Gestion Mémoire

Vincent Archambault-B
IFT 2035 - Université de Montréal

Pour obtenir le code source de ce document

- https://github.com/archambaultv/IFT2035-UdeM
- vincent.archambault-bouffard@umontreal.ca

Plan du cours

- Pointeurs
- Pile, tas et zone statique
- Gestion mémoire manuelle
- Gestion mémoire automatique

Pointeurs

Pointeurs

- En C, les pointeurs permettent d’avoir un accès direct à la mémoire

```c
#include<stdio.h>

int a = 4;
int* ptrA = &a;

int main(){
  printf("adresse de a : %p\n", ptrA);
  return 0;
}
```

Code C

Pointeurs

pointeur NULL	Valeur spéciale qui ne pointe sur rien
pointeur fou	L’objet pointé n’existe plus
fuite	L’objet pointé n’est plus nécessaire
normal	L’objet pointé existe et est nécessaire

Pointeur NULL

- Souvent utilisé pour un pointeur déclaré mais pas encore utilisé

```
#include<stdio.h>

int* ptrA = NULL;

int main(){
  printf("adresse de NULL : %p\n”,
           ptrA); // Vaut 0x0
  return 0;
}
```

Pointeur NULL

Pointeur NULL

- Utilisé si une fonction qui doit retourner un pointeur échoue

```
#include<stdio.h>
#include<stdlib.h>

int main(){
  int* ptrA = malloc(sizeof(int));

  if (ptrA == NULL)
    {
      printf("Plus de mémoire");
    }
  else
    {
      printf("adresse : %p\n", ptrA);
    }
}
```

Pointeur NULL

Pointeur fou

- Danger de réécrire par dessus d'autres données

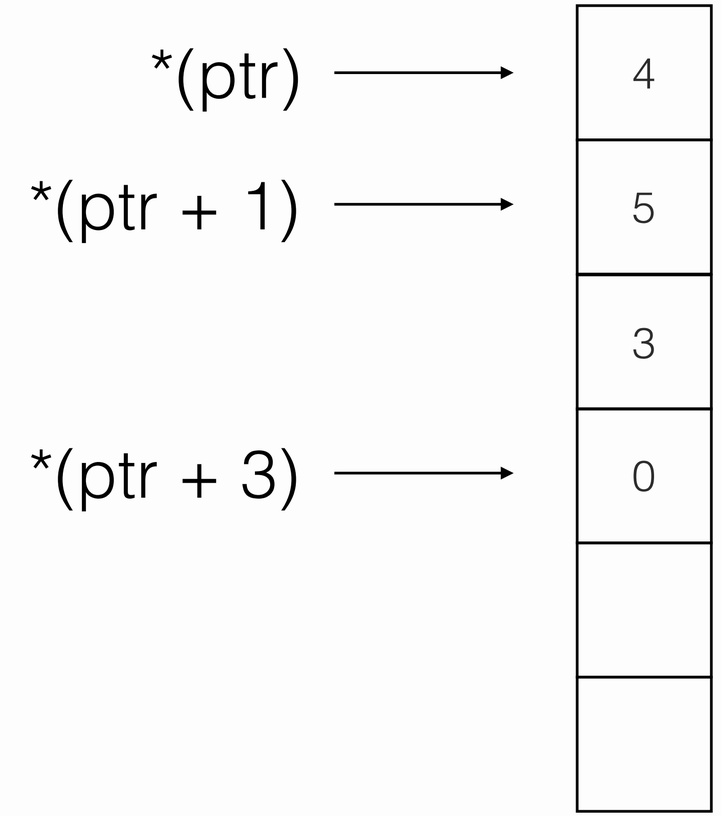
```c
#include<stdio.h>
#include<stdlib.h>

int main(){
  int* ptrA = malloc(sizeof(int));
  if (ptrA == NULL) return 1;
  free(ptrA); // ptrA est fou

  int* ptrB = malloc(sizeof(int));
  if (ptrB == NULL) return 1;
  *ptrB = 3;
  *ptrA = 5; // ptrA fou écrase ptrB

  printf("Valeur : %i\n", *ptrB); // 5
}
```

Pointeur fou

Arithmétique de pointeurs

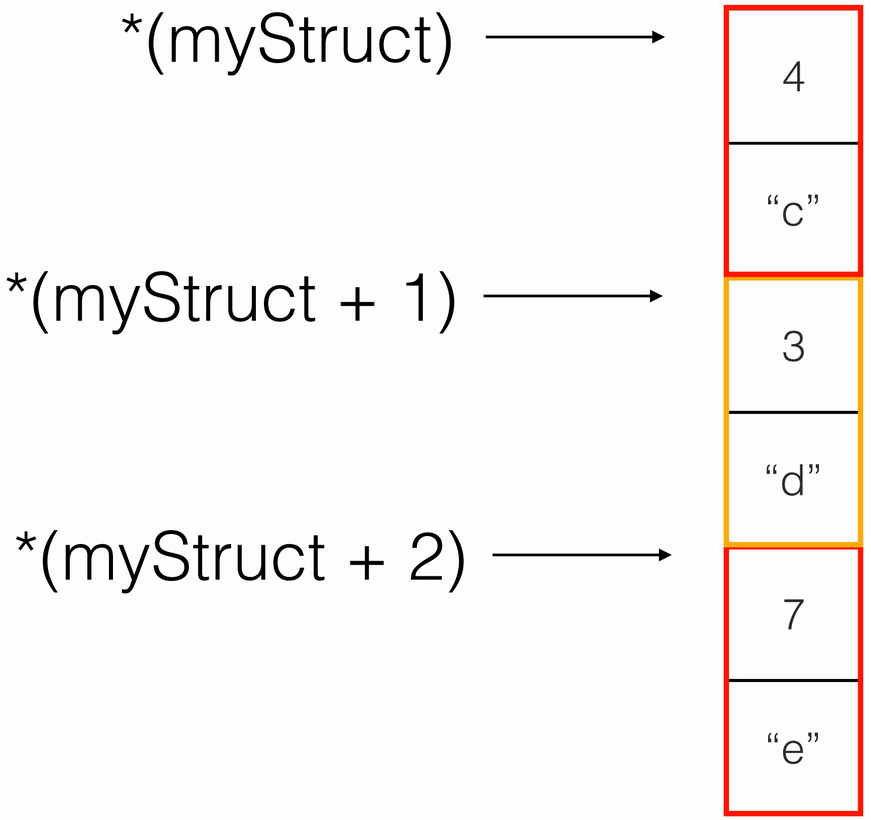
Arithmétique de pointeurs

- En C, il est possible d'additionner un pointeur et un entier
- Cela donne une nouvelle adresse mémoire



Arithmétique de pointeurs

- Pour un pointeur *int, +1 équivaut à additionner 1 fois sizeof(int)
- Pour un pointeur *myStruct, +1 équivaut à additionner 1 fois sizeof(myStruct)



Tableaux

- L’arithmétique de pointeur est utilisée pour créer des tableaux
  - x[n] = ∗(x+n)
- Si le tableau x à une taille 10, x[11] est indéfini

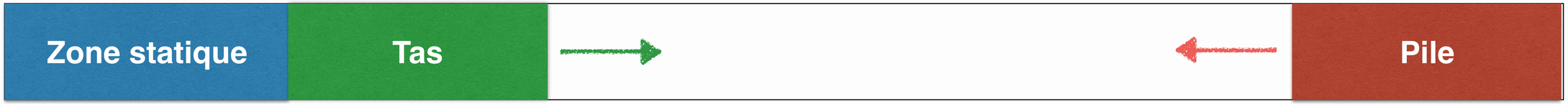
Pile, tas et zone statique

Pile, tas et zone statique

zone statique Emplacement mémoire où se trouvent le code et les constantes

pile Emplacement mémoire où se trouvent les paramètres des fonctions et variables locales

tas Autres allocations dynamiques



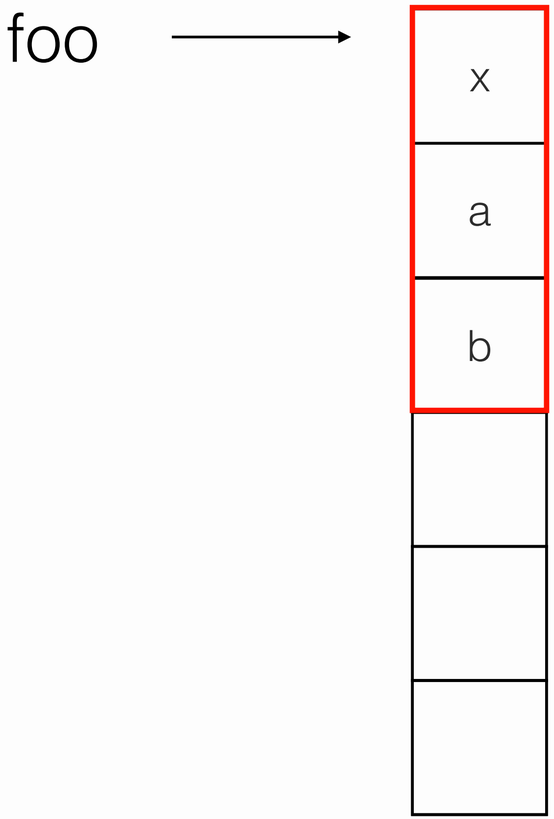
Pile

- Les fonctions ont un nombre connu de paramètres (lors de l'appel)
- Les fonctions ont un nombre connu de variables locales
- L'ordre d'appel des fonctions n'est pas connu

- On peut donc empiler les appels un par-dessus les autres (blocs d'activations)

Pile

```
int times2(int x){
  return x * x;
}

int foo(int x){
  int a = times2(3);
  int b = times2(7);
  return a + b + x;
}
```

