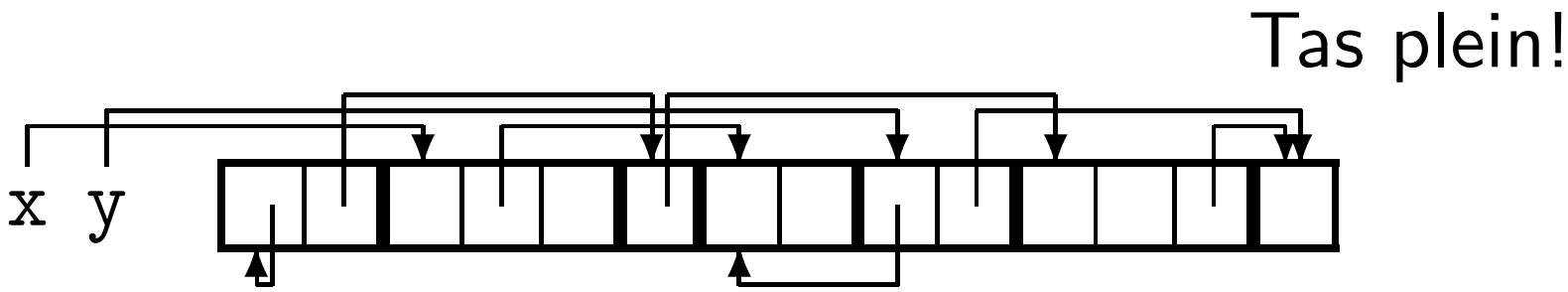
GC “Mark-and-compact”

Description:

- À l’instar du GC “Mark-and-sweep”, un cycle de GC commence par une phase de marquage.
- À ce point, un objet est vivant si et seulement si son bit de marquage est allumé.
- Par une traversée séquentielle du tas, le GC fait glisser les objets marqués vers le bas du tas pour les disposer consécutivement. Il en profite pour éteindre le bit de marquage de ces objets. Les références vers les objets sont mises à jour pour réfléter leur nouvelle position. C’est la phase “compact”. Cette traversée clôt la phase de GC.
- Après le compactage, le tas contient une zone dense d’objets vivants et une zone d’espace libre.

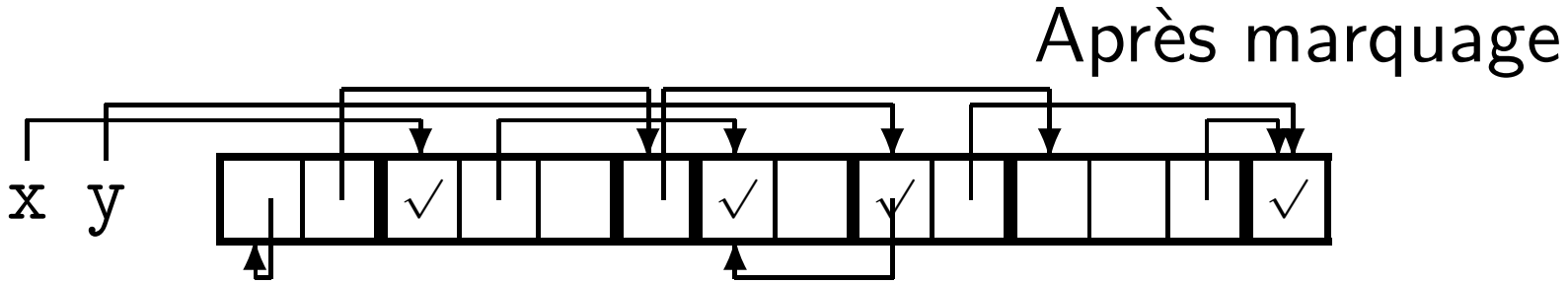
GC “Mark-and-compact”

Exemple:



