

Introduction à l'informatique

pour les mathématiques, la physique et les sciences
computationnelles

Yann Thorimbert



**UNIVERSITÉ
DE GENÈVE**

CENTRE UNIVERSITAIRE
D'INFORMATIQUE

Partie 2 | Chapitre 2

Structures de données

Yann Thorimbert



UNIVERSITÉ
DE GENÈVE

CENTRE UNIVERSITAIRE
D'INFORMATIQUE



Chapitres du cours (seconde partie du cours)

0. Introduction à la complexité algorithmique

1. Algorithmes de tri naïfs

2. Structures de données : tableaux, listes et dictionnaires ←

3. Tri fusion et récursivité

4. Algorithmes de recherche au sein d'une séquence

5. Algorithmes sur graphes

Les types de données

- En programmation, on effectue un traitement sur des données de différentes natures.
- La nature de ces données, au sein d'un langage, est nommée **type**.
- Exemples quelconques (le nom du type dépend du langage) :
 - Entier naturel stocké sur 16 bits.
 - Entier relatif stocké sur 32 bits.
 - Nombre à virgule flottante stocké sur 32 bits.
 - Chaîne de caractères
 - ... et plein d'autres !

Types numériques

- Les types d'entiers et de flottants sont des types fondamentaux. Des circuits électroniques peuvent réaliser des opérations directement sur eux (exemple : additionneur).
- Python : `int`, `float`
(avec une couche logicielle qu'on ignore ici, cf. `bignums`)
- Matlab : `double`, `single`, `int8`, `int16`, ..., `int64`, `uint8`, ..., `uint64`.
- C : `char`, `short`, `int`, `long`, `float`, `double`

Mémoire allouée au processus

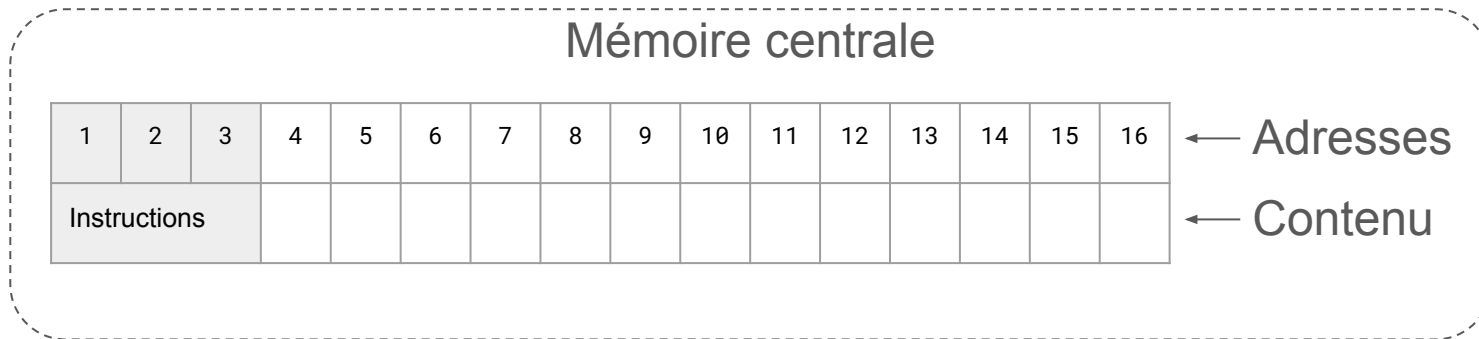
Quel est le lien entre :

- le code du programme,
- le type des variables impliquées,
- le stockage des séquences de bits correspondantes dans la mémoire ?

Mémoire allouée au processus | **Exemple simplifié**

Le processus commence : les instructions elles-mêmes occupent de la place en mémoire, mais une certaine marge est déjà réservée dans la MC pour d'autres données.

Pour des raisons de place, nous imaginons qu'il suffit de 3 octets pour stocker les instructions du programme.



Mémoire allouée au processus | **Exemple simplifié**

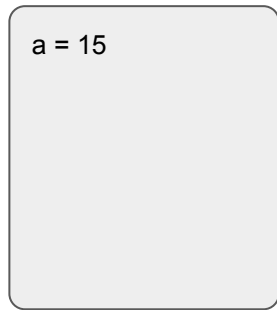
Un entier est déclaré (disons sur 2 octets pour l'exemple).

a = 15

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Instructions			15																			

Mémoire allouée au processus | Exemple simplifié

Un entier est déclaré (disons sur 2 octets pour l'exemple).



a

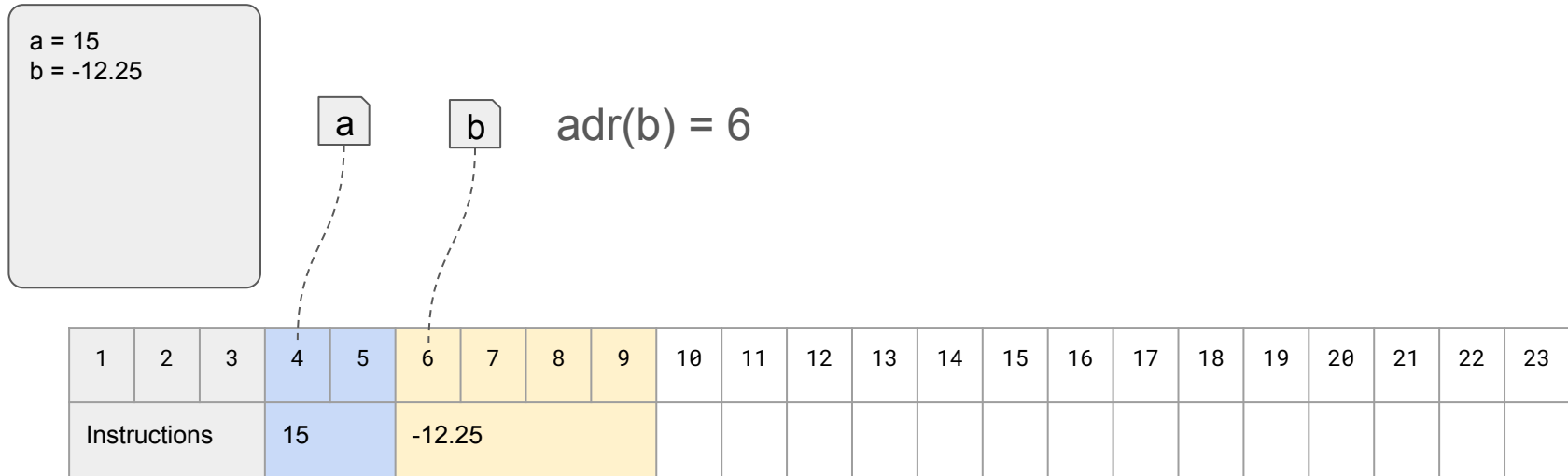
`adr(a) = 4`

Une **variable** désigne 1) un emplacement dans la mémoire et 2) un type de donnée.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Instructions			15																			

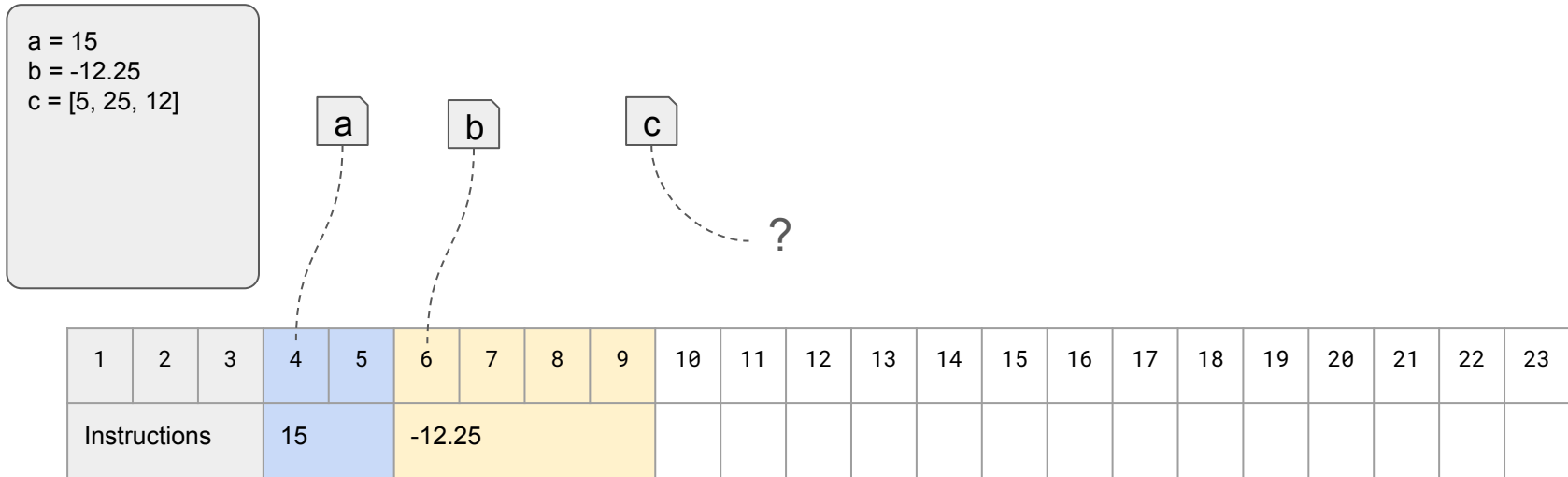
Mémoire allouée au processus | Exemple simplifié

Un float est déclaré (4 octets).



Mémoire allouée au processus | Exemple simplifié

Une **séquence** de 3 entiers est déclarée. À quoi cela correspond-il ?





Tableaux de valeurs

Tableaux de valeurs

- Rappel : dans la mémoire centrale, les données sont adressées de telle sorte que la donnée numéro n est voisine de la donnée $n + 1$.
- On peut exploiter cela pour créer un tableau de valeurs **contiguës** en mémoire.

Adresse	1	2	3	4	5	...
Contenu	10101111	00101101	10111100	00000000	10111100	...



Tableaux de valeurs

- Rappel : dans la mémoire centrale, les données sont adressées de telle sorte que la donnée numéro n est voisine de la donnée $n + 1$.
- On peut exploiter cela pour créer un tableau de valeurs **contiguës** en mémoire $\Rightarrow$ Dans les langages de bas niveau, on peut directement incrémenter l'adresse d'une variable pour accéder à sa voisine.
- Les `array` de MATLAB et les `list` de python exploitent cela.

Tableaux de valeurs

$a = [8, 12, 1, 7]$ (ici on suppose que chaque entier occupe 1 octet)

Adresse	...	155	156	157	158	159	160	...
Contenu	...	10001000	00001000	00001100	0000001	00000111	10111100	...

Tableaux de valeurs

$a = [8, 12, 1, 7]$

Première adresse du tableau (ici 156 par exemple)



Adresse	...	155	156	157	158	159	160	...
Contenu	...	10001000	00001000	00001100	00000001	00000111	10111100	...

Tableaux de valeurs

$a = [8, 12, 1, 7]$

Hors du tableau

Adresse	...	155	156	157	158	159	160	...
Contenu	...	10001000	00001000	00001100	00000001	00000111	10111100	...

Cf. code de démonstration

Tableaux de valeurs

$a = [8, 12, 1, 7]$

Adresse	...	155	156	157	158	159	160	...
Contenu	...	10001000	00001000	00001100	0000001	00000111	10111100	...

En réalité, les types entiers occupent souvent plus de 8 bits.
Dans ce cas, chaque élément du tableau occupe plusieurs adresses.



Dépassement de tableau : C vs Python

```
#include <stdio.h>
int main()
{
    int a[3] = {4, 7, 5};
    printf("%d\n", a[0]);
    printf("%d\n", a[1]);
    printf("%d\n", a[2]);
    printf("%d\n", a[3]);
    return 0;
}
```

Comportement indéfini

```
a = [4, 7, 5]
print(a[0])
print(a[1])
print(a[2])
print(a[3])
```

Erreur explicite à l'exécution

Tableaux de valeurs

- Si le type de chaque valeur stockée occupe k octets : $\text{adr}(i) = \text{adr}(i - 1) + k$.
Ou encore : $\text{adr}(i) = \text{adr}(0) + k \cdot i$.
- On doit donc mémoriser deux valeurs relatives au tableau lui-même, en plus des données : $\text{adr}(0)$ et k .
- Exemple où chaque valeur est codée sur 32 bits (4 octets) :
Ici, $\text{adr}(0) = 156$ et $k = 4$.

Adresse	...	156	157	158	159	160	161	162	163	...
Contenu	...	Première valeur				Seconde valeur				...



Tableaux de valeurs | **Note sur l'indexation**

- Dans certains langages comme MATLAB, le premier élément du tableau correspond à l'indice 1.
Dans ce cas, $\text{adr}(i) = \text{adr}(0) + k(i - 1)$.
- Dans d'autres langages comme C ou Python, le premier élément du tableau correspond à l'indice 0.
Dans ce cas, $\text{adr}(i) = \text{adr}(0) + k \cdot i$.

Tableaux de valeurs | **Taille fixe**

- Pour que le tableau de valeurs vu précédemment fonctionne, il faut qu'une suite d'adresses ait été **allouée pour lui en mémoire (par l'OS)**.
- Les adresses suivantes peuvent être utilisées par d'autres données du même processus, mais cela peut aussi être une partie de la mémoire dévolue à d'autres processus.
- Le tableau de valeurs vu jusqu'ici doit donc être de **taille fixe**, car s'il augmente de taille il empiète sur les données suivantes !
⇒ On doit mémoriser **trois** choses finalement : $\text{adr}(0)$, k et n .