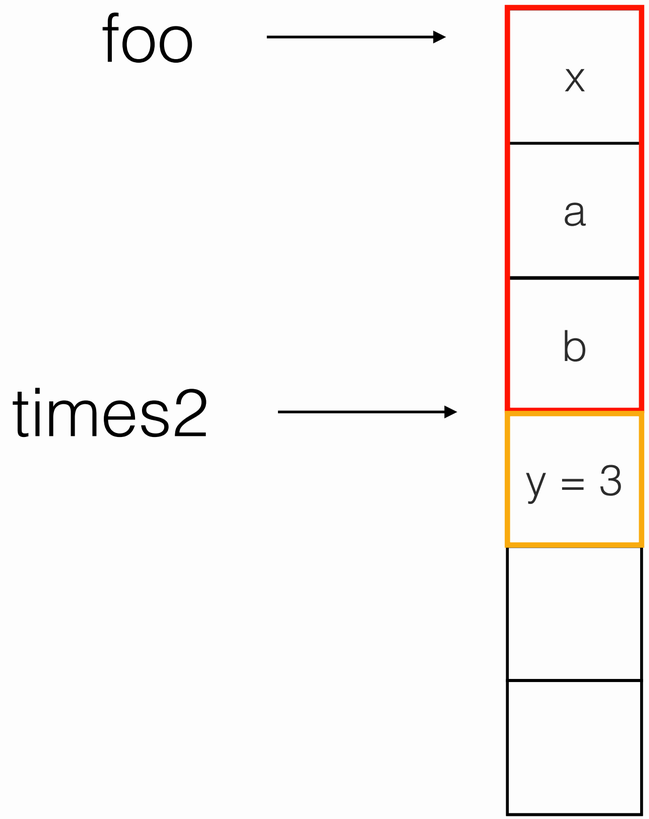


Code C

Pile

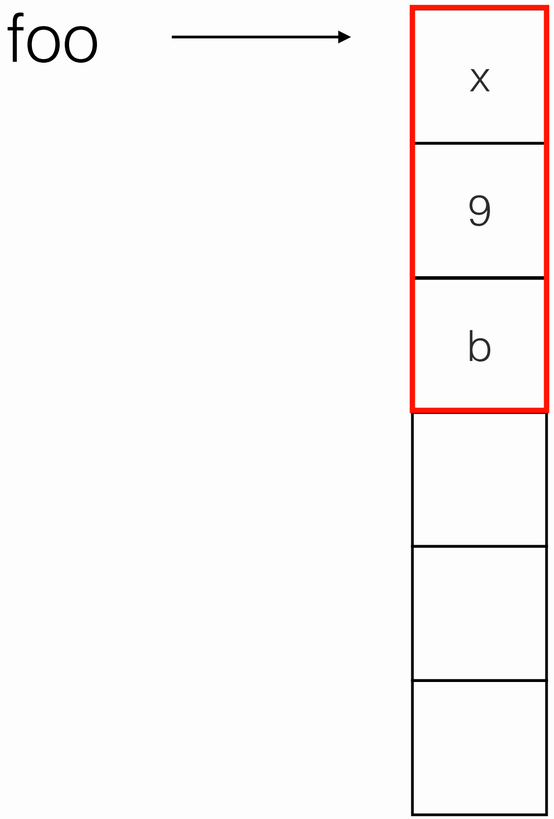
```
int times2(int y){
  return y * y;
}

int foo(int x){
  int a = times2(3);
  int b = times2(7);
  return a + b + x;
}
```

Code C



Pile

```
int times2(int x){
  return x * x;
}

int foo(int x){
  int a = times2(3);
  int b = times2(7);
  return a + b + x;
}
```

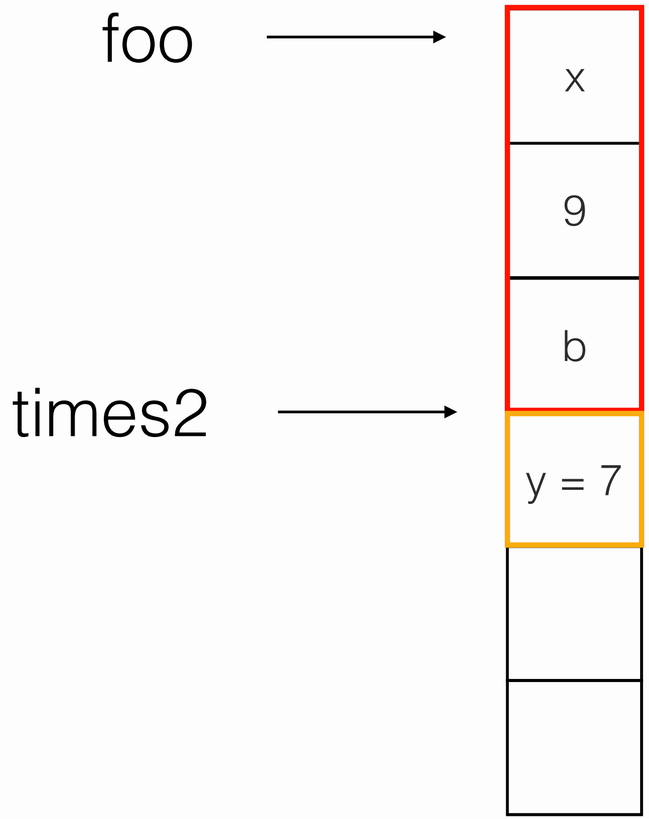


Code C

Pile

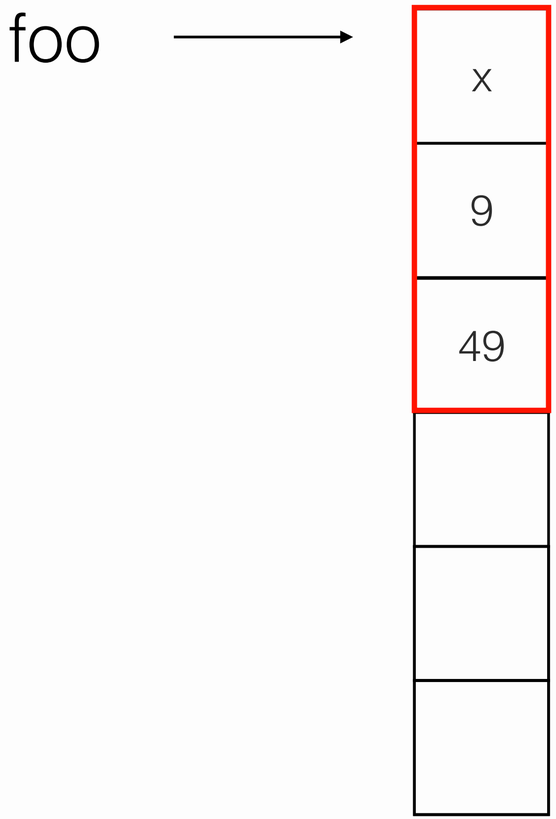
```
int times2(int x){
  return x * x;
}

int foo(int x){
  int a = times2(3);
  int b = times2(7);
  return a + b + x;
}
```

Code C



Pile

```
int times2(int x){
  return x * x;
}

int foo(int x){
  int a = times2(3);
  int b = times2(7);
  return a + b + x;
}
```

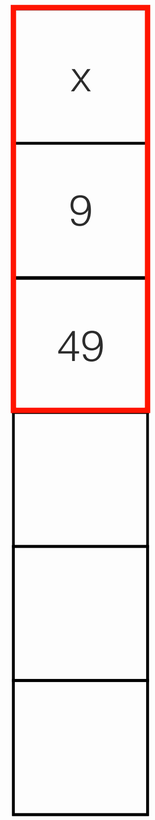


Code C

Pile

```
int times2(int x){
  return x * x;
}

int foo(int x){
  int a = times2(3);
  int b = times2(7);
  return a + b + x;
}
```

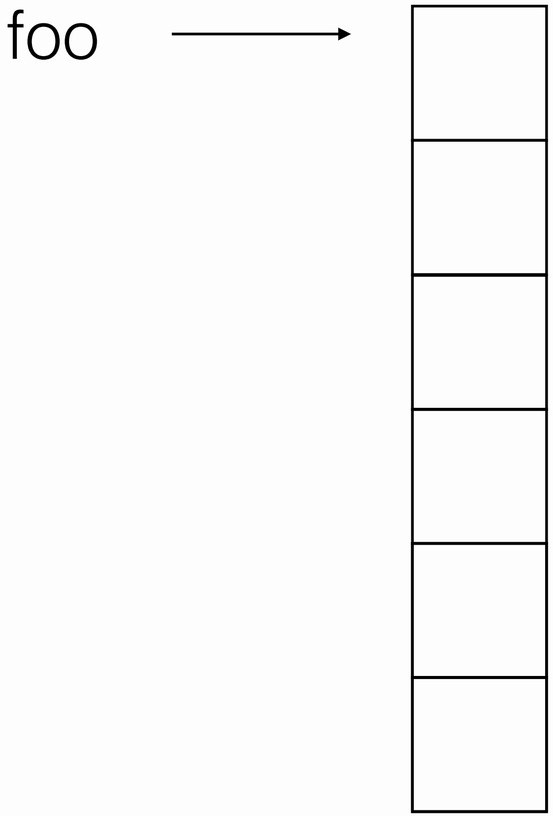
Résultat dans un registre (rax)



Code C

Pile

```
int times2(int x){
  return x * x;
}

int foo(int x){
  int a = times2(3);
  int b = times2(7);
  return a + b + x;
}
```



Code C

Pile

- Le schéma précédent fonctionne bien en C
  - Il n'y a pas de fermeture
  - Une fonction appelée est seulement active durant le bloc d'activation de l'appelant
- De façon générale, le bloc d'activation contient aussi l'adresse de retour

Tas

- Permets l'allocation dynamique de la mémoire
- En C, il faut utiliser malloc et free
- Toujours utiliser sizeof pour calculer la taille

```c
#include<stdlib.h>

int main(){
  int* ptrA = malloc(sizeof(int));
  if (ptrA == NULL) return 1;
  free(ptrA);
  return 0;
}
```

Code C

Gestion manuelle

Gestion manuelle

- Le programmeur doit libérer lui-même la mémoire (à l'aide de free)
- Très grand contrôle sur l'utilisation de la mémoire, mais il est facile de commettre des erreurs

Gestion automatique

Comptage de références

- À chaque objet est associé un compteur qui indique combien de pointeurs existent. Lorsque le compteur passe à 0, on peut désallouer l’objet.

Comptage de références

```c
#include<stdlib.h>
#include<stdio.h>

int compteur = 0;

int* makeInt(){
  int* n = malloc (sizeof(int));
  compteur++;
  return n;
}

void copy(int* old, int* new){
  new = old;
  compteur++;
  return;
}
```

Code C

```c
void freeInt(int* p){
  compteur--;
  if (compteur == 0){ free(p);};
  return;
}

int main(){
  int* n = makeInt();
  printf("Compteur : %i\n", compteur);
  int* p;
  copy(n, p);
  printf("Compteur : %i\n", compteur);
  freeInt(n);
  printf("Compteur : %i\n", compteur);
  freeInt(p);
  printf("Compteur : %i\n", compteur);
  return 0;
}
```