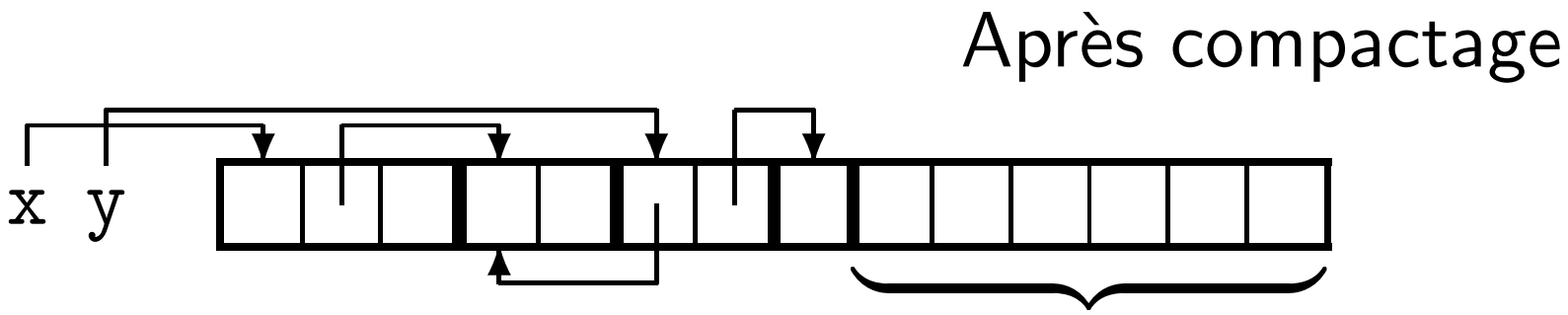
GC “Mark-and-compact”

Exemple:



GC “Mark-and-compact”

Exemple:



GC “Mark-and-compact”

Avantages et inconvénients:

+ Il élimine la fragmentation.

– Sa phase de compactage est plus coûteuse que la phase de balayage du GC “Mark-and-sweep” (par un facteur constant).

– Elle est aussi plus complexe.

– La mobilité des objets pose des contraintes supplémentaires sur l'application. En effet, la référence à un objet doit être considérée comme invalide chaque fois qu'une opération pouvant déclencher le GC est effectuée.

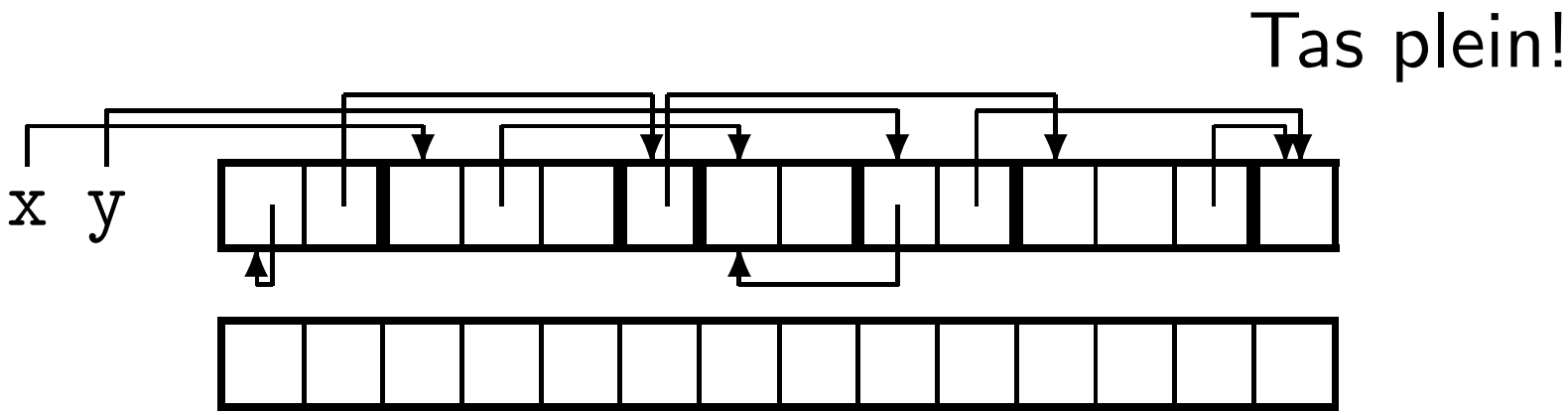
GC "Stop-and-copy"