Adresse du premier élément taille d'un élément nombre d'éléments



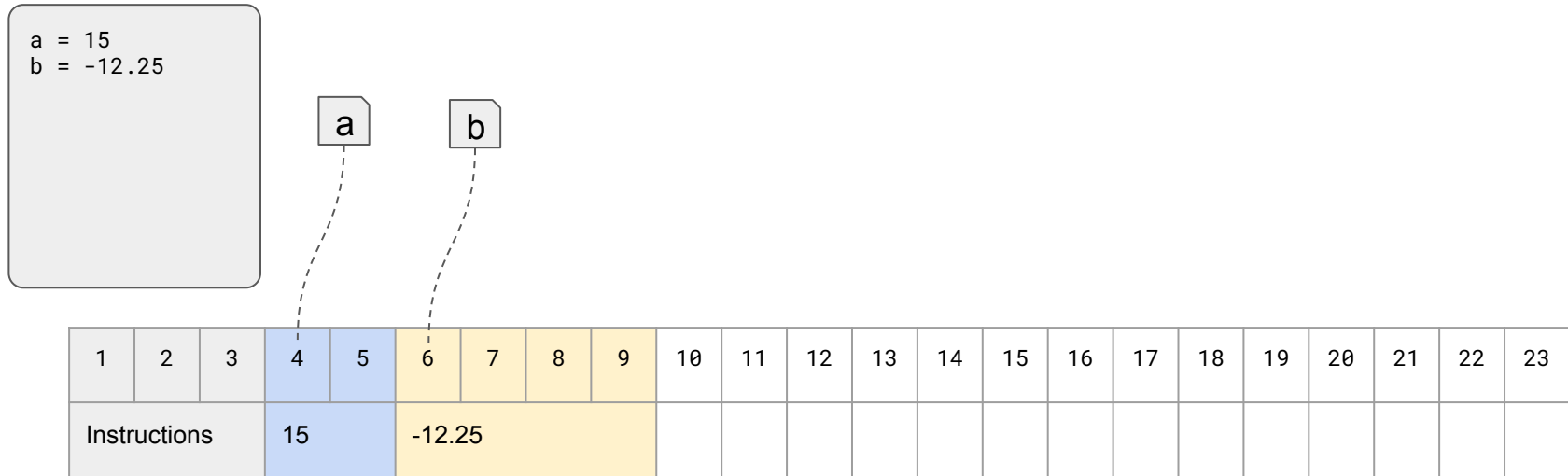
Tableaux de valeurs | **Autres commentaires**

- En MATLAB comme en Python, les tableaux sont de taille variable. Des ajustements (réallocation de mémoire) sont donc cachés au programmeur.
- En Python, le tableau se "comporte" aussi un peu comme une liste, qui encourage davantage les insertions d'éléments $\Rightarrow$ Cf. librairie NumPy pour déclarer de vrais tableaux statiques en Python.
- De plus, le type des valeurs d'une liste en Python donne l'impression d'être mixte (on peut mélanger différents types). Cela cache en réalité un tableau d'adresses d'autres variables !

Une variable qui contient un nombre représentant une adresse de la mémoire est nommée un **pointeur**.

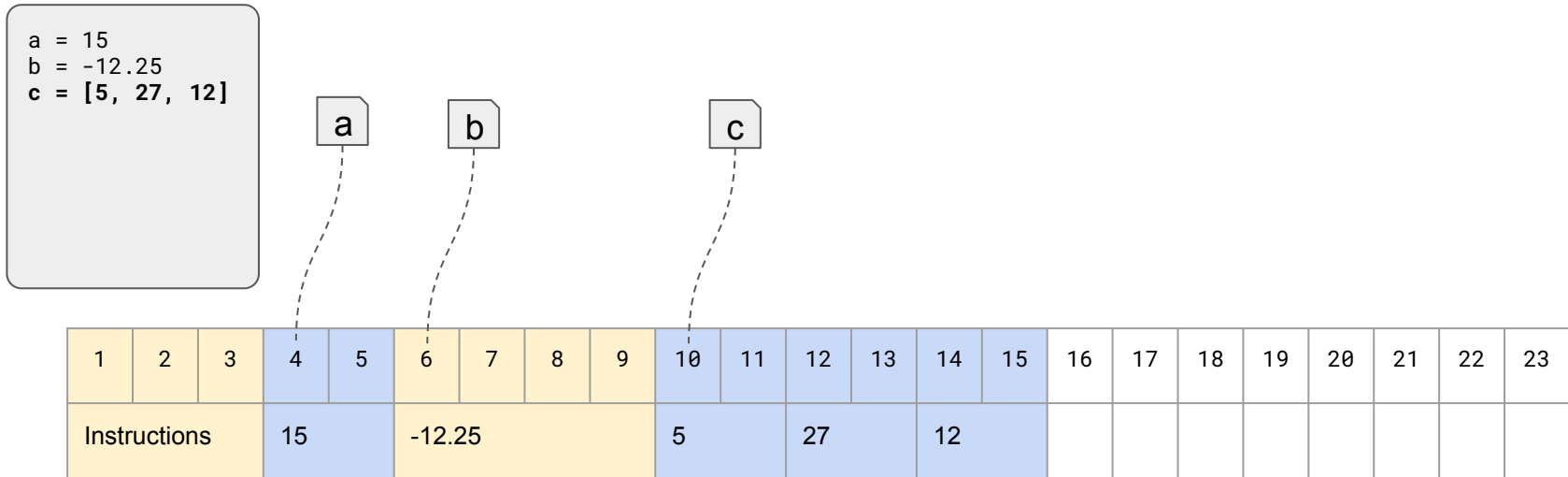
Mémoire allouée au processus | Exemple simplifié

Reprenons où nous en étions.



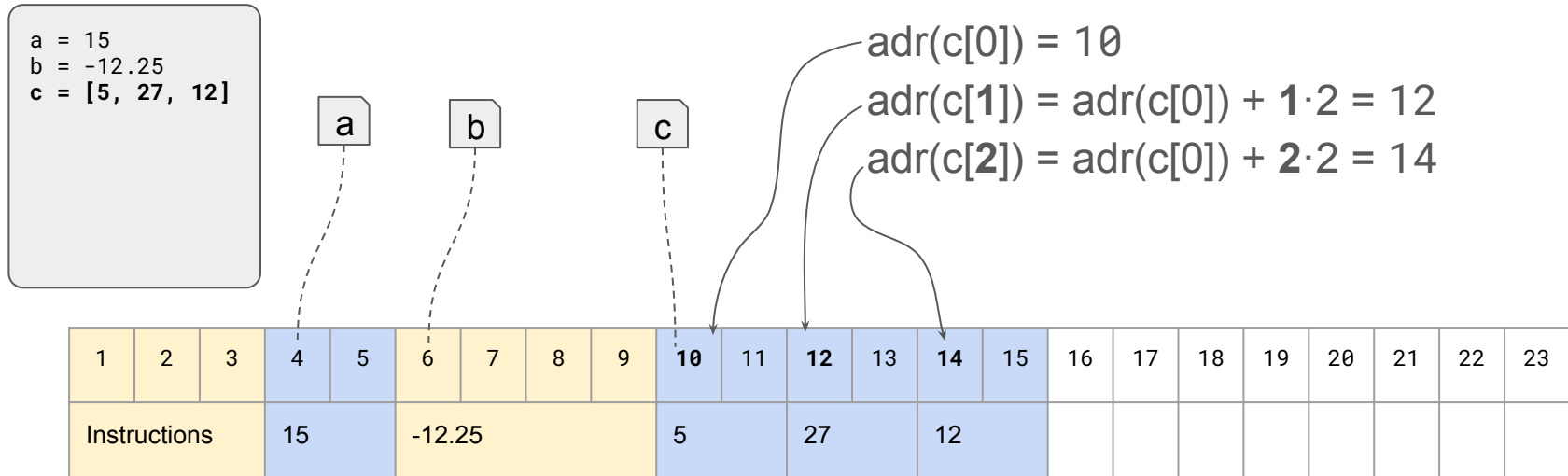
Mémoire allouée au processus | Exemple simplifié

Tableau contenant 3 entiers (chacun codé sur 2 octets pour l'exemple).



Mémoire allouée au processus | Exemple simplifié

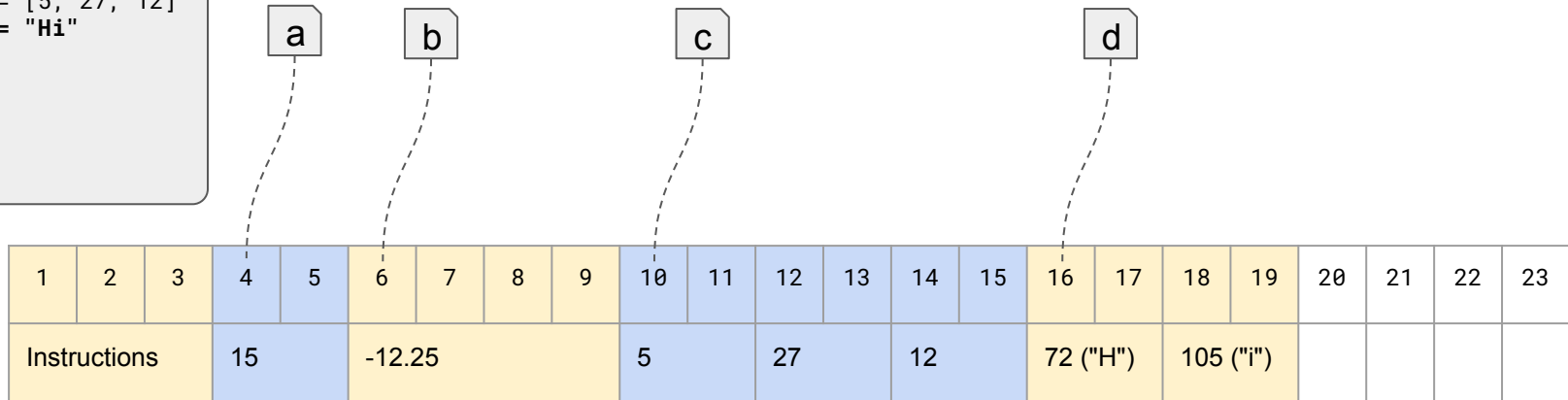
Tableau contenant 3 entiers (chacun codé sur 2 octets pour l'exemple).



Mémoire allouée au processus | Exemple simplifié

Tableau contenant 2 entiers (chacun codé sur 2 octets pour l'exemple).

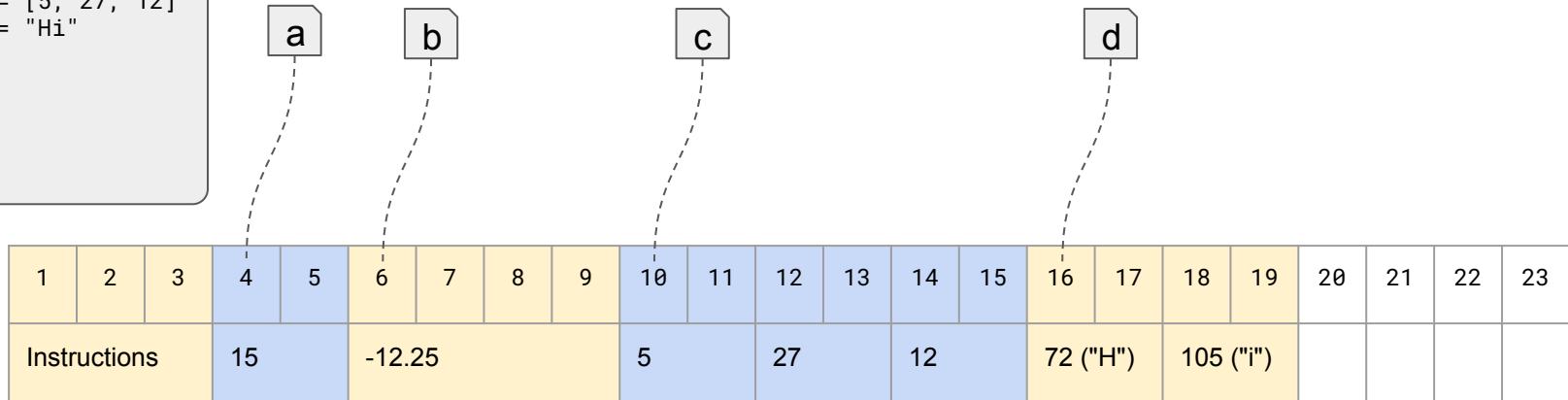
```
a = 15
b = -12.25
c = [5, 27, 12]
d = "Hi"
```



Mémoire allouée au processus | Exemple simplifié

Et si l'on veut ajouter une valeur au sein du tableau c ?

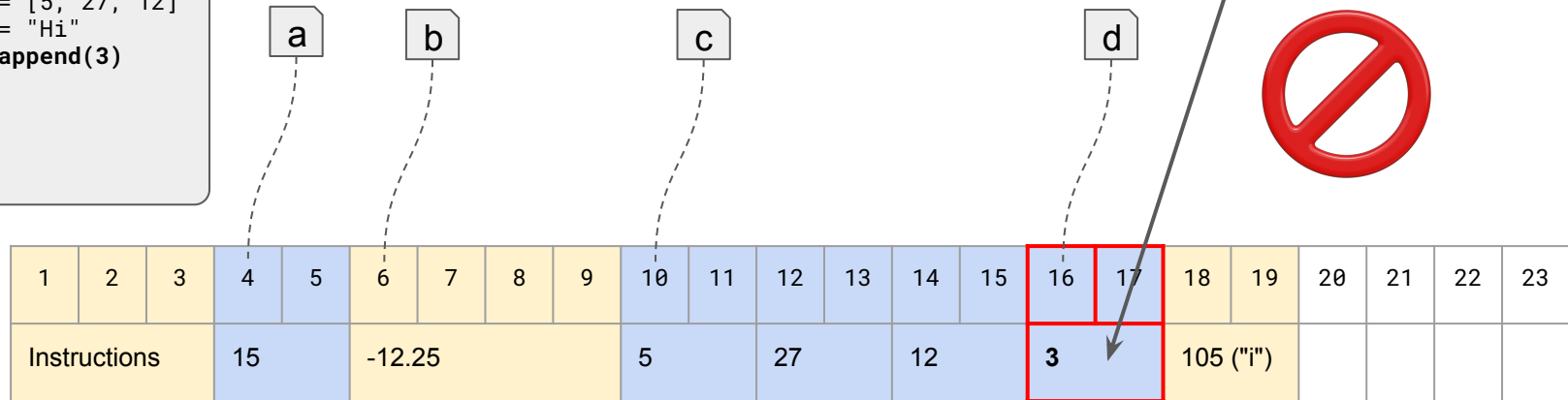
```
a = 15
b = -12.25
c = [5, 27, 12]
d = "Hi"
```



Mémoire allouée au processus | Exemple simplifié

Et si l'on veut ajouter une valeur au sein du tableau c ?

```
a = 15
b = -12.25
c = [5, 27, 12]
d = "Hi"
c.append(3)
```



Tableaux de valeurs dynamiques



Retour aux tableaux | Tableaux de taille variable

- Pour que les tableaux de valeurs vus précédemment fonctionnent, il faut qu'une suite d'adresses ait été **réservée en mémoire**.
- Les adresses suivantes peuvent être utilisées par d'autres données du même processus, mais cela peut aussi être une partie de la mémoire dévolue à d'autres processus.
- Les tableaux de valeurs vus jusqu'ici sont donc de **taille fixe**.
- Comment réaliser un tableau de taille variable, aussi nommé **tableau dynamique** ?