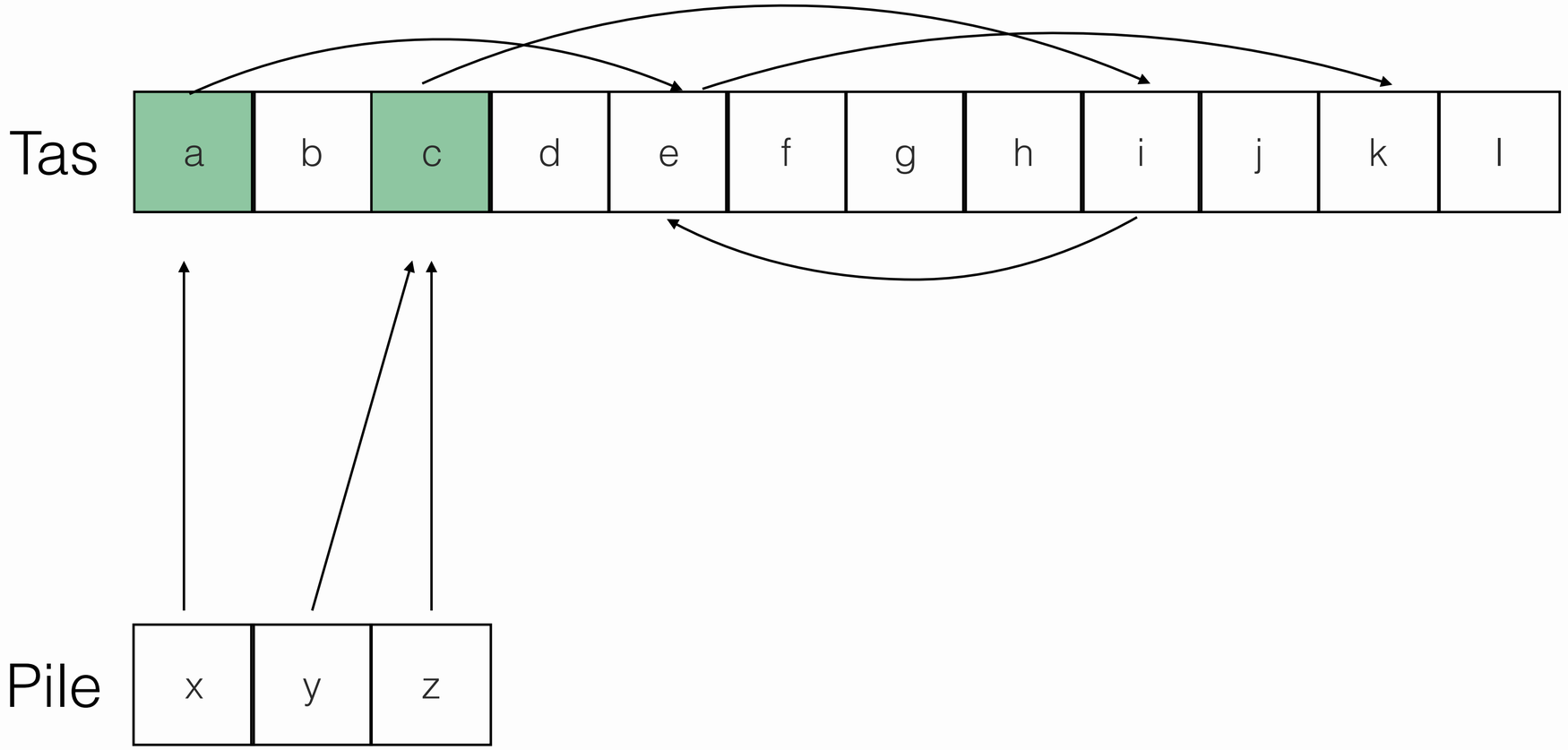
suite …

Garbage Collector

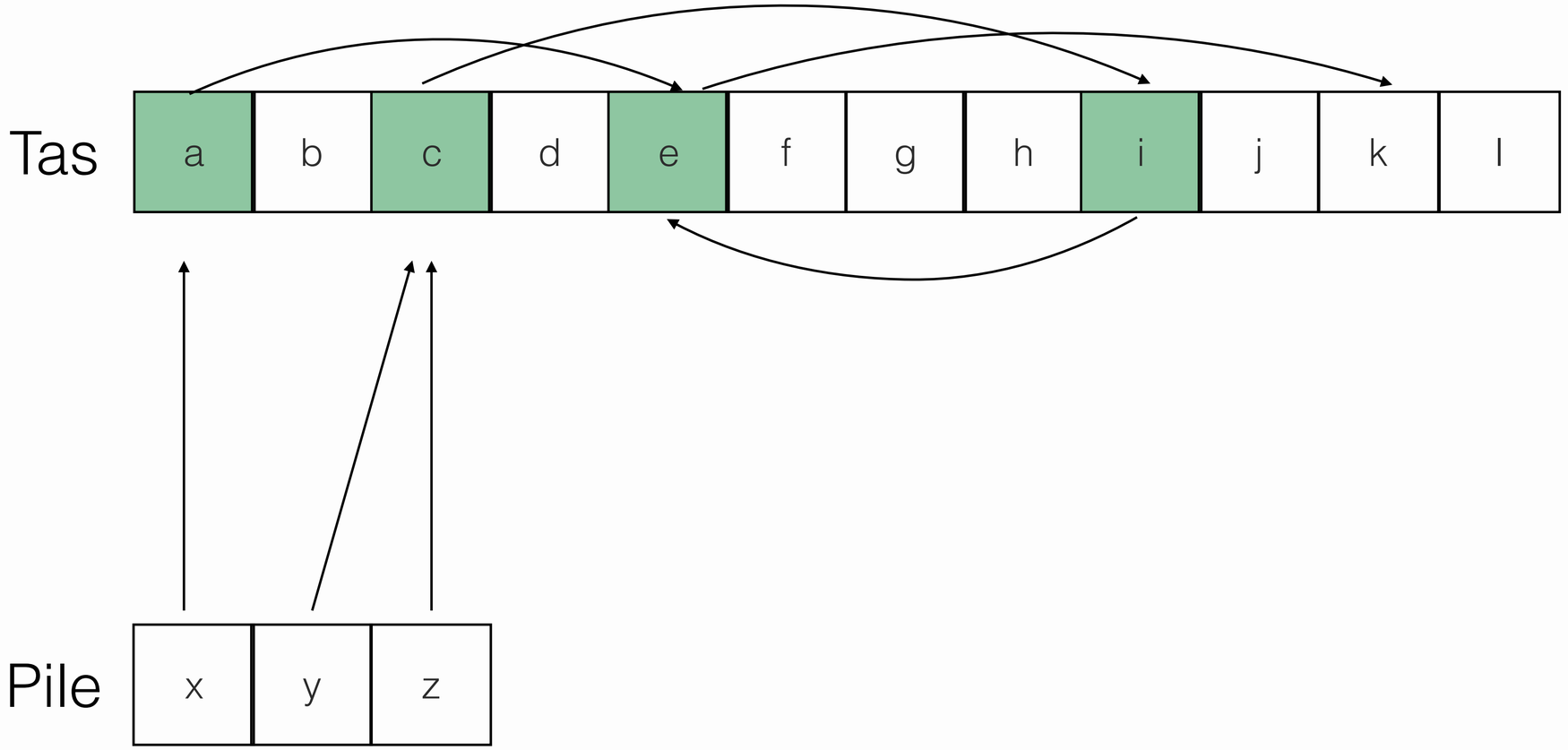
- À partir des objets globaux et sur la pile (les racines), traverser toutes les structures de données. Les objets non atteints peuvent être désalloués.

Mark & Sweep

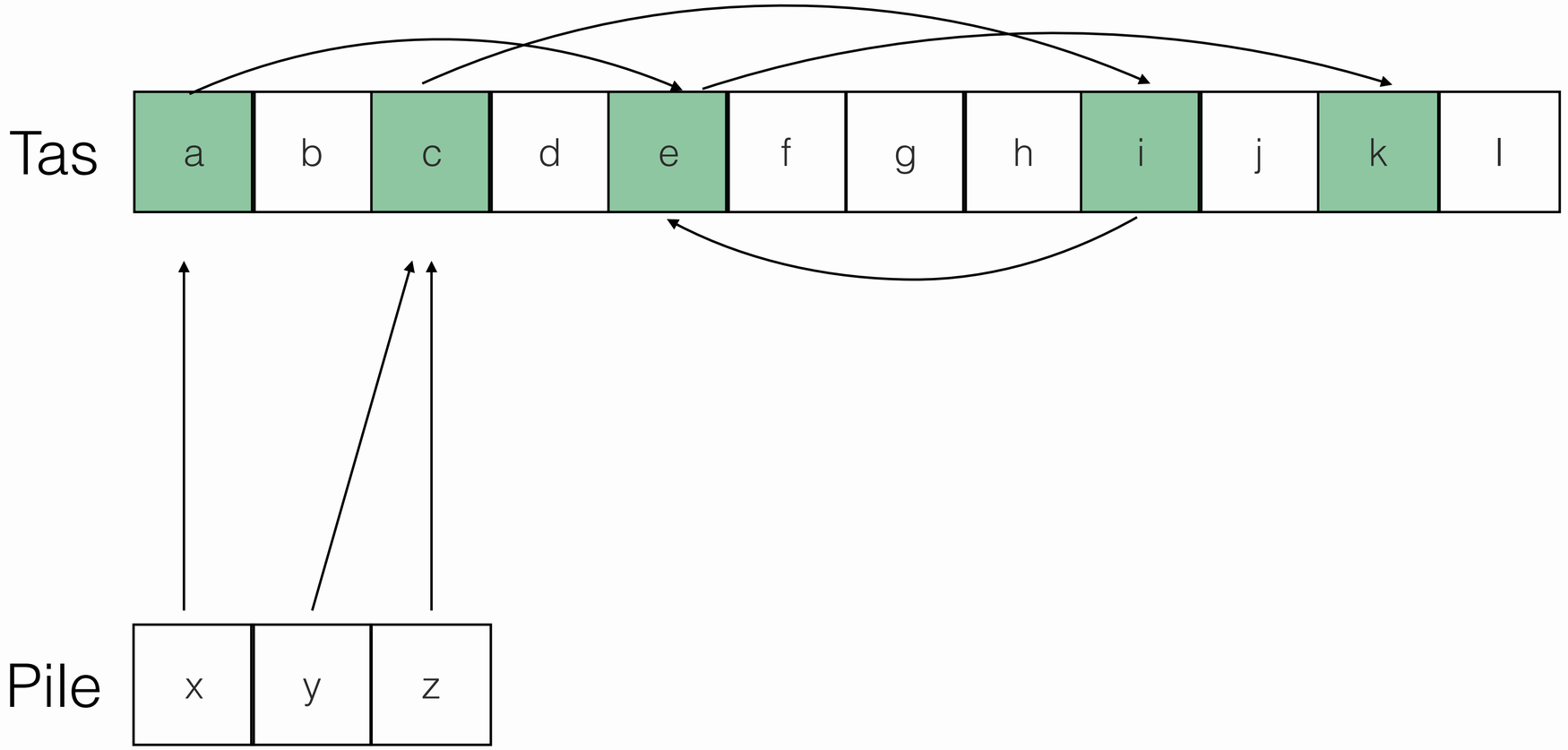
Mark & Sweep - Marquer depuis les racines