Description:

- L'espace disponible pour le tas est divisé en deux semi-espaces.
- Lorsque le GC n'est pas actif, un seul des deux semi-espaces est utilisé.
- Un cycle de GC est déclenché lorsque le semi-espace courant est trop plein.
- Lors d'un cycle de GC, on nomme le semi-espace trop plein le "*from-space*" et l'autre semi-espace, le "*to-space*". Leur nom vient du fait que les objets vivants sont transférés du "from-space" vers le "to-space".
- Par une traversée de graphe à partir des racines, les objets vivants sont marqués et copiés vers le bas du "to-space". Les références sont mises à jour au cours de la traversée elle-même.
- Lorsque la traversée est terminée, tous les objets vivants ont été copiés dans le bas du "to-space", le reste du "to-space" contient de l'espace libre et, enfin, le contenu du "from-space" peut être oublié.
- La mise à jour des références se fait grâce à l'installation de "*forwarding-pointers*" dans les anciennes copies des objets vivants. Ainsi, lorsqu'on souhaite mettre à jour une référence vers un objet déjà copié, on n'a qu'à suivre le "forwarding-pointer" pour trouver la nouvelle copie.
- Après le cycle de GC, c'est le "to-space" qui devient le seul semi-espace utilisé.

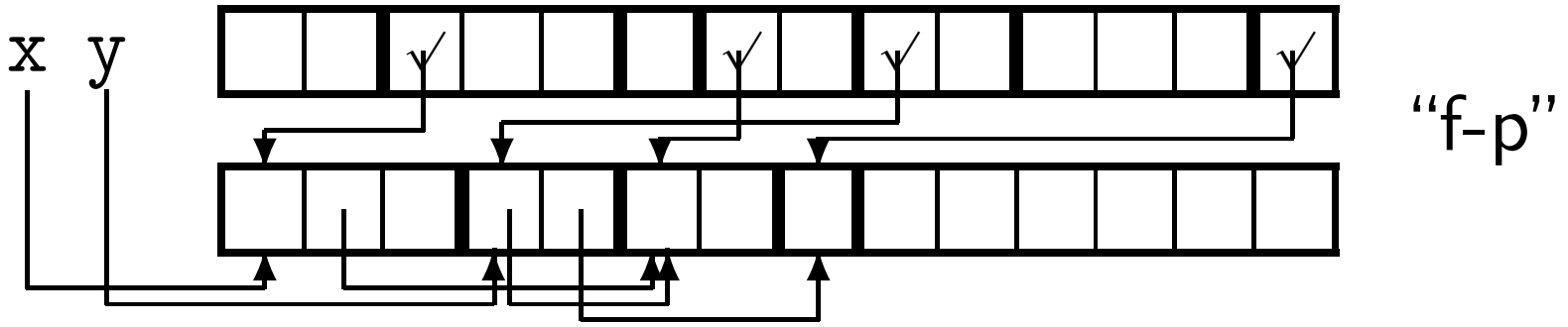
GC “Stop-and-copy”

Exemple:



GC “Stop-and-copy”

Exemple:



GC “Stop-and-copy”

Avantages et inconvénients:

+ Ce GC possède les mêmes avantages que le GC “Mark-and-compact”.

+ Un cycle du GC “Stop-and-copy” prend un temps proportionnel à la quantité d'objets vivants (au lieu de la taille du tas).