Tableaux dynamiques | **Comportement**

- Contient une suite ordonnée de valeurs.
- Deux comportements souvent attendus d'un tableau dynamique :
 - `append(t, n)` permet d'ajouter la valeur `n` à la fin du tableau `t`.
 - `pop(t)` permet de supprimer le dernier élément du tableau `t`.
- NB : le **nom** des méthodes/fonctions d'une implémentation donnée peut varier.
Par exemple : `append` VS `push_back` VS `push`

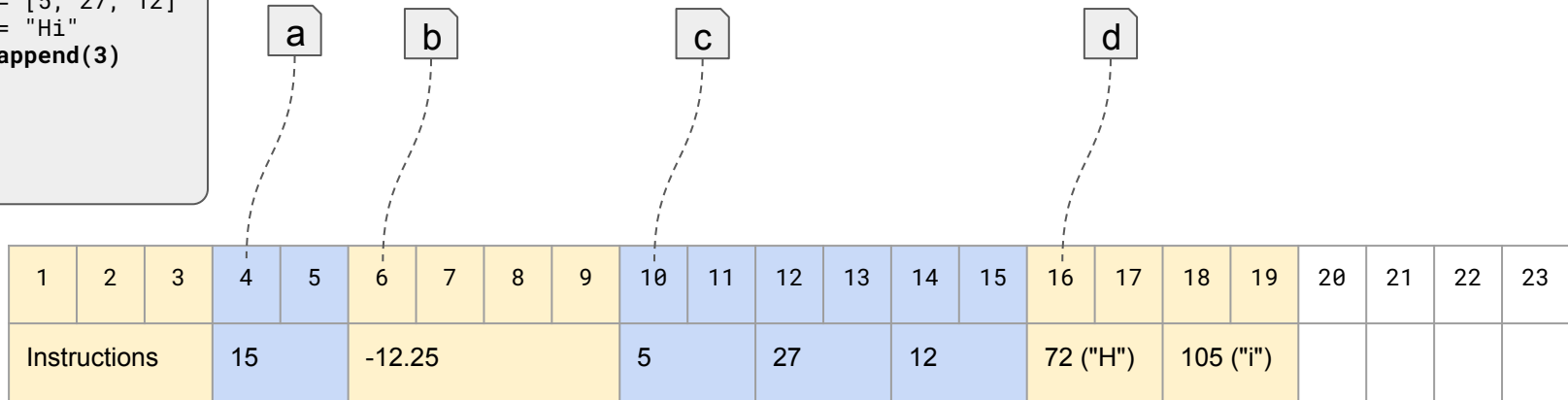
Tableaux dynamiques | Implémentation possible

- Tout comme pour les tableaux statiques, on doit se souvenir de l'adresse de base $\text{addr}(0)$ du tableau ainsi que du nombre k d'octets occupés par chaque valeur.
- Une façon commune de procéder est d'allouer une certaine capacité maximale M au tableau et de distinguer cette valeur du nombre d'éléments L réellement insérés au sein du tableau.
- À chaque fois que $\text{append}()$ est appelée, si $L = M - 1$ on "déménage" entièrement le tableau ailleurs en mémoire (réallocation), en choisissant pour la suite une valeur de M plus grande.

Mémoire allouée au processus | Exemple simplifié

Et si l'on veut ajouter une valeur au sein du tableau c ?

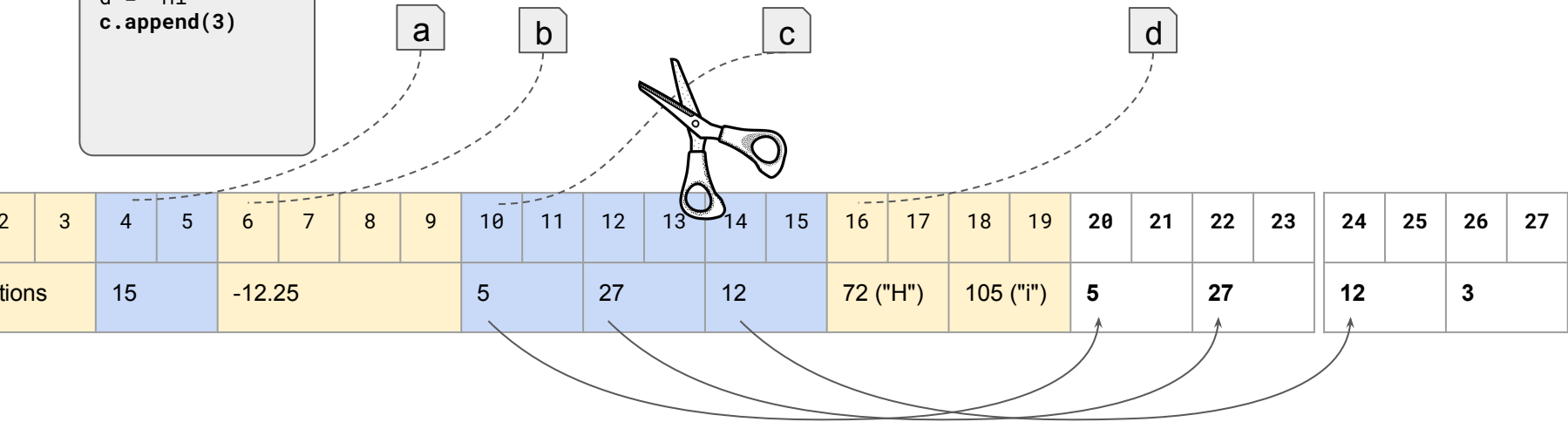
```
a = 15
b = -12.25
c = [5, 27, 12]
d = "Hi"
c.append(3)
```



Mémoire allouée au processus | Exemple simplifié

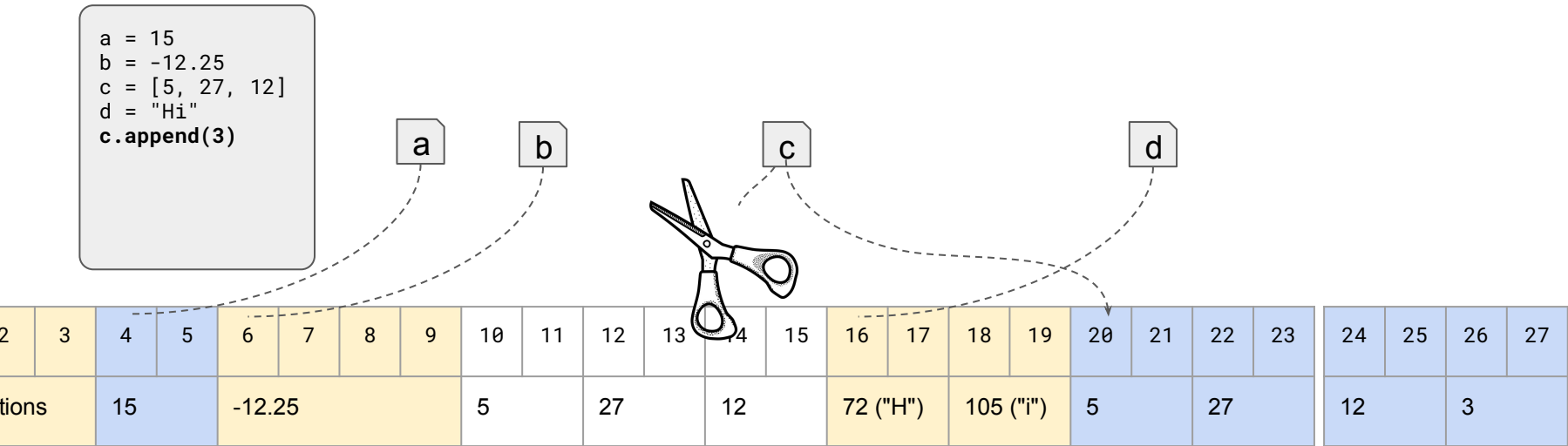
Réallocation de la mémoire nécessaire pour la "nouvelle version" de c en mémoire.
Copie des valeurs.

```
a = 15
b = -12.25
c = [5, 27, 12]
d = "Hi"
c.append(3)
```



Mémoire allouée au processus | Exemple simplifié

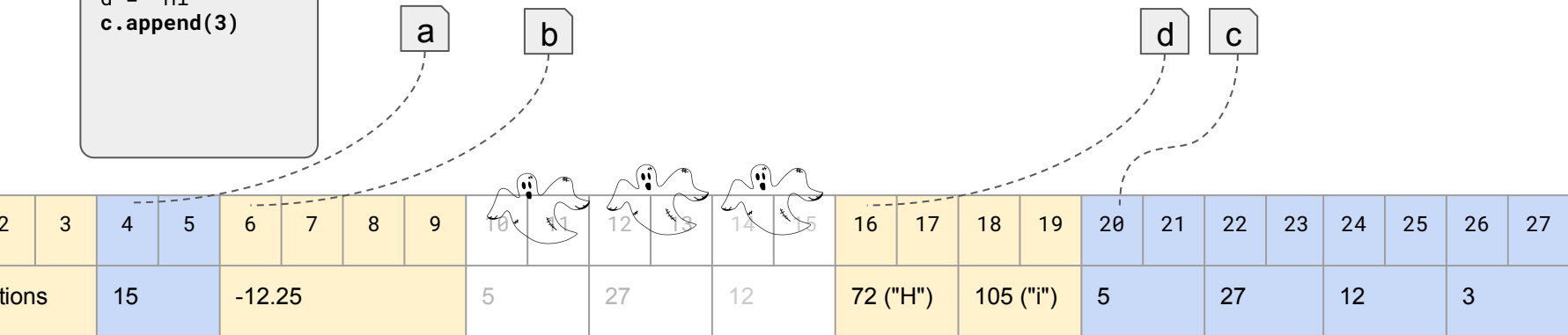
Libération de la mémoire qui était occupée par c dans sa "version précédente".



Mémoire allouée au processus | Exemple simplifié

Le "trou" est réutilisable, mais encore faut-il que les données à venir aient la place d'y entrer. Pour l'instant, la mémoire est donc **fragmentée**.

```
a = 15
b = -12.25
c = [5, 27, 12]
d = "Hi"
c.append(3)
```



Tableaux | Avantages et inconvénients



Tableau statique



Rapide d'accès	Nombre fixe d'éléments
Compact en mémoire	



Tableau dynamique



Rapide d'accès	Mal adapté (lent) si la taille change fréquemment ou qu'on insère des éléments ailleurs qu'à la fin
Nombre variable d'éléments	Fragmentation de la mémoire $\Rightarrow$ gaspillage + inefficacité de la mémoire cache



Tableaux en MATLAB et Python

- En MATLAB : `array`.
- En Python : `NumPy array`.
(Ainsi que `list` comme discuté plus bas)



Listes chaînées

Liste chaînée | Pourquoi s'y intéresser ?

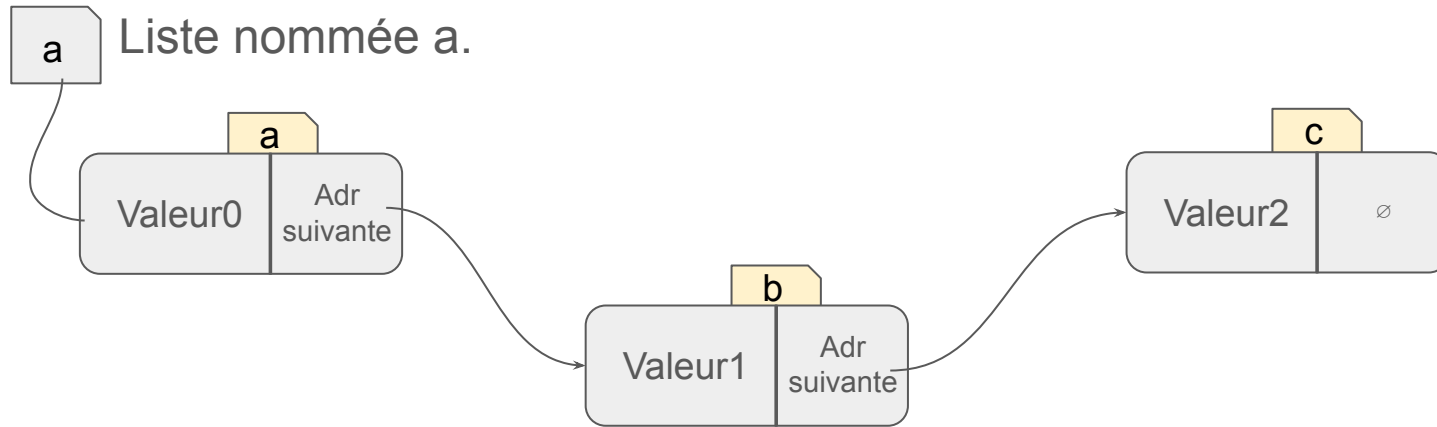
- Si non seulement la taille d'un tableau est amenée à varier, mais qu'en plus celle-ci varie **fréquemment**, les réallocations de mémoire du tableau dynamique peuvent devenir trop handicapantes.
- Par ailleurs, il peut être intéressant d'insérer ou de supprimer des éléments d'indice quelconque (pas seulement à la fin).
- Exemples :
 - Lignes ou bouts de texte au sein d'un logiciel de traitement de texte (insertions et suppressions très fréquentes).
 - Historique des actions dans un navigateur, dans un logiciel, dans un jeu.



Listes chaînées | Comportement

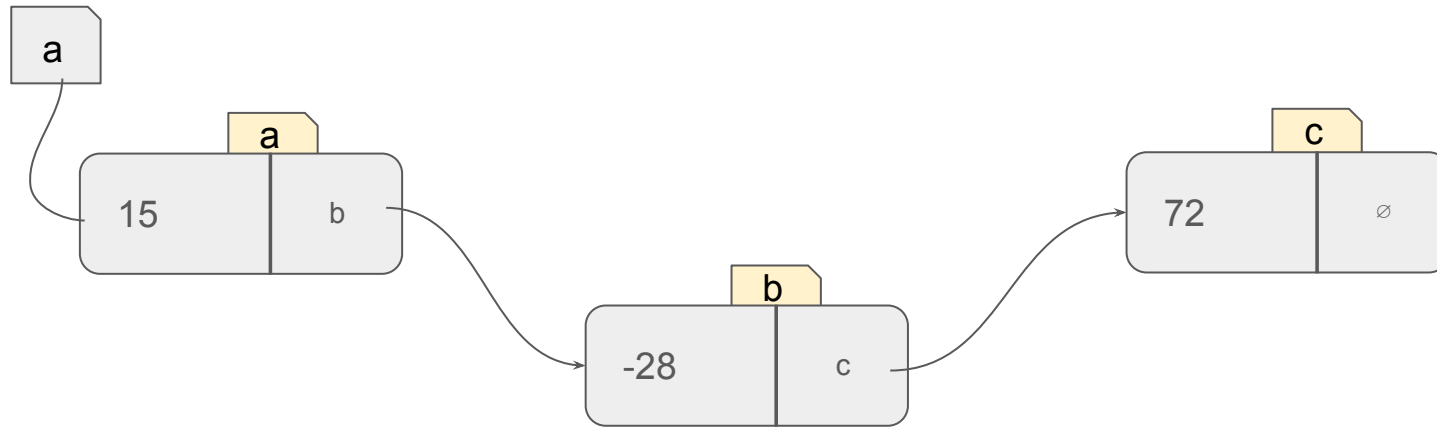
- Contient une suite ordonnée de valeurs.
- Doit permettre à l'utilisateur d'invoquer les comportements suivants :
 - `append(t, n)` ajoute la valeur `n` à la fin de la liste `t`.
 - `remove(t, i)` supprime l'élément numéro `i` de la liste `t`.
 - `insert(t, v, i)` insère la valeur `v` devant l'élément `i` de la liste `t`.