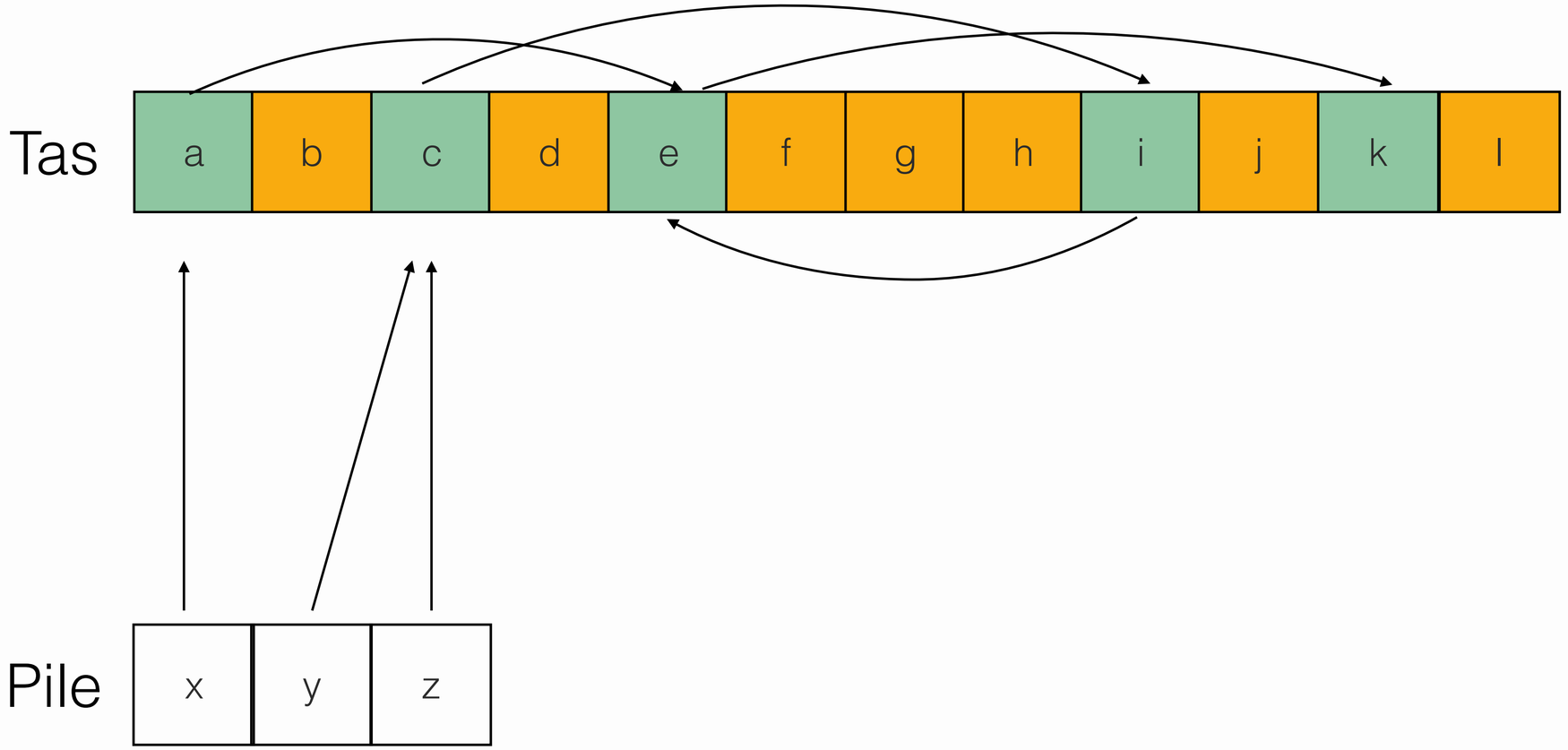
Mark & Sweep - Marquer les objets pointés



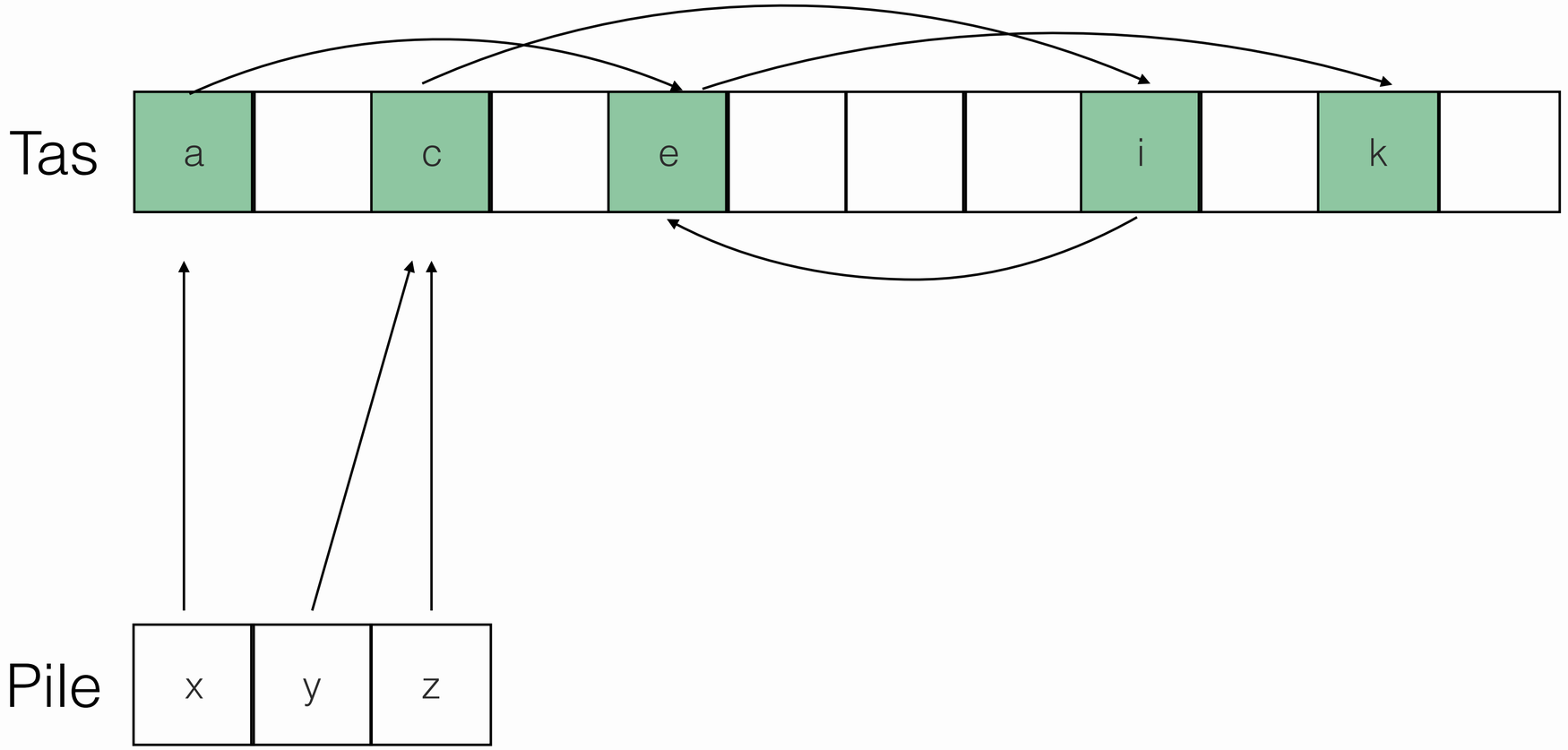
Mark & Sweep - Marquer les objets pointés



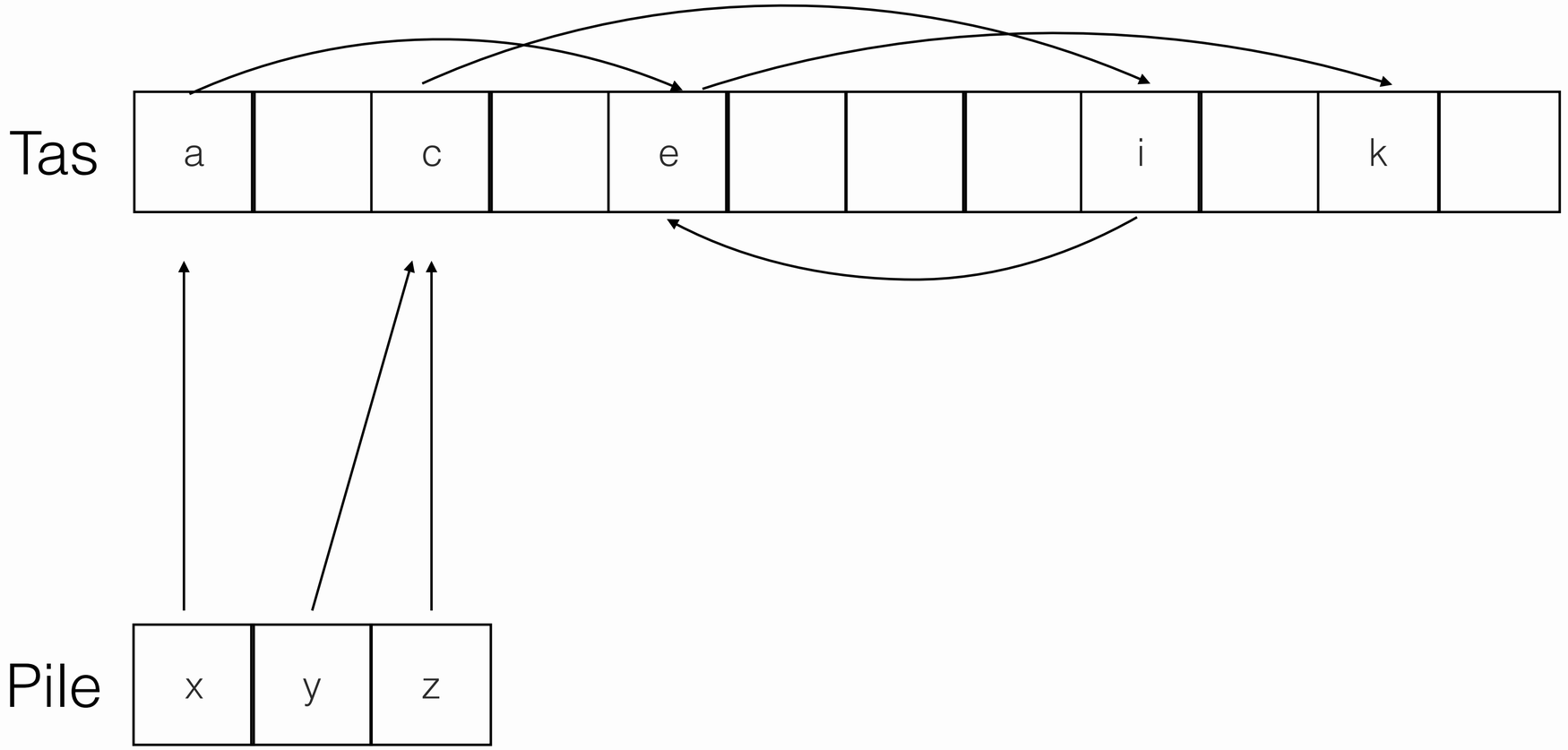
Mark & Sweep - Libérer les objets non marqués



Mark & Sweep - Libérer les objets non marqués



Mark & Sweep - Remettre le marquage à zéro



Mark & Sweep - Pseudocode

```
marknSweep(){
   roots = getRoots();
   for root in roots
     mark(root);
   sweep();
}
```