– Il a des besoins accrus en mémoire.

Récapitulation

Les GC “Mark-and-sweep”, “Mark-and-compact” et “Stop-and-copy” ont certains points en commun:

- Plus on utilise un tas de grande taille, plus le GC est efficace (en autant qu'on ignore les effets de cache et de mémoire virtuelle).
- En fait, un cycle de GC touche à presque toute la mémoire occupée par le tas. Donc, l'utilisation d'un tas de trop grande taille n'est pas favorable pour la cache.
- Quand un cycle de GC est déclenché, l'application est bloquée.

Afin de répondre aux premier et deuxième points, les GC *générationnels* ont été créés. Afin de répondre au troisième point, les GC *incrémentiels* et *en temps-réel* ont été créés.

GC générationnel

Description:

- Un GC générationnel est une variante d'un GC conventionnel ("Mark-and-sweep", "Mark-and-compact" ou "Stop-and-copy") où le tas est divisé en *générations*. Habituellement, deux générations sont employées, la *jeune* et la *vieille*, mais il peut y avoir autant de générations qu'on veut (N).
- Dans le cas $N = 2$, on distingue les GC *partiels* et les GC *globaux*.
- Lors d'un GC partiel, seuls les objets morts de la jeune génération sont ramassés. Lorsqu'un objet a survécu à k GC partiels, on le considère comme étant mature et on le transfère dans la vieille génération. On appelle ce transfert la *promotion* de l'objet.
- Les GC partiels sont priviligiés en autant qu'il reste assez de place dans la vieille génération pour accommoder toutes les éventuelles promotions. La vieille génération est normalement beaucoup plus grande que la jeune génération.
- Typiquement, la plupart des objets meurent très jeunes. Donc, il y a peu de promotions et les GC globaux sont rares. Aussi, les GC partiels, bien que fréquents, sont rapides et favorables à la cache.
- Dans les cas $N > 2$, on a diverses tailles de générations et divers critères de promotion. De tels GC sont généralement d'une plus grande efficacité.
- Des mécanismes additionnels doivent être prévus afin de tenir compte des références allant de la vieille génération vers la jeune. On appelle de telles références des références *inversées*. La difficulté vient du fait qu'un GC partiel ne marque que les objets de la jeune génération et qu'un jeune objet référencé uniquement depuis la vieille génération semblerait mort.

GC incrémentiel

Description:

- Il s'agit d'une technique orthogonale aux techniques précédentes qui consiste à séparer un cycle de GC en plusieurs étapes, appelées *incréments*, afin que chaque pause dans l'exécution de l'application soit plus courte.
- Ceci améliore la continuité dans l'exécution de l'application.
- Le design d'un GC incrémentiel demande beaucoup de soin car la partie "utile" de l'application, appelée *mutateur*, et le GC ont tendance à se nuire l'un l'autre. Exemples de problèmes potentiels:
  - Si le mutateur demande une allocation entre deux incréments d'un même cycle de GC, le gestionnaire de la mémoire est-il apte à répondre à la demande?
  - Si le mutateur modifie le graphe des objets accessibles grâce à des effets de bord, le GC risque-t-il d'être confondu?
  - Si le GC laisse le tas dans un état temporairement instable en redonnant le contrôle au mutateur, que se passe-t-il?

  Des mécanismes doivent être prévus afin de s'assurer que le mutateur fontionne toujours correctement et que le GC soit capable de terminer chaque cycle avant qu'il ne manque d'espace libre pour le mutateur.

GC en temps-réel

Description:

- Il s'agit d'un GC incrémentiel dont les pauses, en pire cas et sur un matériel donné, sont suffisamment courtes et suffisamment espacées pour permettre au mutateur de fonctionner avec une fluidité d'exécution qui respecte scrupuleusement un standard donné.
- Exemple d'application du GC temps-réel: intégration dans les logiciels contrôlant un bras robotique à haute précision.