Liste chaînée | Visualisation

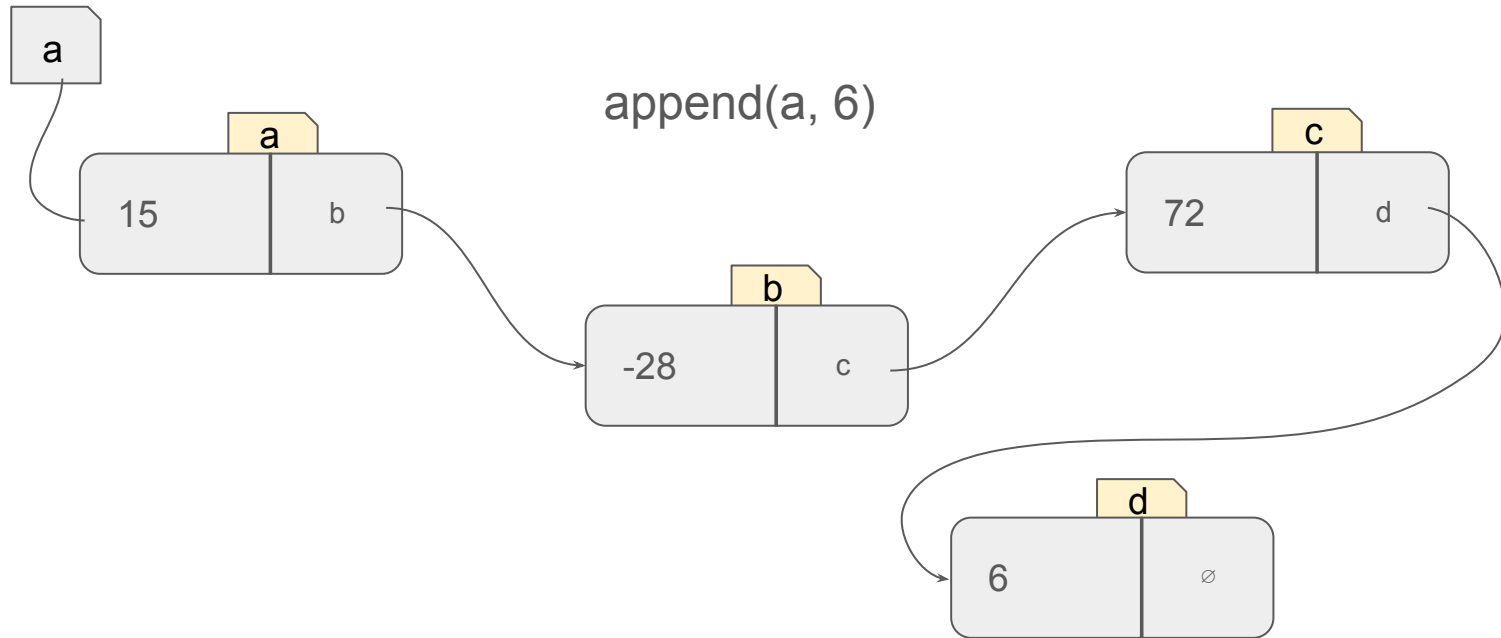


Chaque élément contient à la fois une **valeur** et l'**adresse du prochain** élément.

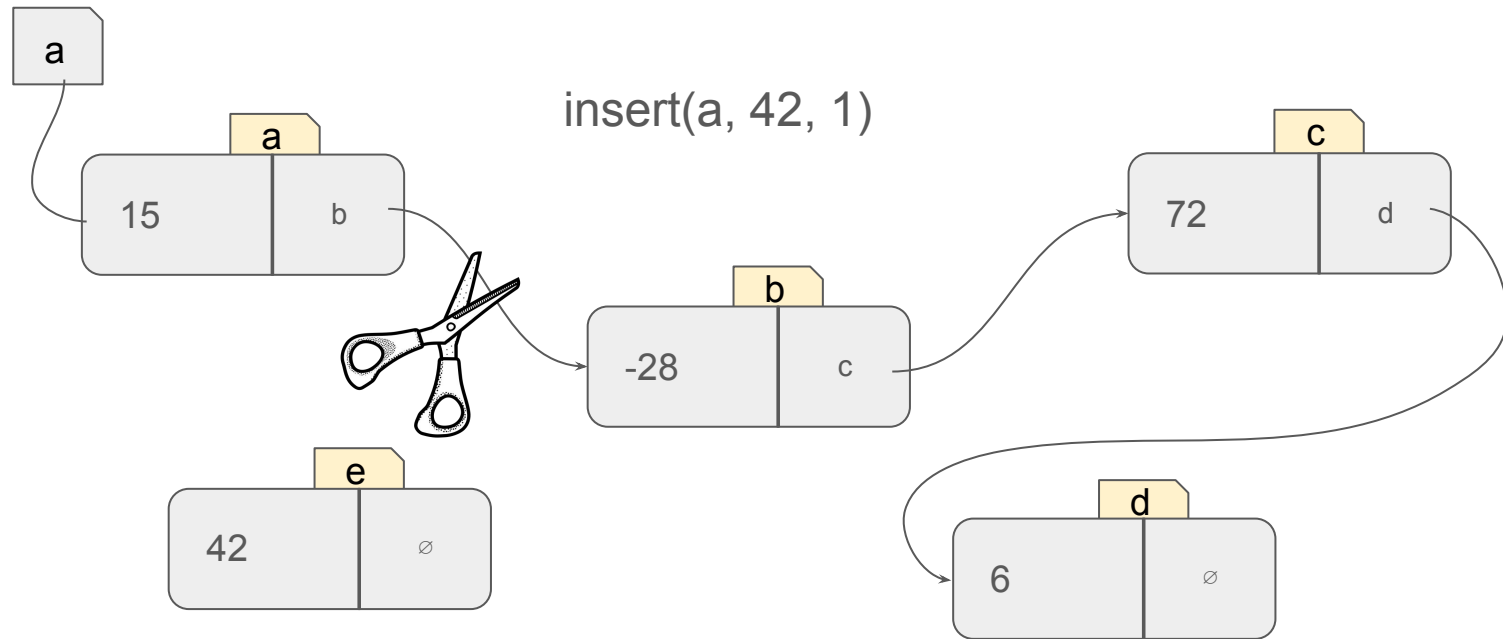
Liste chaînée | Visualisation



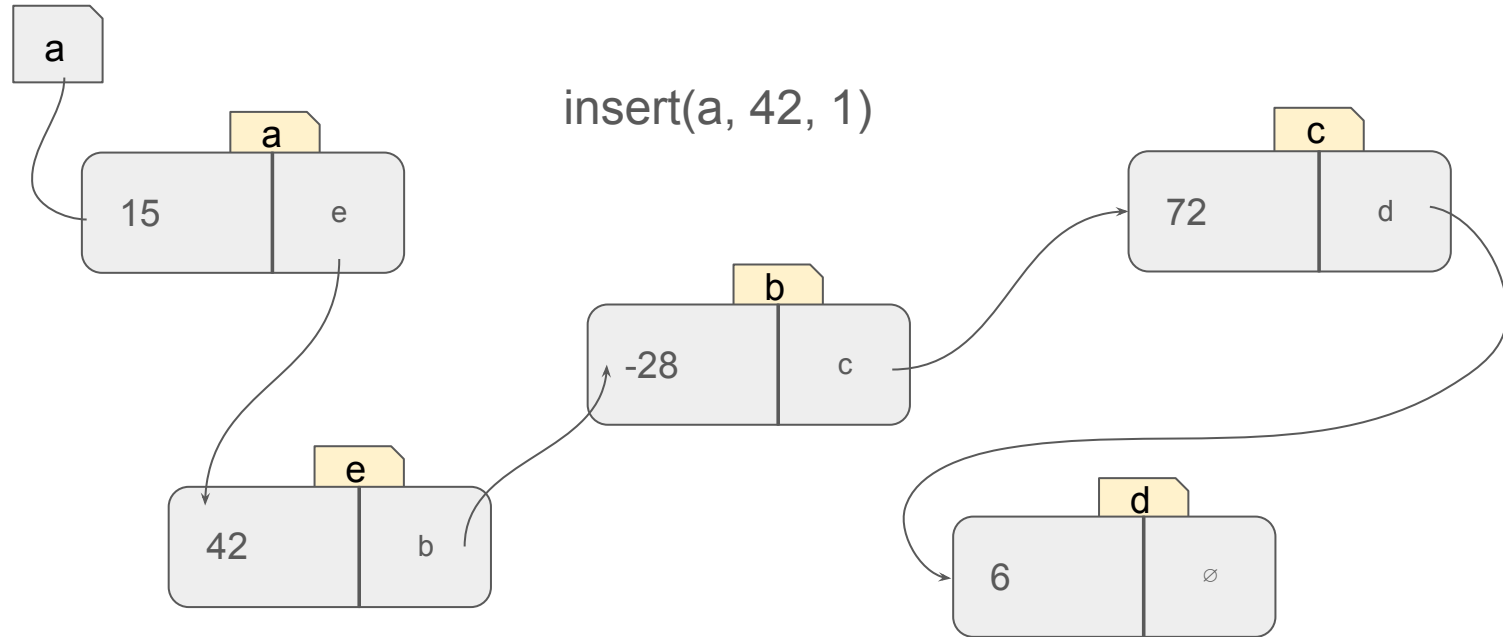
Liste chaînée | Insertion d'un élément à la fin