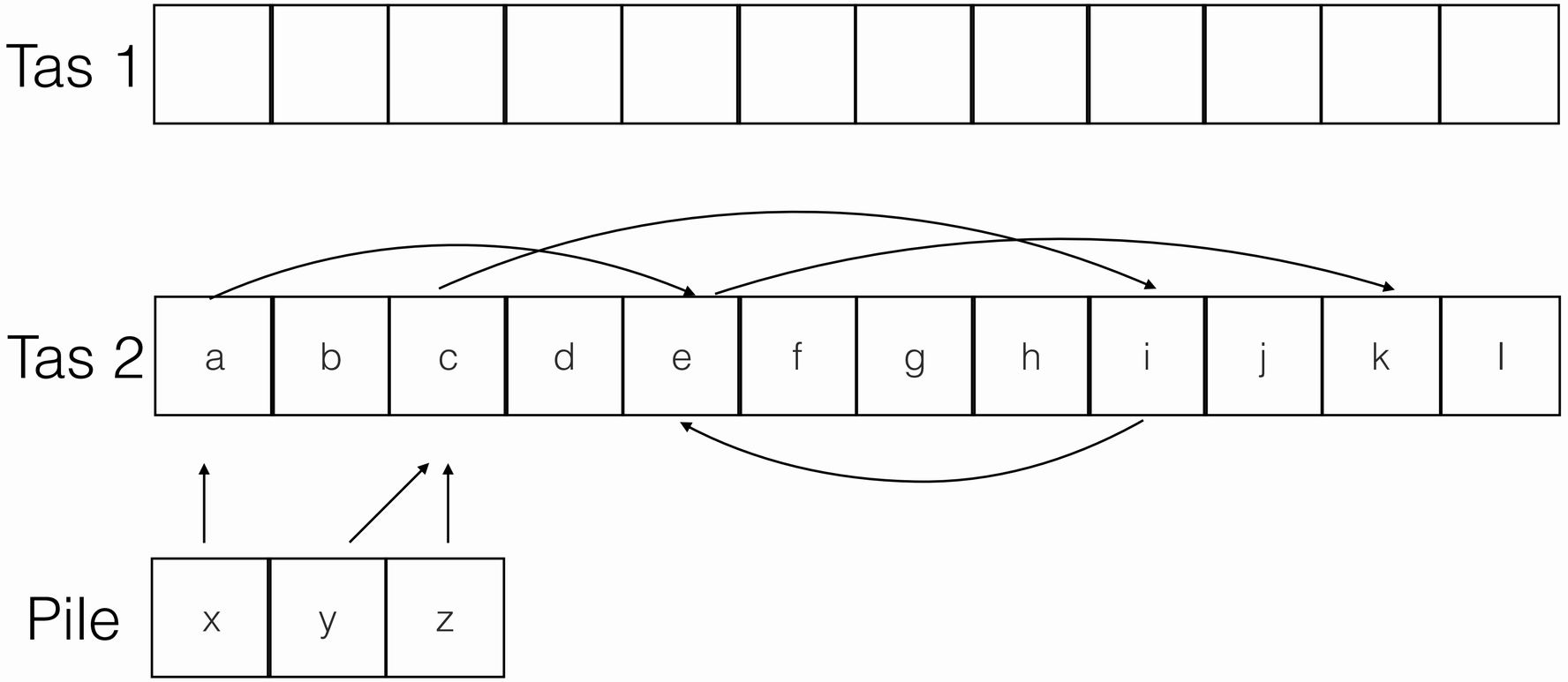
Pseudocode

```
mark(obj){
  if (obj->marked) return;
  obj->marked = True;
  for c in obj.children
     mark(c);
}

sweep(){
   h = heapStartPointer;
   do {
     if (h->marked)
        h->marked = False;
     else
        free(h);
   } while (ptr = next_object(h))
}
```

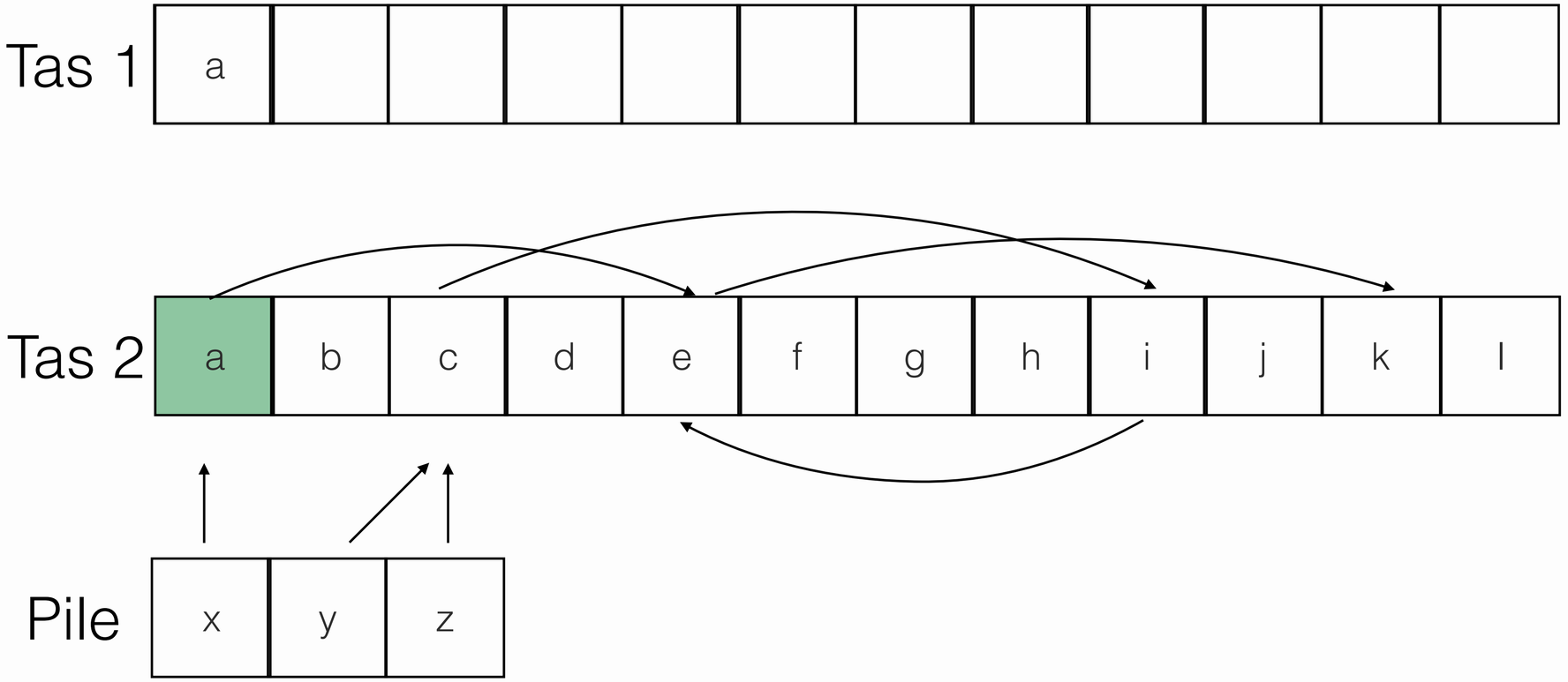
Pseudocode

Stop & Copy

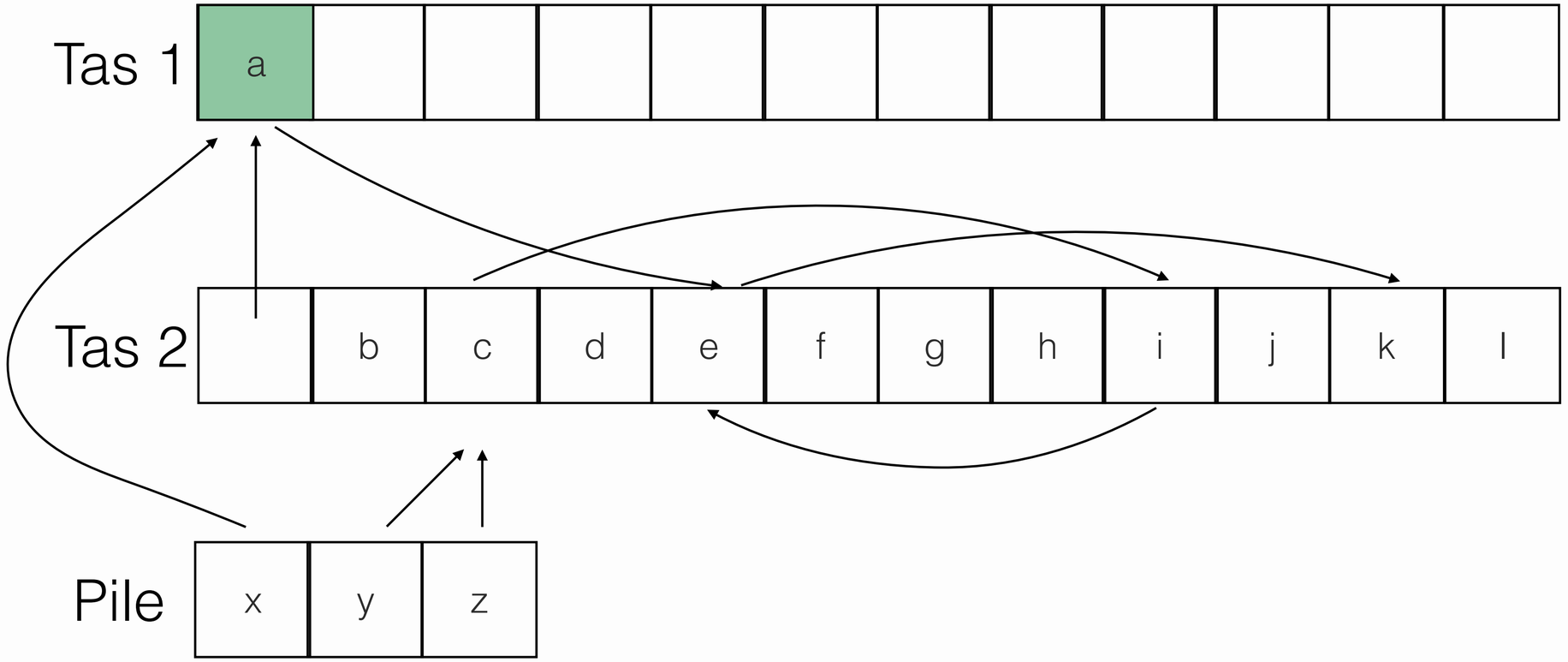
Stop & Copy - Arrêter le programme



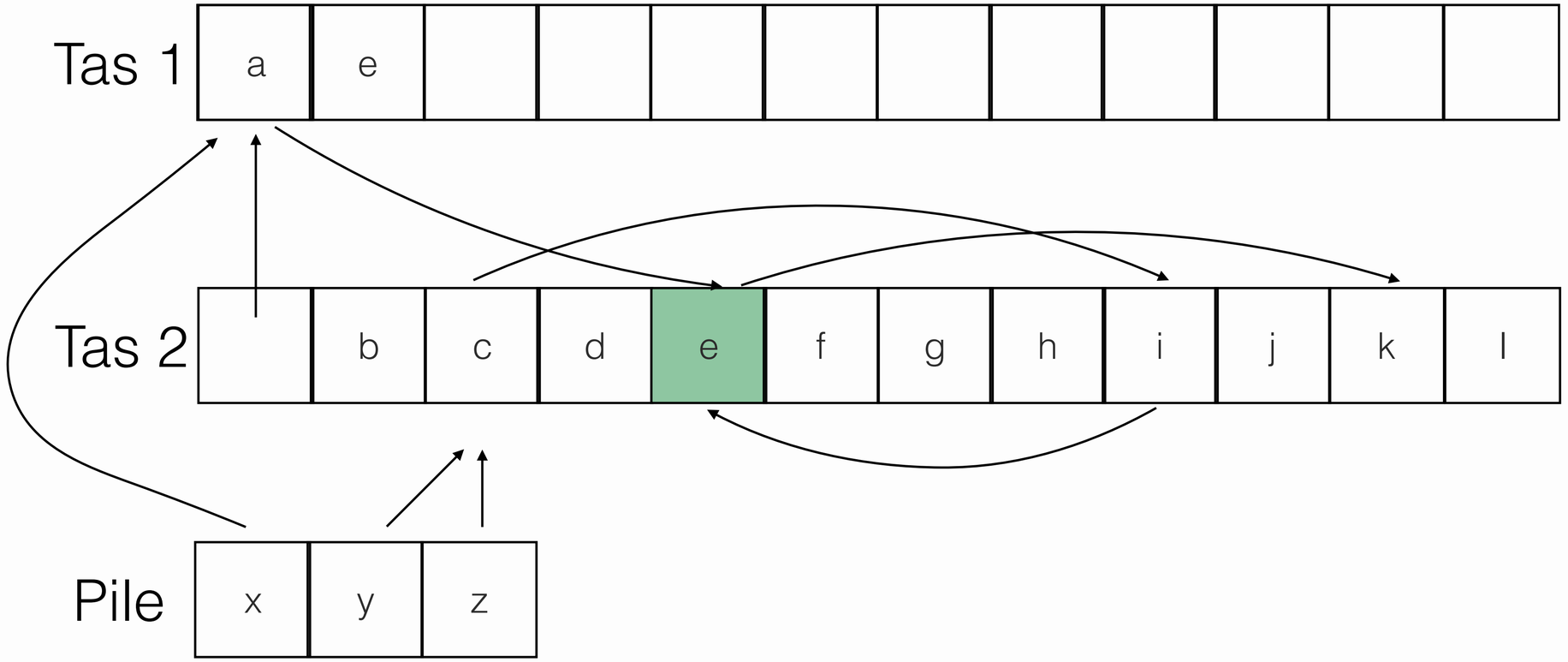
Stop & Copy - Copier un élément accessible depuis la racine



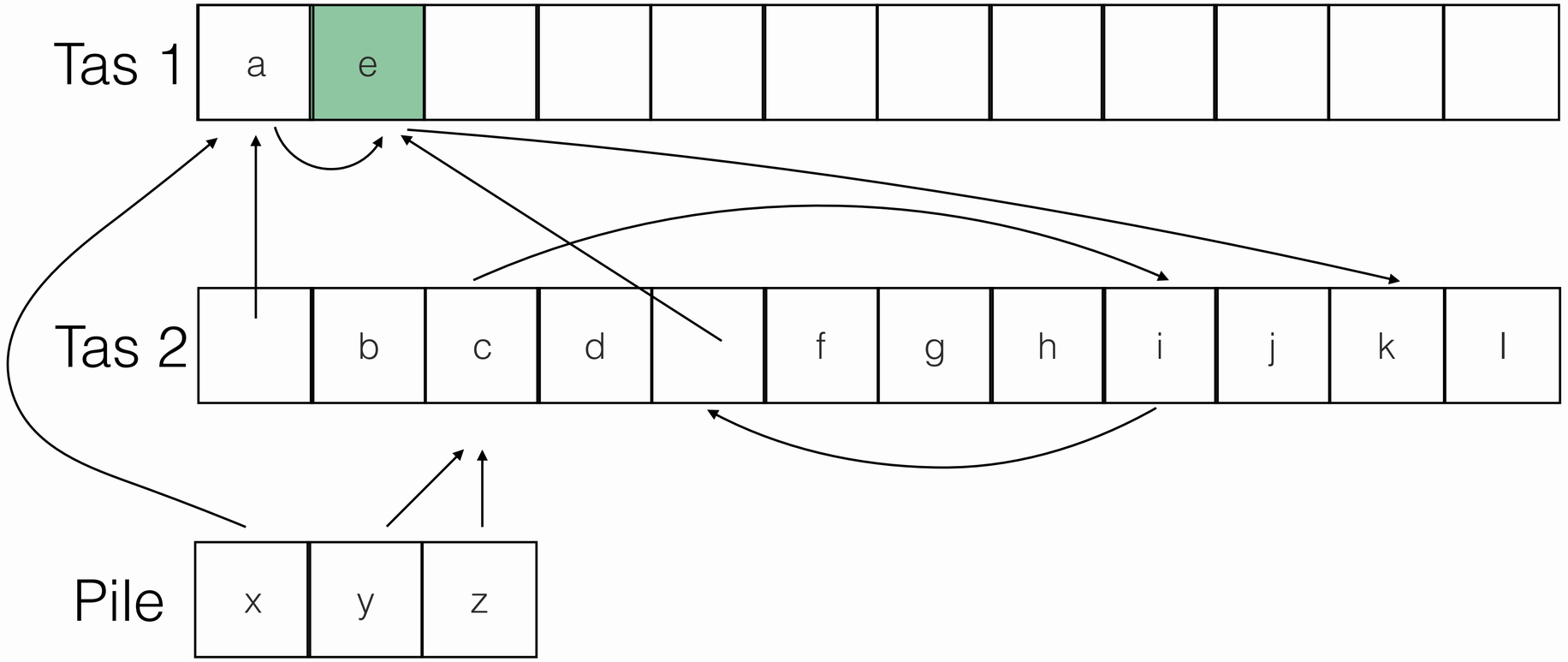
Stop & Copy - Copier un élément accessible depuis la racine



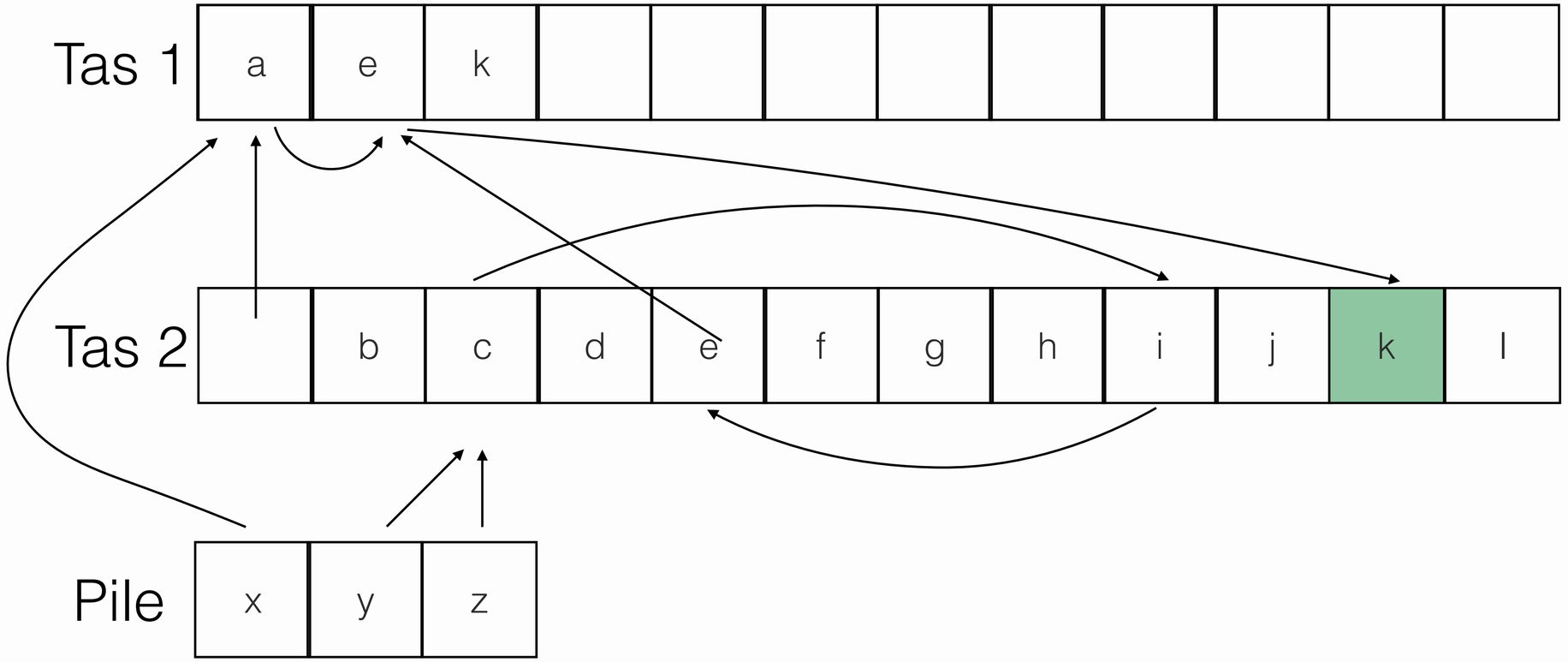
Stop & Copy - Copier un élément accessible depuis l'élément copié



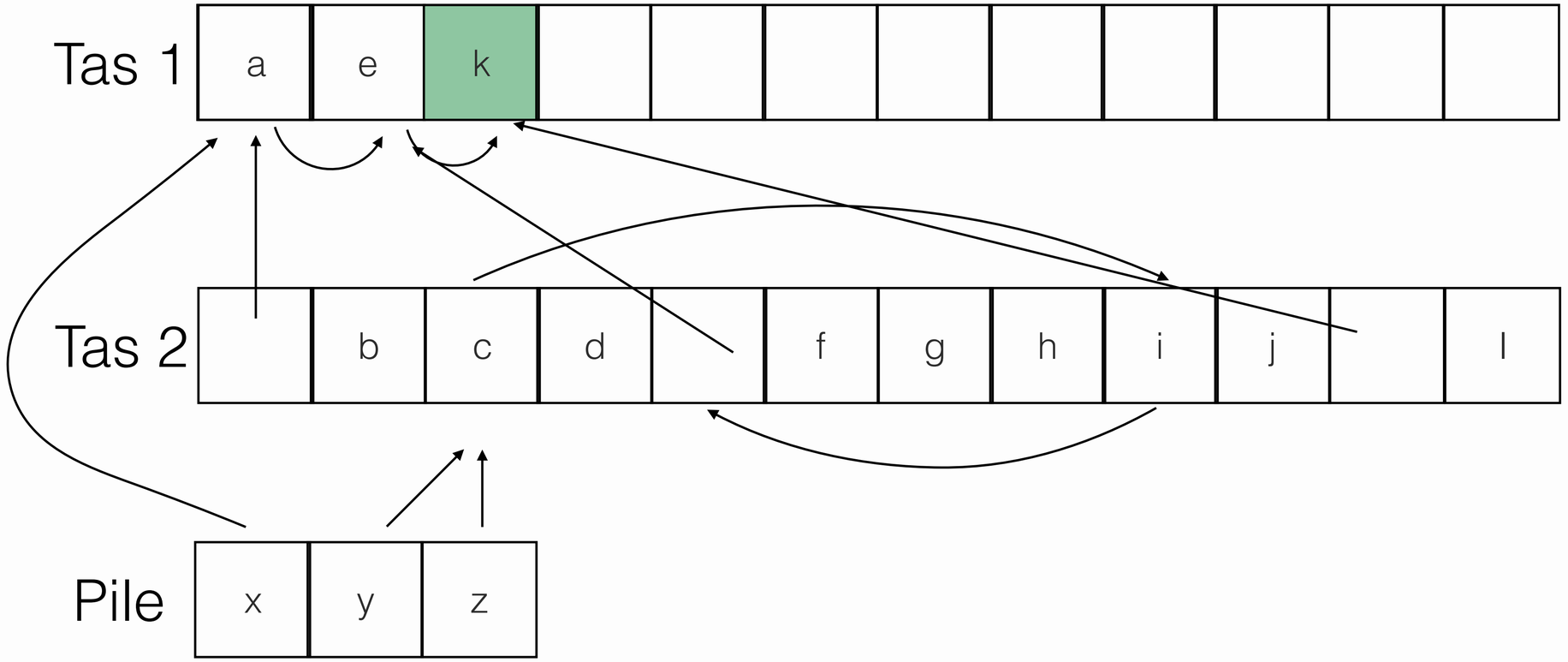
Stop & Copy - Copier un élément accessible depuis l'élément copié