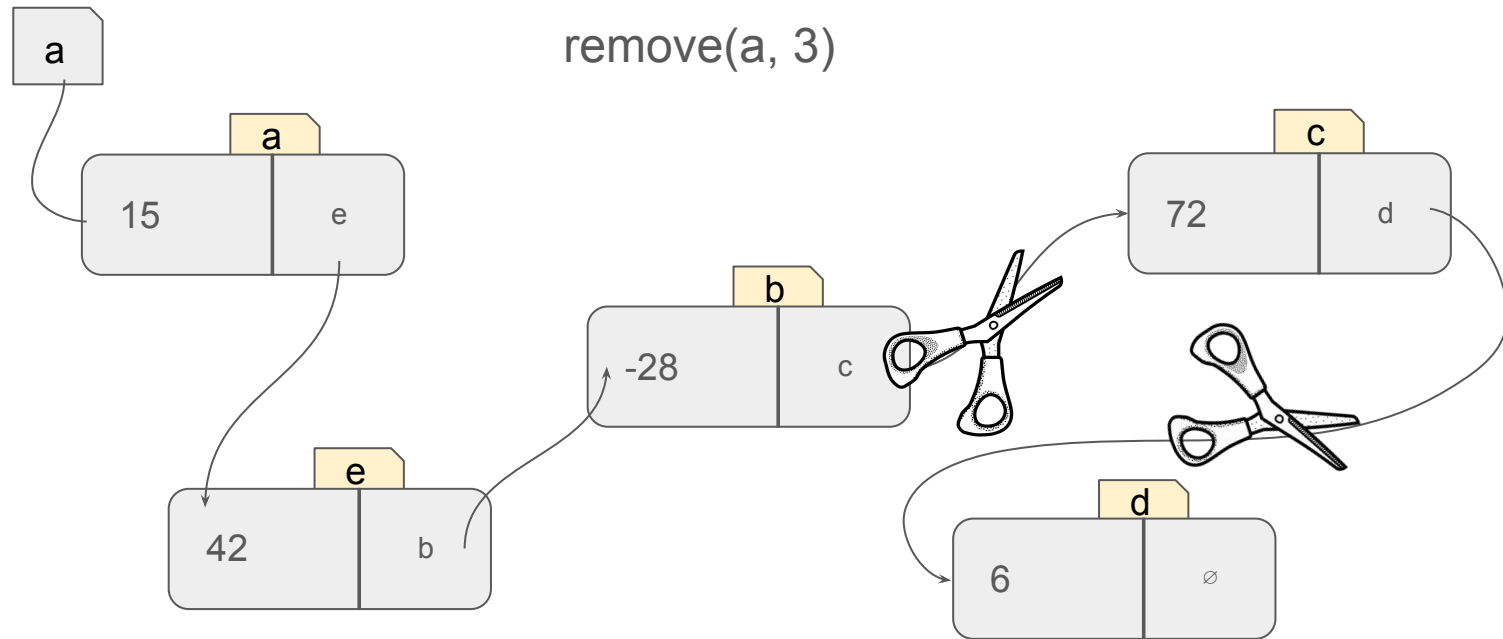
Liste chaînée | Insertion arbitraire



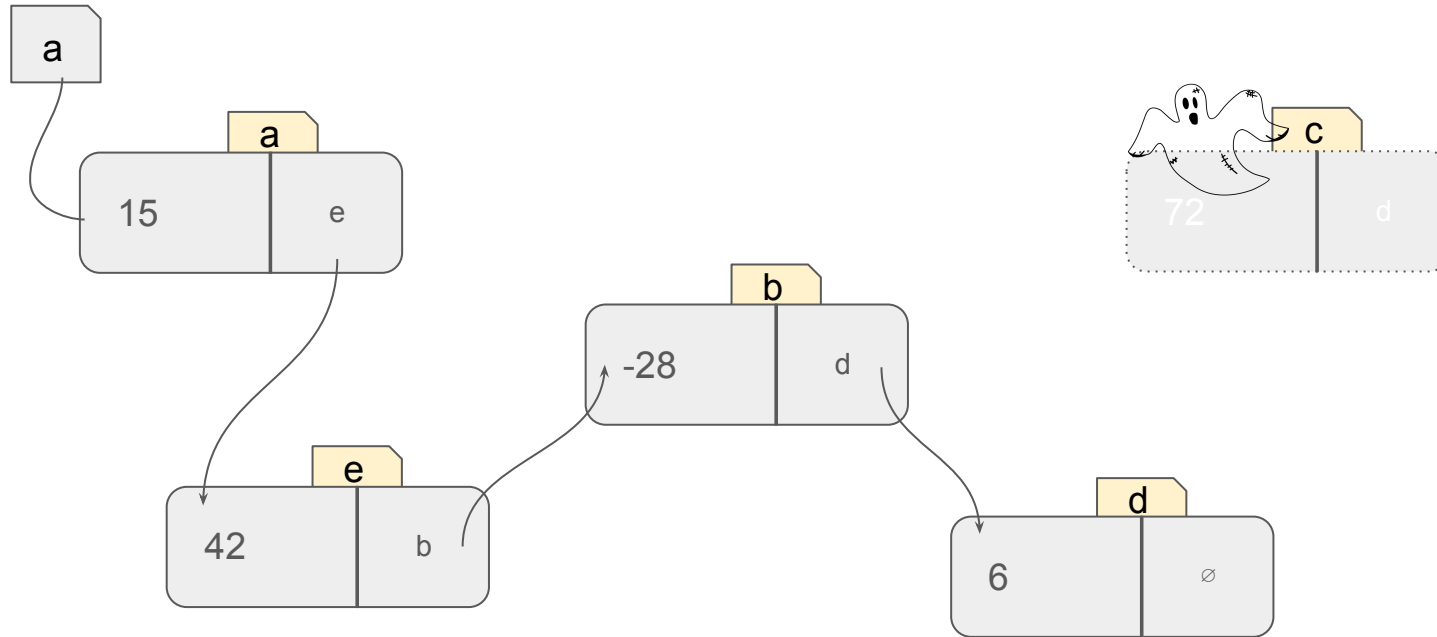
Liste chaînée | Insertion arbitraire



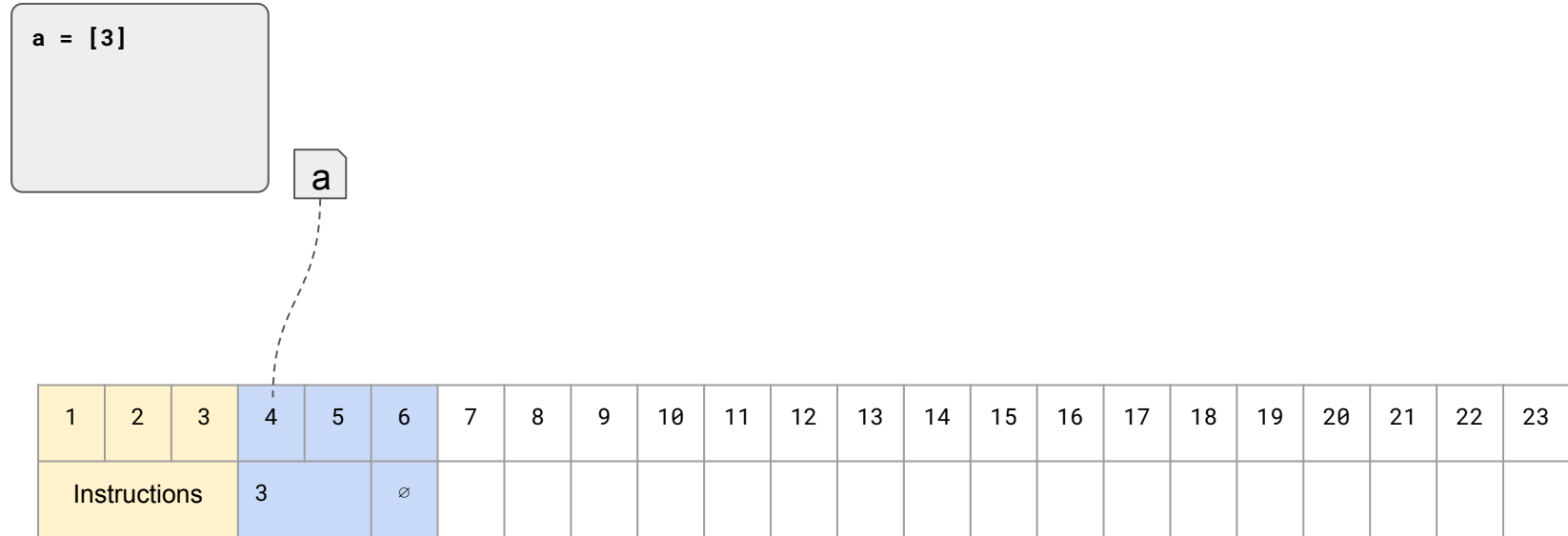
Liste chaînée | **Suppression arbitraire**



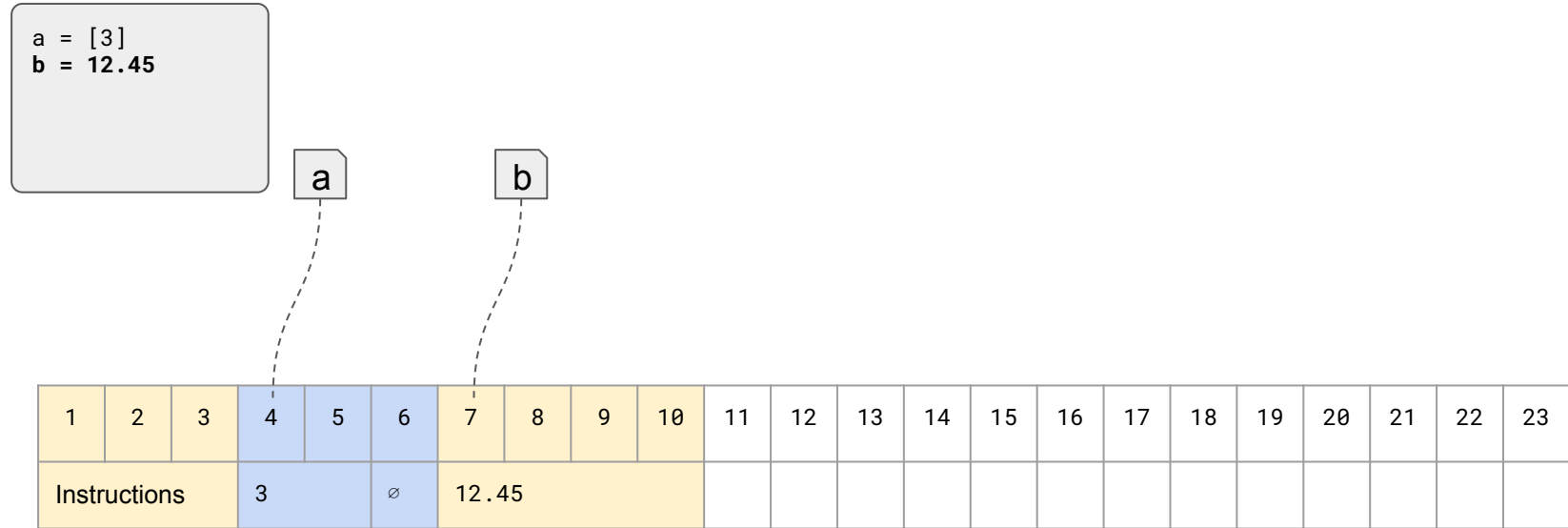
Liste chaînée | **Suppression arbitraire**



Liste chaînée | Principe



Liste chaînée | Principe

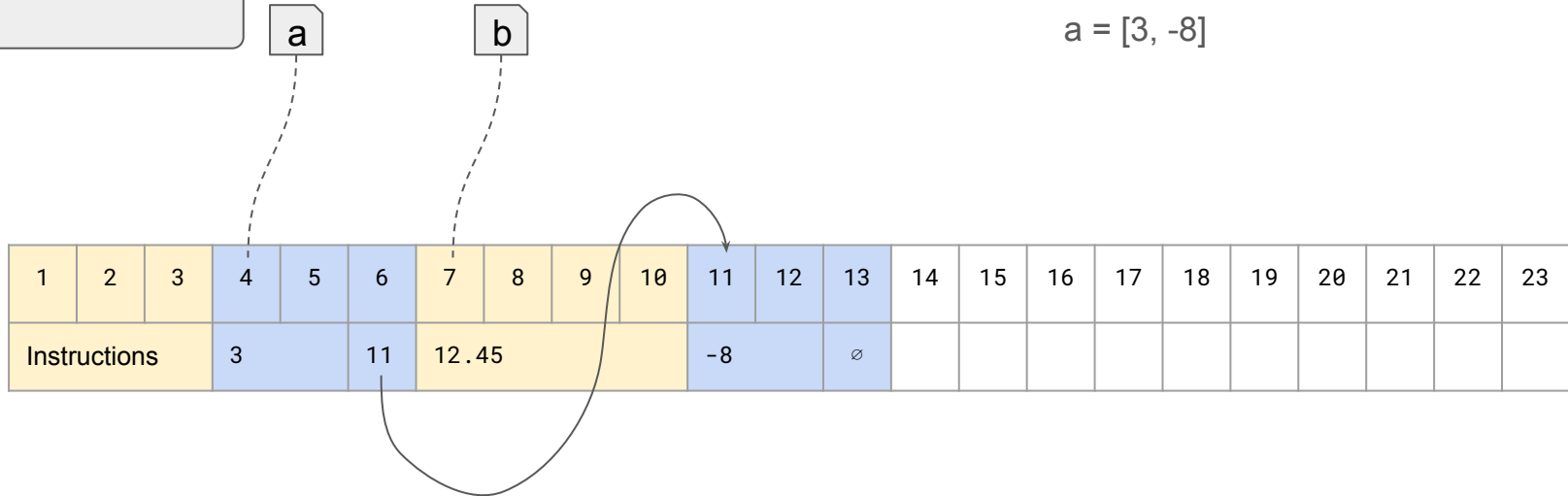


Liste chaînée | Principe

```
a = [3]
b = 12.45
append(a, -8)
```

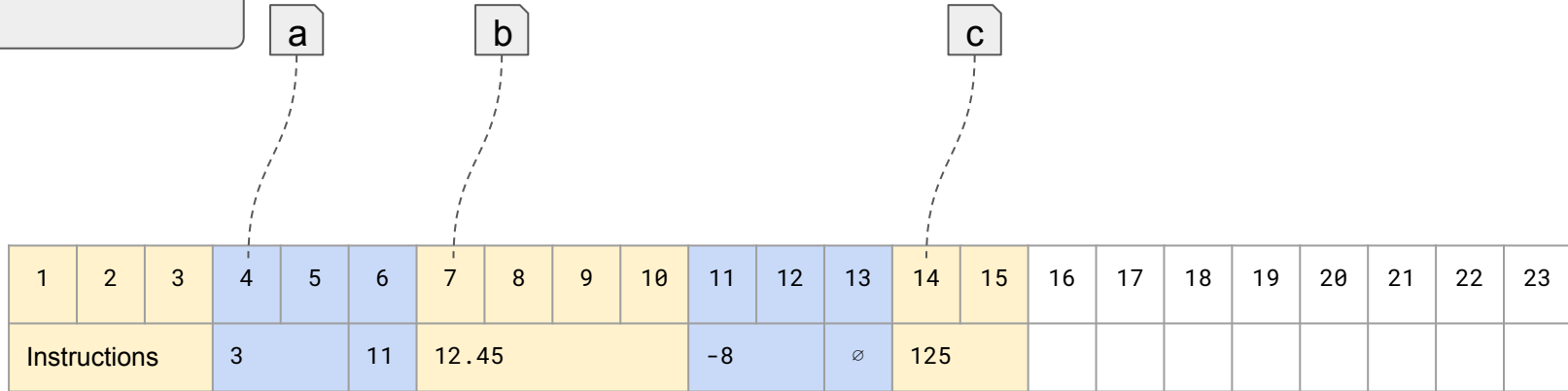
Les éléments peuvent être éparpillés en mémoire.

a = [3, -8]



Liste chaînée | Principe

```
a = [3]
b = 12.45
append(a, -8)
c = 125
```



Liste chaînée | Principe

```
a = [3]
b = 12.45
append(a, -8)
c = 125
insert(a, 72, 1)
```

a

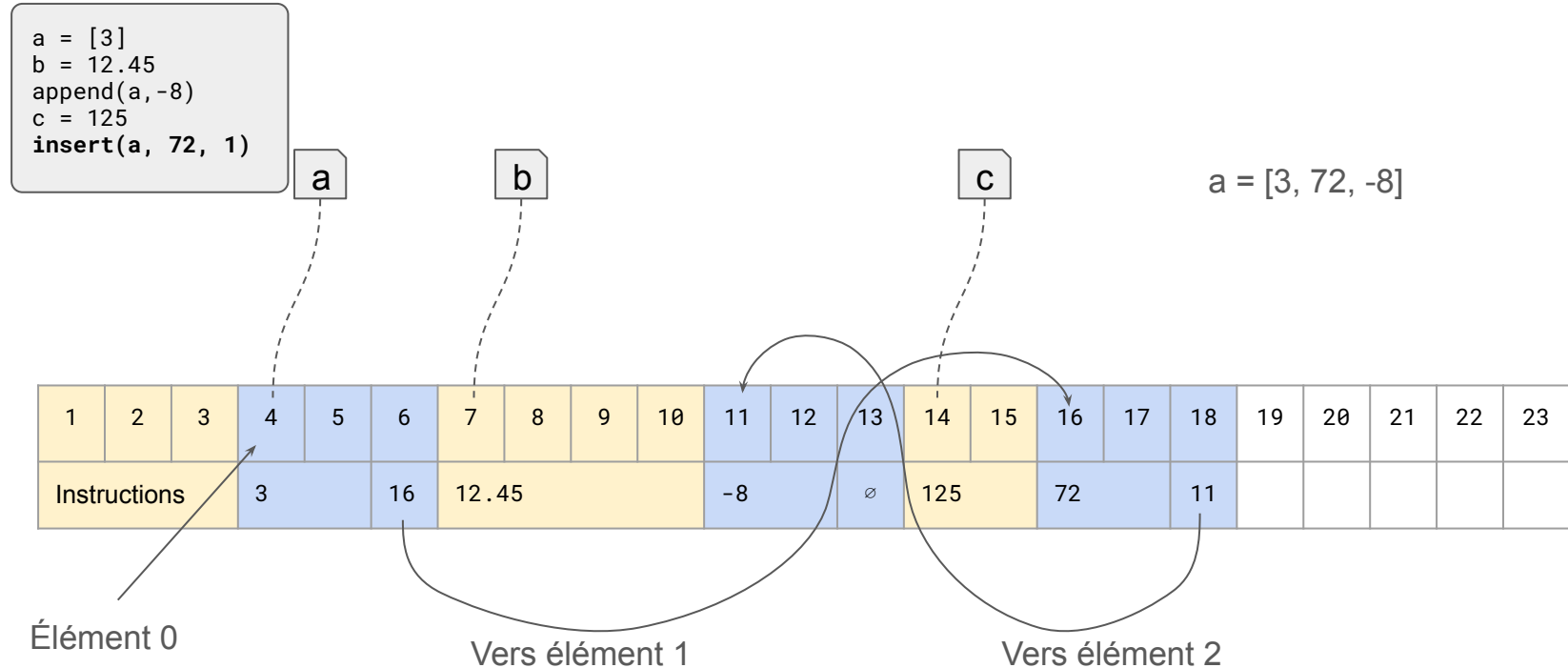
b

c

a = [3, 72, -8]

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Instructions			3		16	12.45				-8		∅	125		72		11					

Liste chaînée | Principe



Liste chaînée | **Complexité d'accès à un élément**

- En **combien d'étapes** peut-on trouver l'adresse de l'élément i de la liste ?
- Une fois cette adresse trouvée, on peut y accéder en un temps égal au temps d'accès à la mémoire centrale, qui ne dépend pas de l'adresse.
- Le **temps d'accès** est approximativement proportionnel au nombre d'étapes nécessaires pour trouver l'adresse de l'élément.
- On va s'intéresser au comportement de ce temps d'accès. Est-il une constante ? Est-il proportionnel à la taille n de la liste ? Pire que cela ? ...