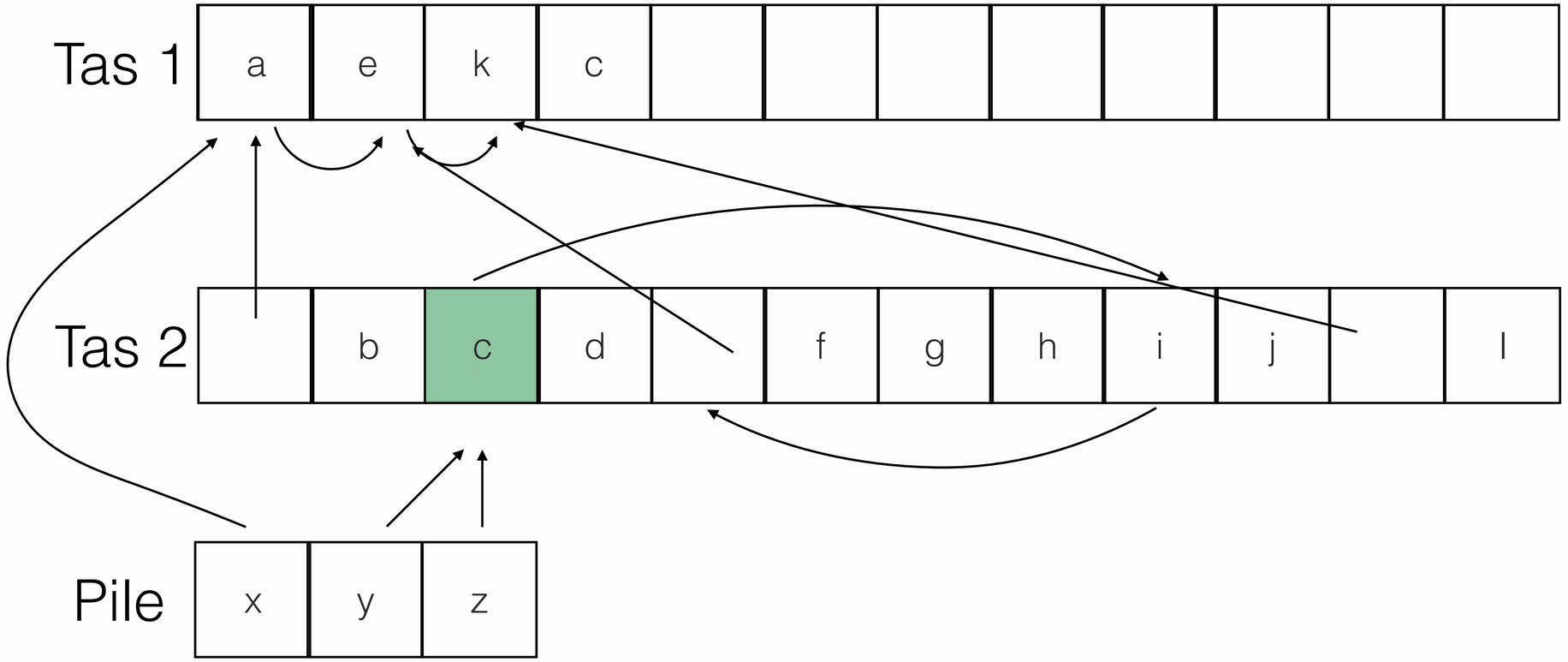
Stop & Copy - Copier un élément accessible depuis l'élément copié



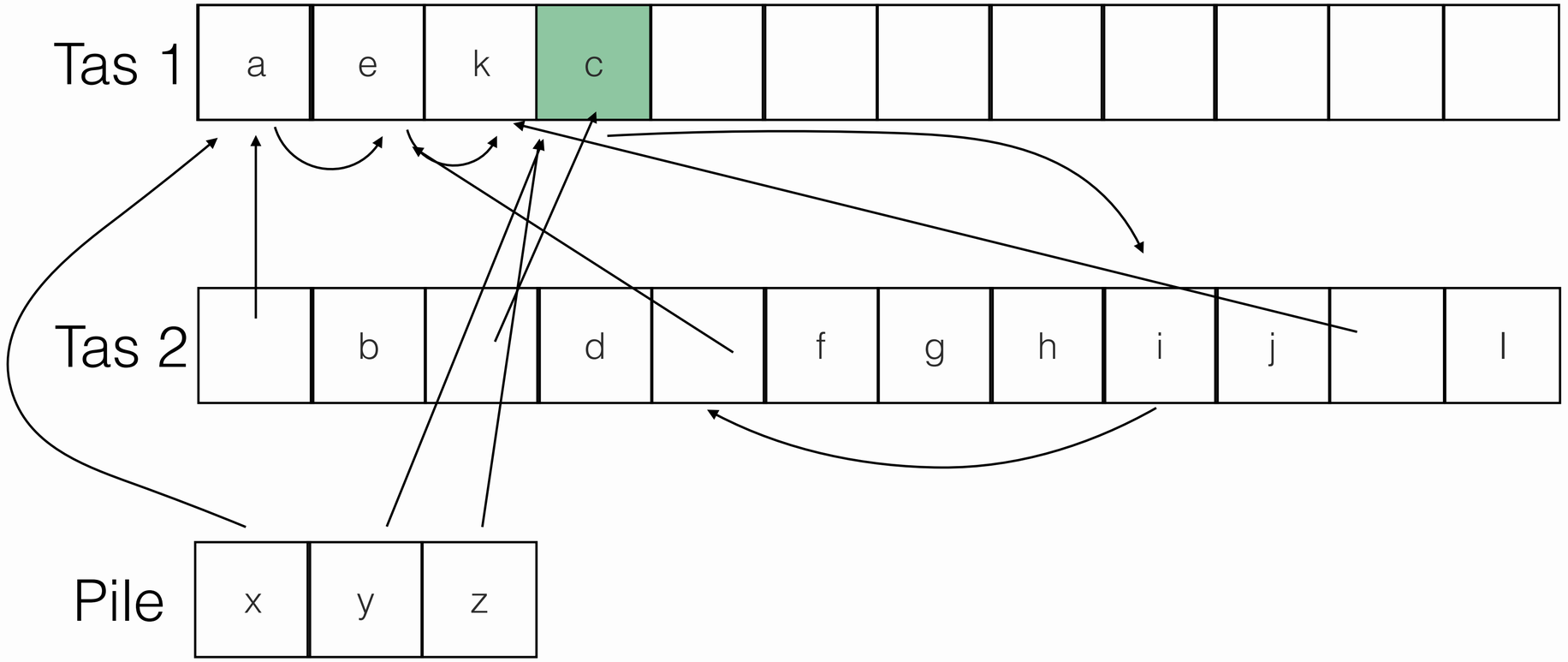
Stop & Copy - Copier un élément accessible depuis l'élément copié



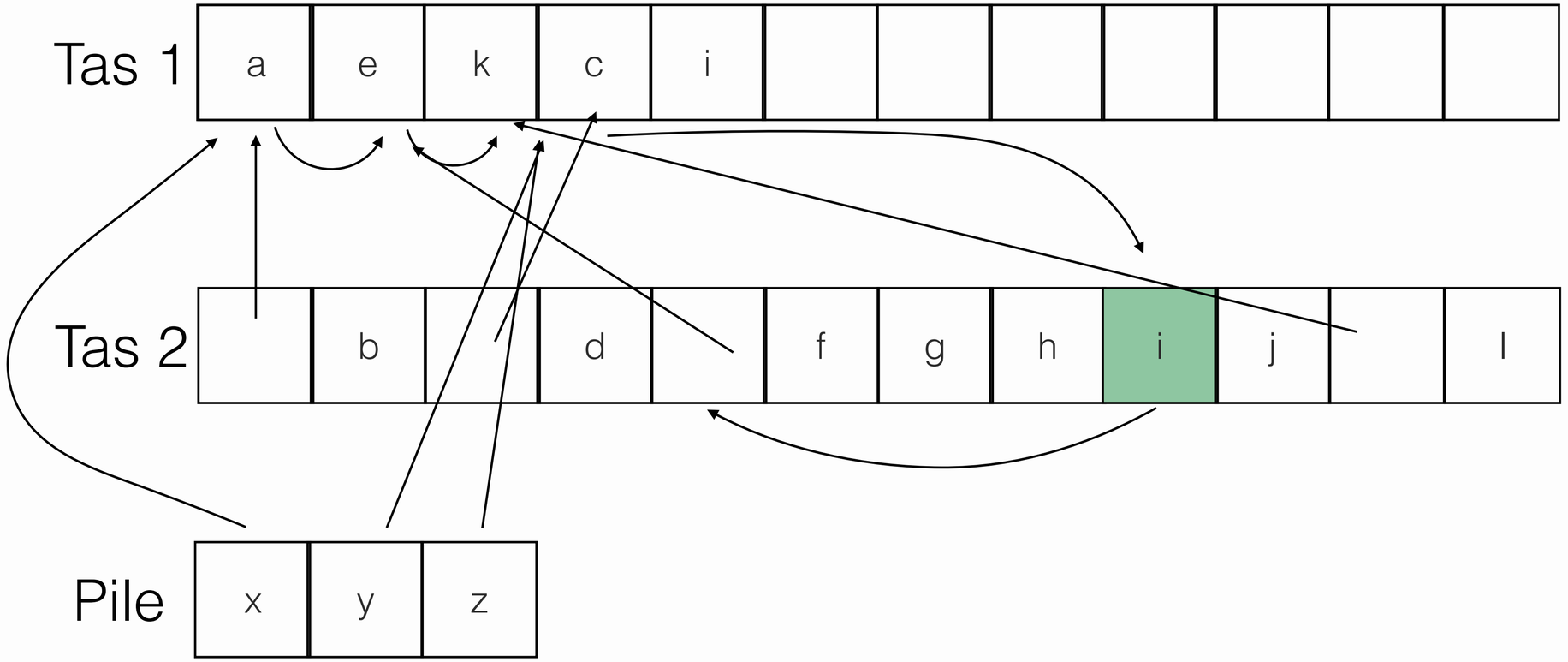
Stop & Copy - On recommence avec les autres racines