Liste chaînée | **Complexité d'accès à un élément**

- Pour **accéder** à l'élément numéro i , il faut partir de l'élément numéro 0 puis "suivre le fil" $\Rightarrow$ Dans le pire des cas, le temps d'accès est proportionnel à la taille de la liste !
On écrit : temps d'accès $\sim$ nombre d'étapes = $O(n)$.

Liste chaînée | Complexité d'accès et d'insertion

- Pour **accéder** à l'élément numéro i , il faut partir de l'élément numéro 0 puis "suivre le fil" $\Rightarrow$ Dans le pire des cas, le temps d'accès est proportionnel à la taille de la liste !

On écrit : temps d'accès $\sim$ nombre d'étapes = $O(n)$.



Notation "Grand **O**" pour "**O**rdre de".

Cf. chapitre précédent.

Ce qu'il faut retenir : $O(n)$ signifie "est proportionnel à n ".

Se lit : "complexité d'accès en $O(n)$ ", ou encore "complexité linéaire"

Liste chaînée | Complexité d'accès et d'insertion

- Pour **accéder** à l'élément numéro i , il faut partir de l'élément numéro 0 puis "suivre le fil" $\Rightarrow$ Dans le pire des cas, le temps d'accès est proportionnel à la taille de la liste !

On écrit : temps d'accès $\sim$ nombre d'étapes = $O(n)$.



Ici, $O(n)$ en moyenne tout comme dans le pire des cas.

Liste chaînée | Complexité d'accès et d'insertion

- Pour **accéder** à l'élément numéro i , il faut partir de l'élément numéro 0 puis "suivre le fil" $\Rightarrow$ Dans le pire des cas, le temps d'accès est proportionnel à la taille de la liste !

On écrit : temps d'accès $\sim$ nombre d'étapes = $O(n)$.

- Pour **insérer** un élément à l'indice i , il faut se rendre à la bonne adresse (complexité $O(n)$), puis modifier correctement les deux pointeurs concernés (complexité $O(1)$).

Signifie "nombre d'étapes constant"



Liste chaînée | **Complexité d'accès et d'insertion**

- Pour accéder à un élément : $O(n)$.
- Pour insérer un élément : $O(n)$.
- Pour supprimer un élément : $O(n)$.
- Et les tableaux ?

Liste chaînée | Complexité d'accès et d'insertion

Liste chaînée

Accès : $O(n)$.

Insertion : $O(n)$.

Suppression : $O(n)$.

Car nombre
constant d'
étapes pour
trouver/calculer
l'adresse.

Tableau dynamique

Accès : $O(1)$

Insertion : $O(n)$.

Suppression : $O(n)$.

Liste chaînée | Complexité d'accès et d'insertion

Liste chaînée

Accès : $O(n)$.

Insertion : $O(n)$.

Suppression : $O(n)$.

Tableau dynamique

Accès : $O(1)$

Insertion : $O(n)$.

Suppression : $O(n)$.

Même
complexité, mais
certainement pas
même temps !

Liste chaînée | Complexité d'insertion quelconque

	Liste chaînée	Tableau dynamique
Pire des cas	Insertion à la fin	Insertion au début
Implication sur l'allocation mémoire	Allouer de l'espace pour le prochain élément	Réallouer de l'espace pour tout le "nouveau" tableau
Fréquence typique du pire des cas	"Fréquent"	"Peu fréquent"
Gravité du pire des cas	"Légère"	"Sévère"

Liste chaînée | Complexité d'insertion quelconque

	Liste chaînée	Tableau dynamique
Pire des cas	Insertion à la fin	Insertion au début
Implication sur l'allocation mémoire	Allouer de l'espace pour le prochain élément	Réallouer de l'espace pour tout le "nouveau" tableau
Fréquence typique du pire des cas	"Fréquent"	"Peu fréquent"
Gravité du pire des cas	"Légère"	"Sévère"
Complexité	$O(n)$	$O(n)$

Liste chaînée | Complexité d'accès et d'insertion

Liste chaînée

Accès : $O(n)$.

Append : $O(n)$

Pop : $O(n)$

Insertion : $O(n)$.

Suppression : $O(n)$.

Mauvaise foi ?
Cf. complexité
amortie pour
traiter les cas
"catastrophiques".

Tableau dynamique

Accès : $O(1)$

Append : $O(1)$

Pop : $O(1)$

Insertion : $O(n)$.

Suppression : $O(n)$.

Complexité amortie | Exemple de append sur tableau

- L'opération `append()` sur un tableau dynamique peut (ou pas) nécessiter une réallocation du tableau.
- À chaque "déménagement" du tableau entier, ce déménagement est $O(n)$ si le tableau est de taille n à cet instant.
- La complexité d'append sur un tableau de taille n est donc :
 - $O(1)$ s'il n'y a pas de déménagement à faire.
 - $O(n)$ s'il y a un déménagement à faire.
- Question : si je fais grandir un tableau de la taille 0 à la taille n , quelle est la complexité moyenne par opération `append` ?
- Réponse : dépend de la stratégie de redimensionnement !



Complexité amortie | **Exemple de append sur tableau**

Stratégie de redimensionnement commune : si le tableau est de taille n et qu'il va déborder, on le déménage et sa nouvelle taille est $2n$.

Tableaux et listes chaînées | Avantages et inconvénients



Tableau statique



Rapide d'accès : $O(1)$	Nombre fixe d'éléments
Compact en mémoire	



Tableau dynamique



Rapide d'accès : $O(1)$	Mal adapté (lent) si la taille change fréquemment
Nombre variable d'éléments	Potentielle fragmentation



Liste chaînée



Nombre variable d'éléments	Lente d'accès : $O(n)$
Adaptée si la taille change fréquemment	Potentielle fragmentation
Adaptée pour l'insertion à des indices arbitraires	

Commentaires à propos de l'efficacité des tableaux

- Les éléments des listes chaînées ont plus de chance de ne pas se trouver dans la **mémoire cache**, comme ils ne sont pas contigus.
- La plupart du temps, les tableaux offriront de meilleures performances en raison du gain drastique de temps d'accès procuré par la mémoire cache (en plus des raisons plus fondamentales discutées auparavant).
- Les tableaux sont particulièrement inefficaces pour insérer des éléments à un indice arbitraire, mais restent efficaces si on se contente d'ajouter des éléments à la fin.
- Pour une utilisation optimale des tableaux dynamiques, évitez de modifier leur taille.
Exemple en MATLAB:
a = zeros(1, 200), puis remplissage des valeurs, est plus efficace que :
a = [], puis peuplement de 200 éléments l'un après l'autre.

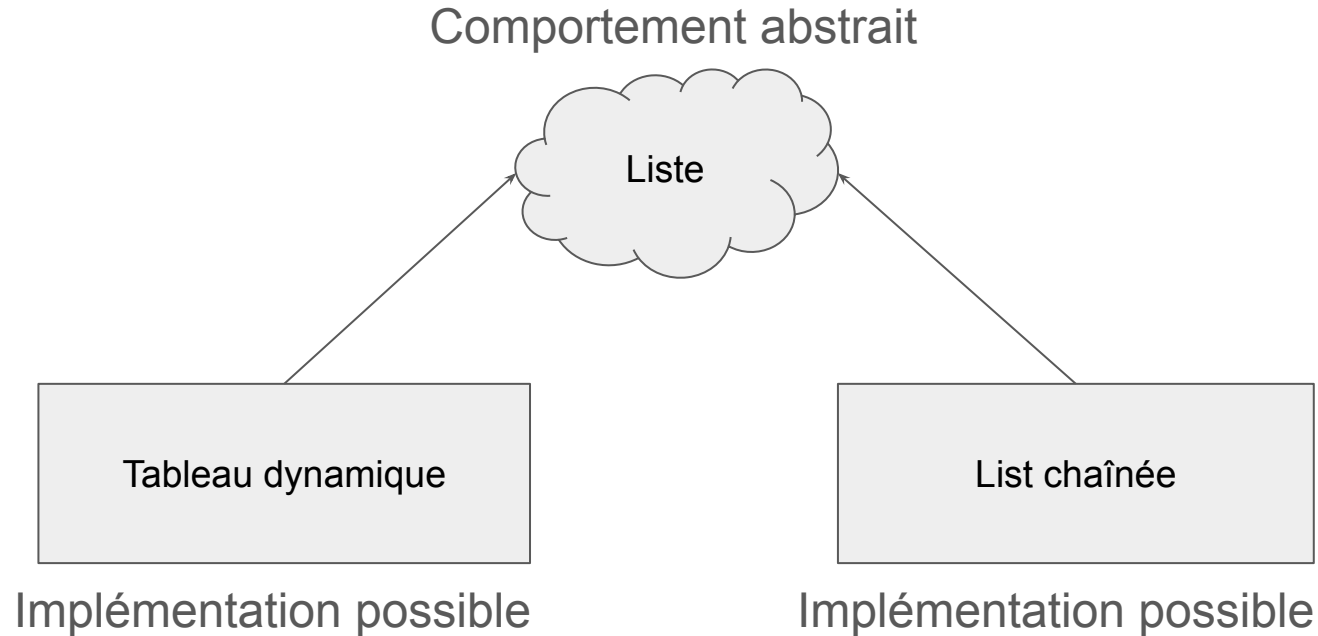
Commentaires à propos des listes chaînées

- L'accès à un élément arbitraire de la liste est $O(n)$...
- ... mais dans certains cas, l'algorithme peut exploiter l'ordre du parcours des éléments (cf. recherche du min)
⇒ toujours se demander si on parle de :
 - `get(ma_liste, i_quelconque)` (ceci est $O(n)$)
 - ou de `get_next(mon_noeud, ma_liste)` (ceci est $O(1)$)
- Une boucle qui itère sur les éléments d'une liste chaînée reste donc $O(n)$.

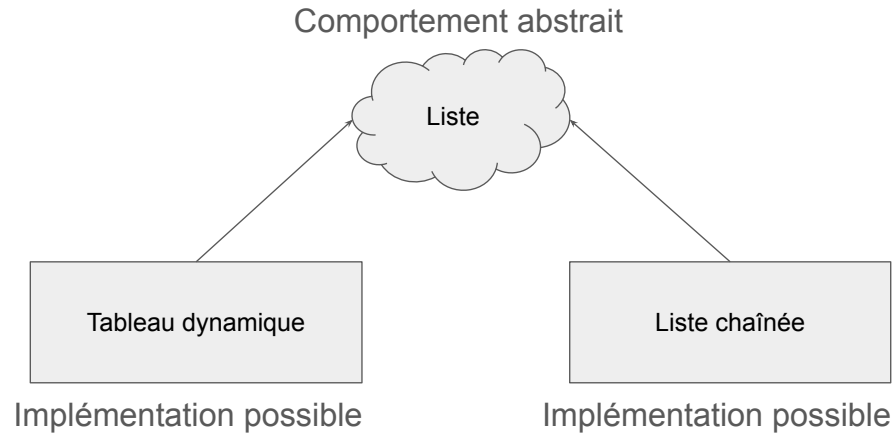
Types de données abstraits

- Types concrets (int, float) : traités par le processeur en un cycle d'horloge, contrairement à des types abstraits.
- Certains types de données correspondent à des concepts plus élaborés qu'un simple nombre où suite de nombres en mémoire.
- Ces types, qui sont **définis par la façon** dont le programmeur peut **interagir** avec eux, sont nommés **types de données abstraits**.
- Au final, l'implémentation d'un type abstrait repose sur sa décomposition et son traitement via des types concrets, au sein d'une **structure de donnée**.
- La façon dont un type abstrait peut être implémenté *via* une structure de donnée n'est pas unique.

Types de données abstraits VS concrets | **Exemple**



Types de données abstraits VS concrets | **Exemple**



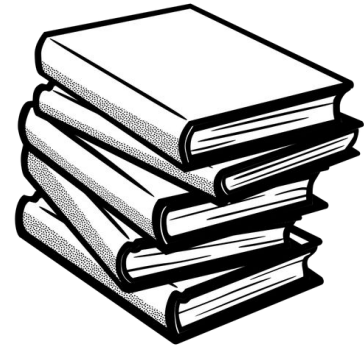
En l'occurrence, l'implémentation CPython du type `list` correspond à un tableau dynamique : <https://github.com/python/cpython/blob/main/Objects/listobject.c>

Quelques exemples de types abstraits

Type abstrait (comportement attendu)	Structure de donnée sous-jacente (exemples)	Implémentation/interface précise
Tableau dynamique	Tableau statique avec stratégie de réallocation	C++ <code>std::vector</code> Python <code>liste</code> Matlab <code>array</code>
Liste	Tableau statique avec stratégie de réallocation	C++ <code>std::list</code> Python <code>liste</code> Matlab <code>aucune</code>
	Liste chaînée	
Dictionnaire	Table de hachage	C++ <code>std::map</code> Python <code>liste</code> Matlab <code>map</code>
	Arbre de recherche binaire	

Types de données abstraits | Exemple de la pile

- Pile d'éléments : *Last In, First Out (LIFO)*.
- Définie par son **comportement** :
 - `add(pile, n)` : ajoute `n` au sommet de la pile.
 $[a, b, c] \rightarrow [a, b, c, n]$
 - `pop(pile)` : ôte le nombre au sommet de la pile.
 $[a, b, c] \rightarrow [a, b]$
- La façon dont `add` et `pop` sont écrites est "cachée" au programmeur ; il n'a pas besoin de savoir comment `add` et `pop` sont écrites pour pouvoir les utiliser.
- Exemple d'**implémentation** : cas particulier de liste chaînée.



Exemple d'implémentation d'une pile en Python

- Dans cet exemple, on utilise une classe uniquement comme un "agrégateur" de variables.
- Afin de ne pas dévier la discussion sur l'utilisation des classes en Python, tout le reste est réalisé sans l'utilisation de fonctionnalités propres aux classes.