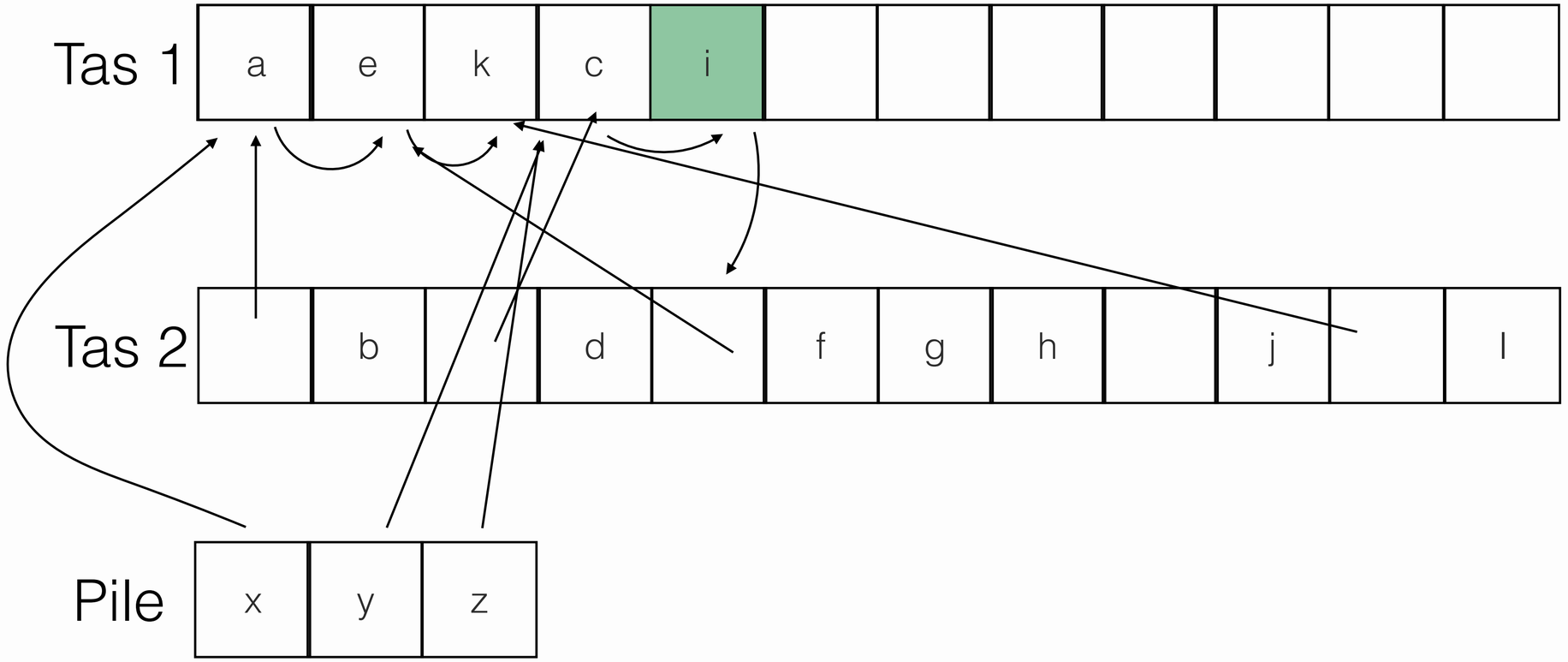
Stop & Copy - On recommence avec les autres racines



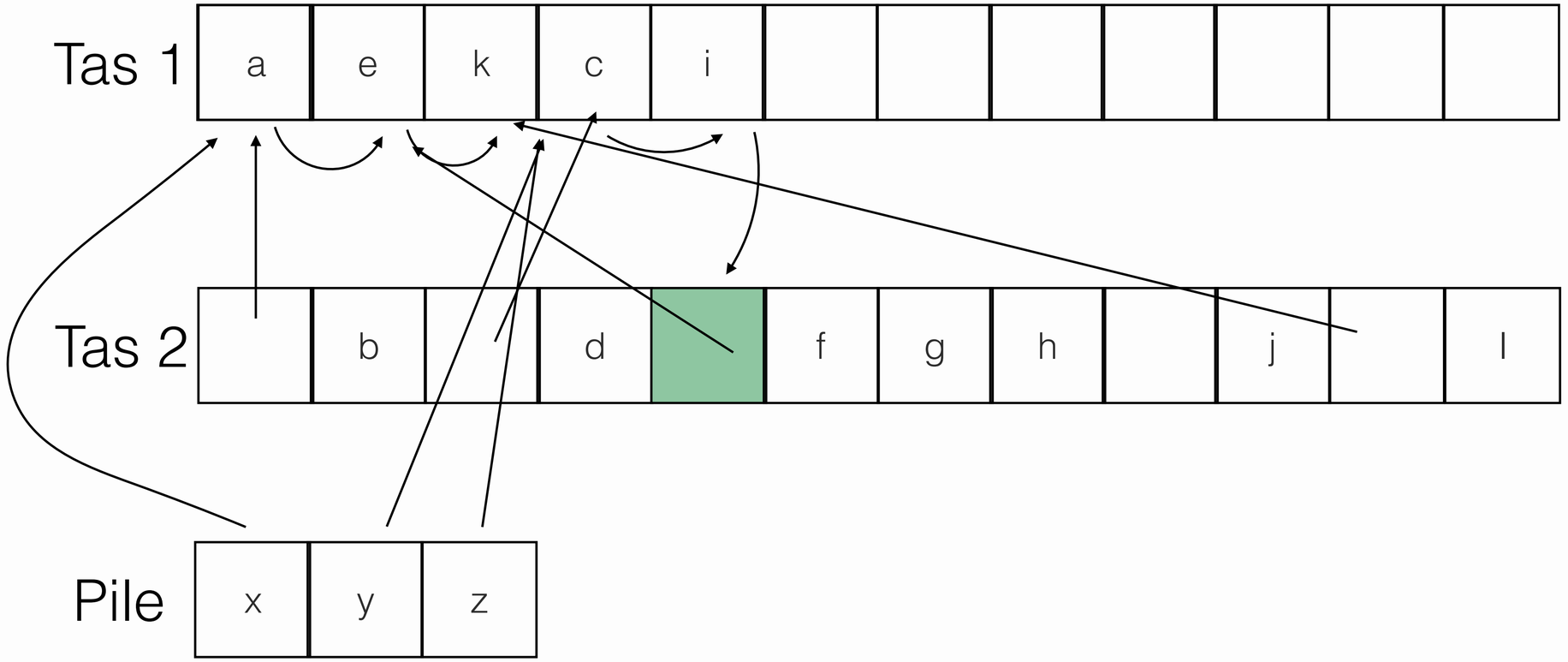
Stop & Copy - On recommence avec les autres racines



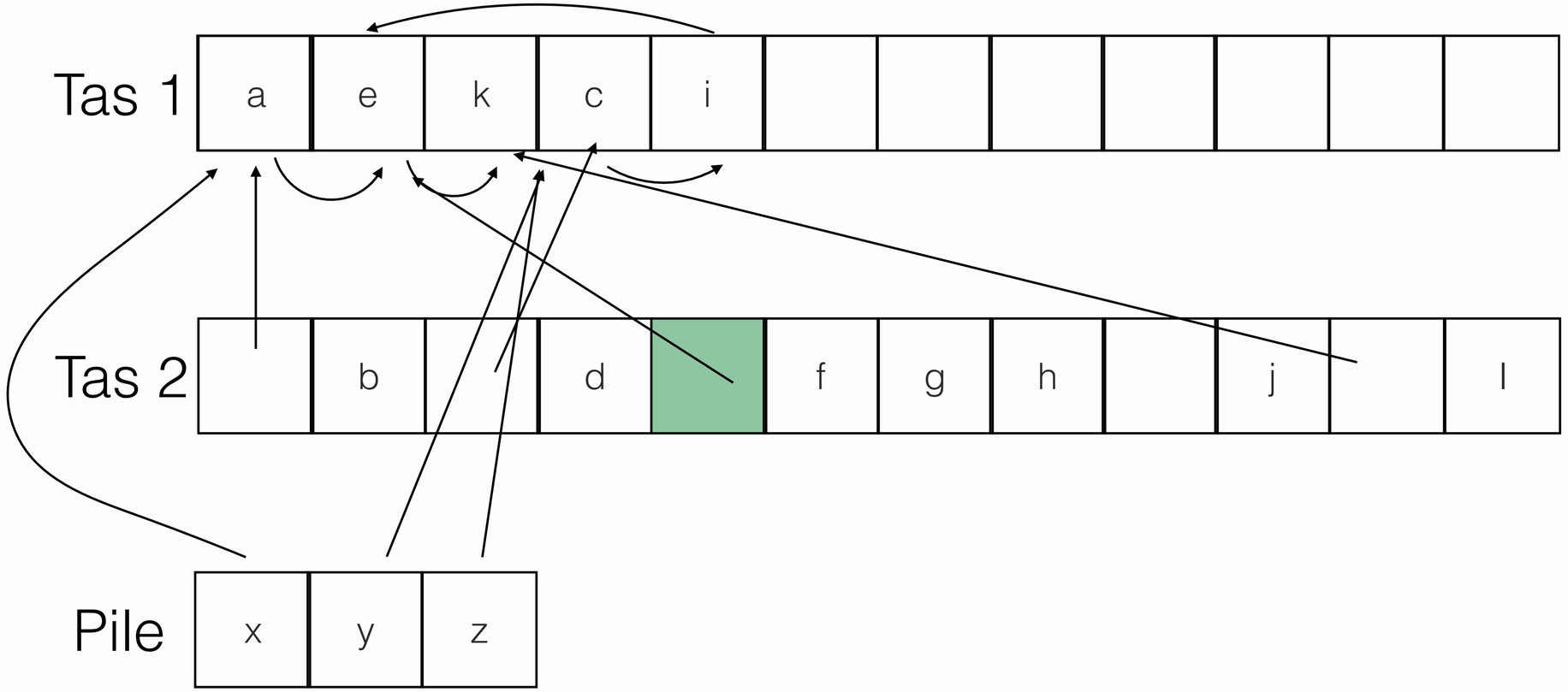
Stop & Copy - On recommence avec les autres racines



Stop & Copy - Lorsqu'on arrive vers un objet déjà copié, on met à jour l'adresse



Stop & Copy - Lorsqu'on arrive vers un objet déjà copié, on met à jour l'adresse



Stop & Copy - Maintenant Tas1 servira à allouer de la mémoire