(Diapositive hors champ)



Exemple d'implémentation d'une pile en Python

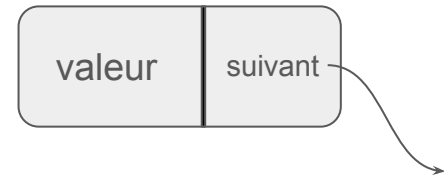
```
class Noeud:  
    def __init__(self, valeur, suivant):  
        self.valeur:int = valeur  
        self.suivant:Noeud = suivant
```

```
class Pile:  
    def __init__(self):  
        self.sommet:Noeud = None
```

(Diapositive hors champ)

Exemple d'implémentation d'une pile en Python

```
class Noeud:  
    def __init__(self, valeur, suivant):  
        self.valeur:int = valeur  
        self.suivant:Noeud = suivant
```



```
class Pile:  
    def __init__(self):  
        self.sommet:Noeud = None
```

Pointe sur le noeud suivant
(sauf si None)

(Diapositive hors champ)



Exemple d'implémentation d'une pile en Python

```
def empiler(pile, valeur): #'push'  
    nouveau_sommet = Noeud(valeur, pile)  
    nouveau_sommet.suivant = pile.sommet  
    pile.sommet = nouveau_sommet  
  
def depiler(pile): #'pop'  
    noeud_a_depiler = pile.sommet  
    pile.sommet = noeud_a_depiler.suivant  
    return noeud_a_depiler.valeur
```

On peut imaginer plein d'autres fonctions
(est_vide, calculer_taille, etc.)

(Diapositive hors champ)



Exemple d'implémentation d'une pile en Python

```
p = Pile()
empiler(p, 3)
empiler(p, -5)
empiler(p, 0)
empiler(p, 7)
a = depiler(p)
print("On vient de dépiler la valeur", a)
b = depiler(p)
print("On vient de dépiler la valeur", b)
print("La valeur du sommet en ce moment est", p.sommet.valeur)
```

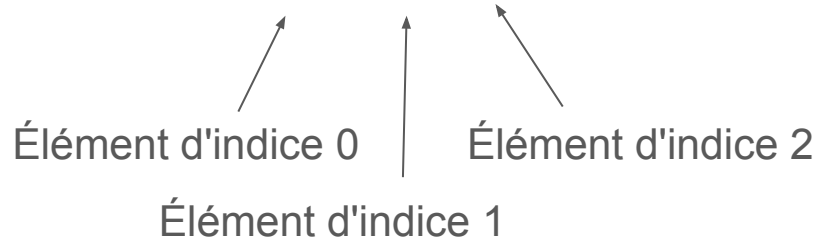
(Diapositive hors champ)



Dictionnaires (tableaux associatifs)

Clés et valeurs | Le cas particulier des tableaux

- Les tableaux peuvent être vus comme des couples (clé : valeur).
- Les clés sont toujours des entiers, en commençant par 0 (Python) ou 1 (MATLAB) pour le premier élément du tableau.
- Exemple en Python : `a = [-7, 3, 5]`



Clés et valeurs | Le cas particulier des tableaux

- Exemple : $T = [-7, 3, 5]$
La clé 0 correspond à la valeur -7.
La clé 2 correspond à la valeur 5.
- Avantage : la clé est implicite, pas besoin d'écrire $T = [0:-7, 1:3, 2:5]$
- Désavantage : selon la nature du problème, le fait de pouvoir utiliser des clés arbitraires est désirable.

Besoin d'un dictionnaire

- Selon la nature du problème, le fait de pouvoir utiliser des clés arbitraires est désirable.
- Exemple : au sein de la population suisse, chaque citoyen possède un numéro d'identification (ID) à 8 chiffres.
Supposons que l'on veuille établir une correspondance entre l'ID d'une personne et son nom.

10000001 → "Dupont"
12005082 → "Robert"
20184663 → "Dubois"

Comment stocker cela dans une structure de donnée ? NB : la population suisse est de quelques millions de personnes.

Besoin d'un dictionnaire | **Solution peu satisfaisante**

Première solution possible, utiliser l'ID comme indice des éléments :

noms = ["", "", "", . . . , "", "Dubois", ... , ""]

Analogie : on veut ranger dans des tiroirs les photos des personnes. Ici, le meuble est gigantesque, mais il est facile de trouver le bon tiroir, car il y a une correspondance directe entre l'emplacement du tiroir et son numéro.

Temps d'accès à un élément : $O(1)$

Espace mémoire : $O(\max(\text{IDs}))$



Image générée avec une IA

Besoin d'un dictionnaire | **Solution peu satisfaisante (2)**

- Supposons toujours que l'on veuille établir une correspondance entre l'ID d'une personne et son nom.
- Seconde solution possible :
IDS = [10000001, 12005082, 20184663, ...]
noms = ["Dupont", "Robert", "Dubois", ...]

Quand on cherche le nom associé à une ID, on fait une boucle sur jusqu'à trouver l'indice i de l'ID que l'on cherche. Le nom correspondant est alors `noms[i]`.

- Bien mieux : $O(n)$ en mémoire.
- Problème : **lent car il faut itérer** sur les IDs pour trouver le nom correspondant. En l'occurrence, la complexité d'accès est $O(n)$.

Besoin d'un dictionnaire | **Solution peu satisfaisante (2)**

Seconde solution possible :

```
IDS   = [10000001, 12005082, 20184663, ...]  
noms  = ["Dupont", "Robert", "Dubois", ...]
```

Analogie du meuble à tiroirs : Cette fois, le meuble est plus petit, mais **il faut lire chaque étiquette pour savoir** si le tiroir est celui qui contient la bonne photo.

Temps d'accès : $O(n)$ car il faut potentiellement lire chaque étiquette pour trouver le bon tiroir.

Espace mémoire : $O(n)$ car on utilise autant de tiroirs que de données.



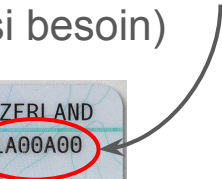
Image générée avec une IA

Dictionnaires

- Supposons toujours que l'on veuille établir une correspondance entre l'ID d'une personne et son nom.
- Solution idéale :
Concevoir un type abstrait qui permette d'écrire des choses telles que :
`{10000001 : "Dupont", 12005082 : "Robert", 20184663 : "Dubois", ...}`

⇒ Comment **implémenter** cela sans utiliser l'une des méthodes précédentes ?
- Ce type de donnée abstrait est nommé **dictionnaire** ou encore **tableau associatif**.

- On veut implémenter un type abstrait tel que
`{10000001:"Dupont", 12005082:"Robert", 20184663:"Dubois", ...}`
- Plus généralement, on aimerait pouvoir utiliser une séquence arbitraire comme clé !
(On utilisera des entiers pour coder les caractères si besoin) /

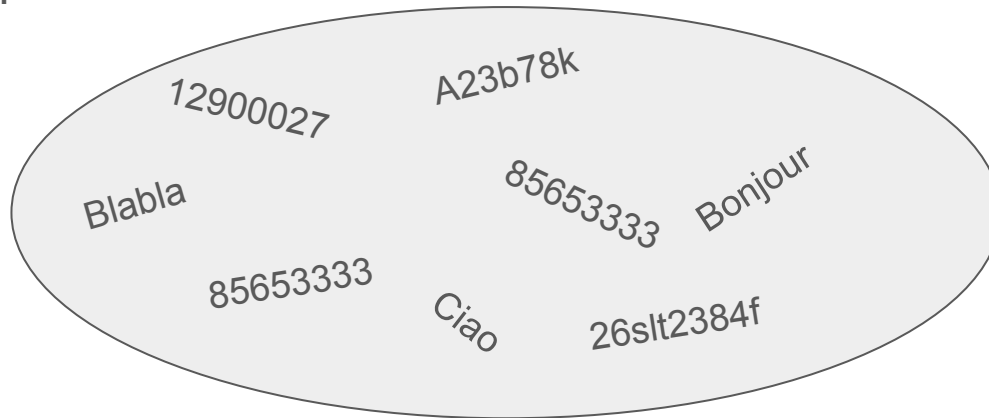


Comment implémenter un dictionnaire

- Comment l'implémenter en étant à la fois efficace en temps et en mémoire ?
Spoiler : il existe un moyen d'avoir un accès moyen en $O(1)$ comme pour la solution 1, tout en occupant un espace mémoire en $O(n)$ comme pour la solution 2.
- Deux approches populaires pour implémenter un dictionnaire : l'une utilise des arbres, l'autre des **tables de hachage**. Nous examinons ici la seconde.

Tables de hachage | Principe de base (1)

- L'ensemble des **clés possibles** est beaucoup plus grand que le nombre de **valeurs** que l'on souhaite réellement stocker.
- Cela nous empêche d'établir de façon raisonnable une correspondance unique entre clés et valeurs.

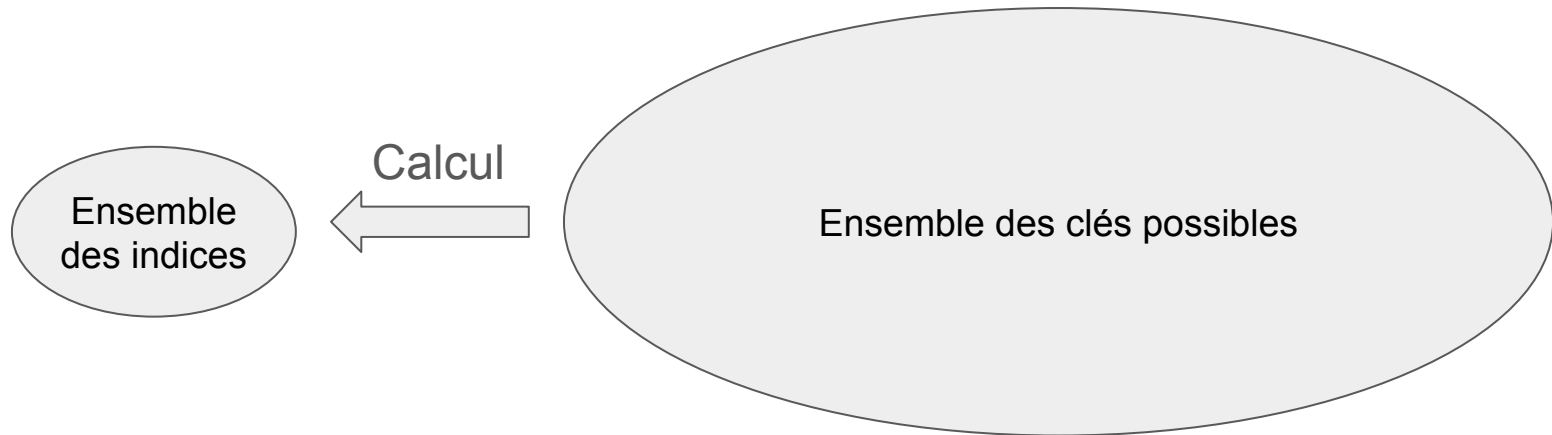


Ensemble des clés possibles
(très vaste)

Tables de hachage | Principe de base

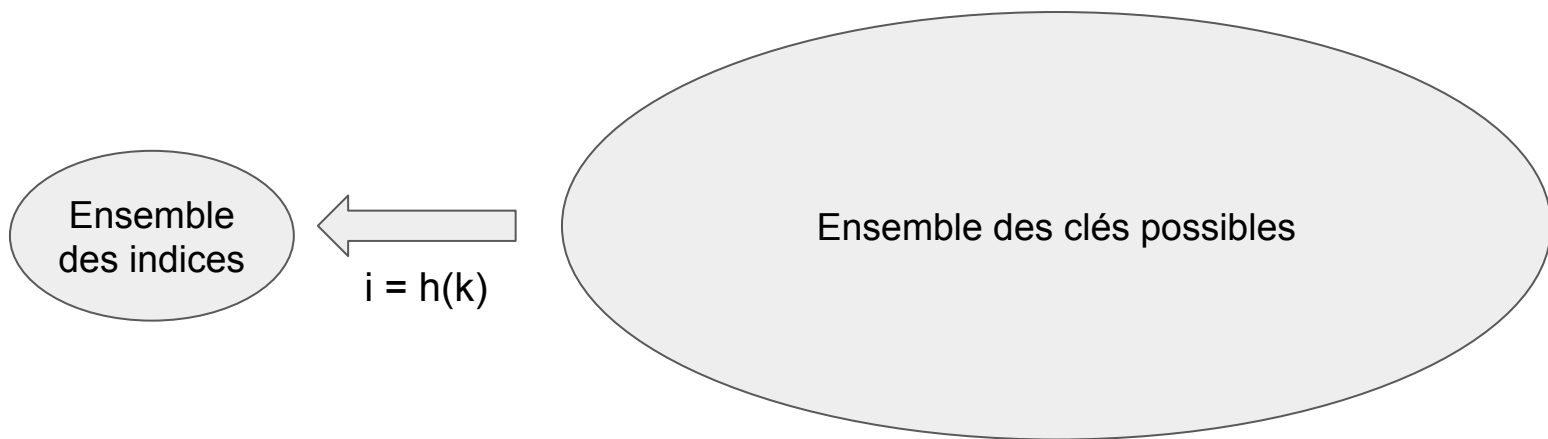
Idée de base :

- 1) Les **valeurs** du dictionnaire sont rangées dans un **tableau**.
- 2) L'**indice** des éléments de ce tableau est **calculé à partir de la clé**.
- 3) L'ensemble des indices possibles peut être restreint comparé à celui des clés.



Tables de hachage | **Fonction de hachage**

- Il nous faut donc appliquer une transformation à toute clé k permettant d'obtenir une valeur i comprise entre 0 et une certaine taille C .
- Cette transformation est réalisée par une **fonction de hachage** $h(k)$.



Tables de hachage | **Fonction de hachage**

Exemple de fonction remplissant partiellement ce rôle : $h(k) = k \bmod C = k \% C$.

L'opération **modulo** C désigne le reste de la division entière par C . On a donc la garantie que le résultat est dans l'intervalle $[0, k]$.

Quelques exemples avec $C = 5$:

$$22 \% 5 = 2, \text{ car } 22 = 4 \cdot 5 + 2$$

$$15 \% 5 = 0, \text{ car } 15 = 3 \cdot 5 + 0$$

$$3 \% 5 = 3, \text{ car } 3 = 0 \cdot 5 + 3$$

$$5678954 \% 5 = 4, \text{ car } 5678954 = 1135790 \cdot 5 + 4$$

Tables de hachage | **Fonction de hachage**

- On peut maintenant ranger les valeurs dans un tableau classique.
- L'indice des éléments est déterminé par la valeur de hachage de leur clé.
- On peut lire et écrire les valeurs en calculant cet indice.
⇒ La complexité d'accès est celle de $h(k)$, donc $O(1)$ en général !

Tableau de valeurs T

<i>indice</i>	<i>valeur</i>
0	∅
1	∅
2	57
3	∅

$h(\text{"jean"}) = 2$
 $\text{age}[\text{"jean"}] = 57$



Tables de hachage | **Le problème des collisions**

- Si l'ensemble des indices est plus petit que l'ensemble des clés possibles, il est **inévitabile** que deux clés potentielles soient associées au même indice. On dit que deux clés menant à la même entrée du tableau de valeurs produisent une **collision**.
- Une bonne fonction de hachage **répartit** au mieux les valeurs, de sorte à éviter au maximum un "empilement" de collisions menant au même endroit du tableau. Peut être optimisé en fonction de la nature des clés.
- Dans le cas général, il nous faut un mécanisme de **gestion des collisions**.

Tables de hachage | **Gestion des collisions (exemple)**

- Prenons la fonction de hachage suivante sur les textes k :
$$h(k) = (k_0 + k_1 + \dots) \% 5.$$
- Exemple : $h(\text{"ciao"}) = (99 + 105 + 97 + 111) \% 5 = 412 \% 5 = 2.$

$$\begin{array}{cccc} \downarrow & \downarrow & \downarrow & \downarrow \\ \text{"c"} & \text{"i"} & \text{"a"} & \text{"o"} \end{array}$$
- Nous allons maintenant associer des textes à des valeurs et voir comment ces valeurs se rangent dans la table de hachage.

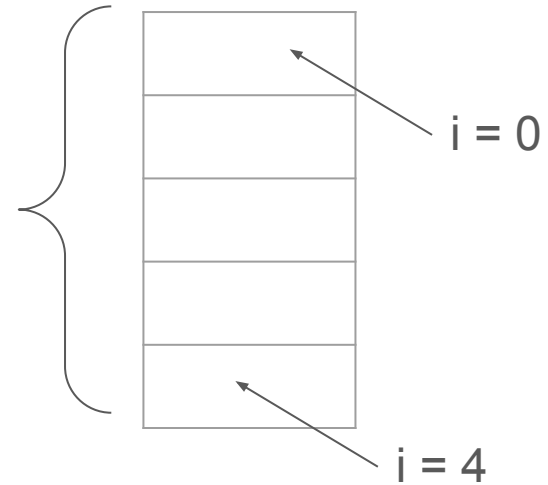
Tables de hachage | **Gestion des collisions**

Associations des prénoms à des âges.

Par exemple, on aimerait associer "sara" à 25 ans.

5 entrées dans le
tableau T en
mémoire centrale

Tableau de valeurs T



Tables de hachage | Gestion des collisions

Exemple : associons un âge à des prénoms.

Chaque **prénom** sera une **clé**.

Chaque **âge** sera la **valeur** associée à cette clé.

Pour chaque clé k , il faut calculer son hash $h(k)$.

Le hash sera utilisé comme indice du tableau.

valeur	k	$h(k)$

↑
âge

↑
prénom

↑
Indice de la cellule où stocker la valeur

Tableau de valeurs T

Tables de hachage | Gestion des collisions

$T["sara"] = 25$

valeur	k	h(k)
25	"sara"	

↑
âge

↑
prénom

T

Tables de hachage | **Gestion des collisions**

$T["sara"] = 25$

valeur	k	h(k)
25	"sara"	3

$$\begin{aligned}h("sara") &= (115+97+114+97)\%5 \\ &= 423\%5 \\ &= \mathbf{3}\end{aligned}$$

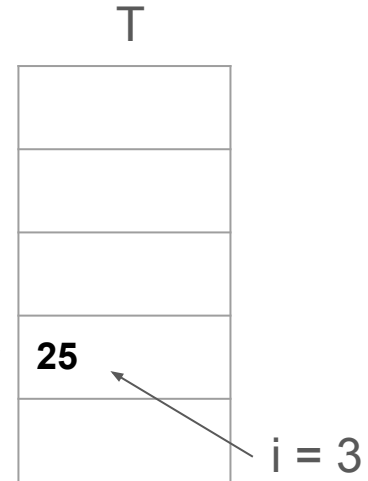
T

Tables de hachage | Gestion des collisions

$T["sara"] = 25$

valeur	k	h(k)
25	"sara"	3

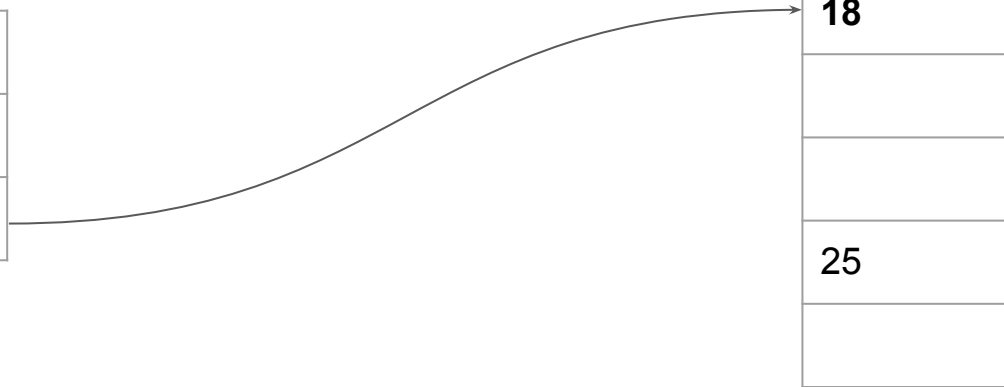
$$\begin{aligned}h("sara") &= (115+97+114+97)\%5 \\ &= 423\%5 \\ &= 3\end{aligned}$$



Tables de hachage | Gestion des collisions

$T["leo"] = 18$

valeur	k	h(k)
25	"sara"	3
18	"leo"	0



Tables de hachage | Gestion des collisions

$T["eva"] = 83$

valeur	k	h(k)
25	"sara"	3
18	"leo"	0
83	"eva"	1



Tables de hachage | Gestion des collisions

$T["zoe"] = 25$

valeur	k	h(k)
25	"sara"	3
18	"leo"	0
83	"eva"	1
25	"zoe"	4

T

18
83
25
25



Tables de hachage | Gestion des collisions

$T["sam"] = 33$

valeur	k	h(k)
25	"sara"	3
18	"leo"	0
83	"eva"	1
25	"zoe"	4
33	"sam"	1

Collision:
 $h("sam") = h("eva")$

!

T

18
83
25
25

Tables de hachage | Gestion des collisions

$T["sam"] = 33$

valeur	k	h(k)
25	"sara"	3
18	"leo"	0
83	"eva"	1
25	"zoe"	4
33	"sam"	1

Plusieurs possibilités :

- Chercher le prochain emplacement libre.
- Rehacher avec $h'(k)$.
- ...

Dans tous les cas, il faut également stocker la clé si l'on veut pouvoir identifier les éléments issus de collisions.

T

18
83
25
25

Tables de hachage | **Adressage ouvert**

valeur	k	h(k)
25	"sara"	3
18	"leo"	0
83	"eva"	1
25	"zoe"	4
33	"sam"	1

Dans tous les cas, il faut également stocker la clé si l'on veut pouvoir identifier les éléments issus de collisions.

T

(18, "leo")
(83, "eva")
(33, "sam")
(25, "sara")
(25, "zoe")

Prochain emplacement libre après $i = 1$

Tables de hachage | **Adressage fermé**

valeur	k	h(k)
25	"sara"	3
18	"leo"	0
83	"eva"	1
25	"zoe"	4
33	"sam"	1

Les éléments de T peuvent aussi être des entrées de listes chaînées.

T

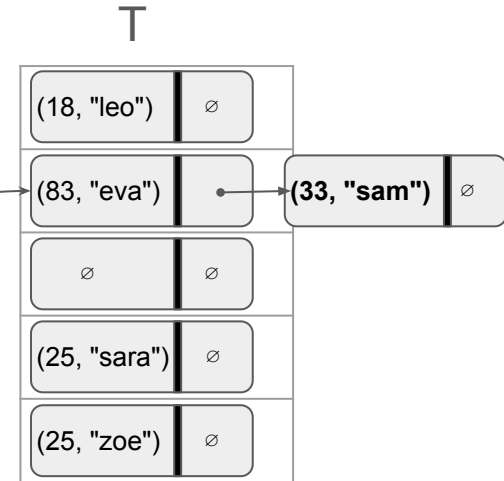
(18, "leo")	∅
(83, "eva")	∅
∅	∅
(25, "sara")	∅
(25, "zoe")	∅

(ou n'importe quelle autre structure qui peut contenir plusieurs éléments)

Tables de hachage | **Adressage fermé**

valeur	k	h(k)
25	"sara"	3
18	"leo"	0
83	"eva"	1
25	"zoe"	4
33	"sam"	1

Les éléments de T peuvent être des entrées de listes chaînées.



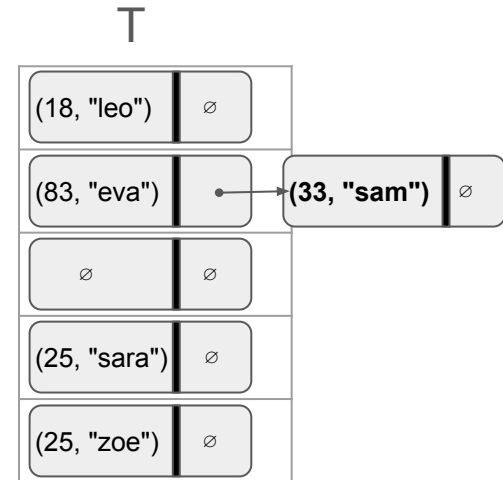
Tables de hachage | Gestion des collisions

Temps d'accès avec adressage fermé comme ouvert :

- Cas de collision est $O(n)$.
- Cas le plus fréquent est $O(1)$.
- Complexité amortie $O(1)$.

Cas d'utilisation :

- Adressage ouvert idéal lorsque peu de collisions sont à prévoir, mais devient handicapant si trop de collisions surviennent dans une zone de T .
- Adressage fermé à privilégier si les collisions sont nombreuses.



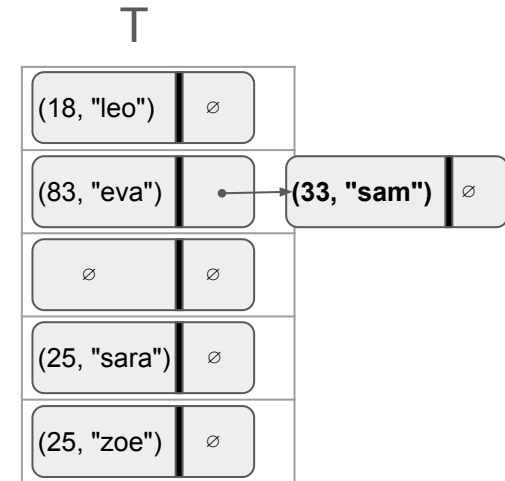
Tables de hachage | Exemple d'accès (adressage fermé)

L'utilisateur veut obtenir la valeur associée à la clé "x".

Code de l'utilisateur : `age_de_x = T["x"]`

Dans ce cas, la fonction qui implémente l'accès aux éléments de la table doit :

1. calculer $i = h("x")$.
2. accéder à la liste chaînée $L = T[i]$.
3. si L est vide, alors la clé n'est associée à rien.
4. sinon, parcourir L et retourner l'élément associé à "x".





Tables de hachage | Résumé

- Constitue une façon d'**implémenter un dictionnaire**.
- Permet d'associer des **clés** hachables à des **valeurs**.
- En moyenne, temps d'accès aux éléments en $O(1)$ tout en nécessitant un espace mémoire en $O(n)$.
- En pratique plus lent qu'un tableau, mais plus général.

Quelques structures de données



Tableau statique



Accès en $O(1)$	Nombre fixe d'éléments
Compact en mémoire	



Tableau dynamique



Accès en $O(1)$	Mal adapté (lent) si la taille change fréquemment
Nombre variable d'éléments	Potentiel gaspillage de mémoire



Liste chaînée



Nombre variable d'éléments	Accès en $O(n)$
Adaptée si la taille change fréquemment	Potentiel gaspillage de mémoire
Adaptée pour l'insertion à des indices arbitraires	



Table de hachage



Accès, insertions, suppressions en $O(1)$ en moyenne	Potentielles collisions faisant chuter l'efficacité
Nombre variable d'éléments	Potentiel gaspillage de mémoire
Permet des associations clés-valeur	

Tables de hachage | **Facteur de charge**

- Il est à noter qu'en pratique, on peut **redimensionner** la table (comme un tableau dynamique) de sorte à générer moins de collisions et à avoir un peu de **marge** en cas de collision.
- Exemple d'heuristique : si T est rempli à plus de $3/4$, on génère une nouvelle version agrandie d'un facteur 2.
- Le taux de remplissage de T est nommé le **facteur de charge** de la table de hachage.
- En fait, la table de hachage peut être vue comme un compromis entre les deux approches naïves considérées plus tôt : tableau surdimensionné et liste de clés-valeurs.





Tables de hachage | **Fonction de hachage - commentaires**

- Le modulo seul ne constitue pas nécessairement une bonne fonction de hachage.
- Une fonction de hachage ne **préserve pas forcément l'ordre** des indices par rapport à celui des clés.
- Une bonne fonction de hachage permet de calculer le hash **rapidement** à partir de la clé.
- Une bonne fonction de hachage permet d'éviter au maximum des **collisions**.

Tables de hachage | **Fonction de hachage - commentaires**

- Pour simplifier, on a pris $h(k) = \text{mod}(k, C)$. En réalité les fonctions de hachage sont plus complexes afin de mieux répartir les valeurs (*pseudo-randomness*) : deux clés très légèrement différentes peuvent donner lieu à des hash complètement différents.
- Dans la littérature, on distingue en général le hash de la clé de l'indice auquel elle correspond : la dernière étape, qui fait passer de l'un à l'autre, est justement l'application du modulo.



Tables de hachage | **Lien avec le domaine de la sécurité**

- Certaines fonctions de hachage ont la propriété qu'il est extrêmement difficile, connaissant le hash, de retrouver la clé d'origine.
- Si par ailleurs de telles fonctions impliquent très peu de collisions et qu'elles renvoient des résultats homogènes, alors elles sont utiles en cryptologie, par exemple pour l'**authentification** des utilisateurs d'un site.



Tables de hachage | **Exemple en Python**

Cf. démonstration en cours et **diapos des TP**

```
ages = {"sara":25, "leo":18, "eva":83}  
ages["l  a"] = 19 #ajout d'un   l  ment apr  s-coup  
print("L'  ge d'Eva vaut", ages["eva"])
```



Tables de hachage | **Exemple en Python**

Cf. démonstration en cours et **diapos des TP**

```
echiquier = {"e1":"Roi", "h2":"Pion", "g1":"Cavalier"}  
cle = input("Quelle case voulez-vous examiner?")  
. . . # (ne reste plus qu'à traiter la valeur associée)
```

D'autres structures de données

- On peut concevoir des structures de données autant que nécessaire afin de résoudre un problème.
- Par exemple : structures de données adaptées pour le traitement informatique des graphes.
- Avant cela, nous allons revenir aux algorithmes de tri.