Informatique tronc commun — MP*

Structures de données et dictionnaires

Pourquoi `append` et `k in d` coûtent $O(1)$

Introduction

Le programme

- **1re année** : dictionnaires en « boîte noire », choix **éclairé** liste / dictionnaire
- **2e année** : **fonctionnement** : hachage, clés immuables
- annexe : opérations exigibles, `insert`, `pop(i)`, `del`, `index`… seulement si l'énoncé les introduit

Le programme : première année

Thèmes	**Exemples d'activité, au choix du professeur et non exigibles des étudiants. *Commentaires.***
Recherche séquentielle dans un tableau unidimensionnel. Dictionnaire.	Recherche d'un élément. Recherche du maximum, du second maximum. Comptage des éléments d'un tableau à l'aide d'un dictionnaire. *Manipulations élémentaires d'un tableau unidimensionnel. Utilisation de dictionnaires en boîte noire. Notions de coût constant, de coût linéaire.*

Notions	**Commentaires**
Explicitation et justification des choix de conception ou programmation.	Les parties complexes de codes ou d'algorithmes font l'objet de commentaires qui l'éclairent en évitant la paraphrase. Le choix des collections employées (par exemple, liste ou dictionnaire) est un choix éclairé.

Le programme : deuxième année

3.2 Dictionnaires et programmation dynamique

Les dictionnaires sont utilisés en boîte noire dès la première année ; les principes de leur fonctionnement sont présentés en deuxième année. Ils peuvent être utilisés afin de mettre en mémoire des résultats intermédiaires quand on implémente une stratégie d'optimisation par programmation dynamique.

Notions	**Commentaires**
Dictionnaires, clés et valeurs.	On présente les principes du hachage, et les limitations qui en découlent sur le domaine des clés utilisables.
Usage des dictionnaires en programmation Python.	Syntaxe pour l'écriture des dictionnaires. Parcours d'un dictionnaire.

Le programme : annexe (opérations exigibles)

Types structurés

— Structures indicées immuables (chaînes, tuples) : `len`, accès par indice positif valide, concaténation `+`, répétition `*`, tranche.
— Listes : création par compréhension `[`*e* **`for`** *x* **`in`** *s*`]`, par `[`*e*`]` **`*`** *n*, par `append` successifs ; `len`, accès par indice positif valide ; concaténation `+`, extraction de tranche, copie (y compris son caractère superficiel) ; `pop` en dernière position.
— Dictionnaires : création `{`c_1 `:` v_1`,` … `,` c_n `:` v_n`}`, accès, insertion, présence d'une clé *k* **`in`** *d*, `len`, `copy`.

Structures de contrôle

— Instruction d'affectation avec `=`. Dépaquetage de tuples.
— Instruction conditionnelle : **`if`**, **`elif`**, **`else`**.
— Boucle **`while`** (sans **`else`**). **`break`**, **`return`** dans un corps de boucle.
— Boucle **`for`** (sans **`else`**) et itération sur `range(`*a*`,` *b*`)`, une chaîne, un tuple, une liste, un dictionnaire au travers des méthodes `keys` et `items`.
— Définition d'une fonction **`def`** *f*`(`p_1`,` … `,` p_n`)`, **`return`**.

Aux concours

- **choisir** un dictionnaire pour la bonne complexité
- **raisonner** avec les coûts des listes (rappelés, ou restreints, en préambule)
- **copier** correctement une liste

Rappels concernant le langage `Python`. Ce sujet utilise les types Python listes et dictionnaires, mais seules les opérations mentionnées ci-dessous sont autorisées dans vos réponses. Quand une complexité est indiquée avec un symbole $(*)$, cela signifie que nous faisons une hypothèse simplificatrice sur sa complexité. La justification de cette simplification est hors-programme.

Si `d` est un dictionnaire `Python` :

- `{key_1: v_1, ..., key_n: v_n}` crée un nouveau dictionnaire en associant chaque valeur `v_i` à une clé `key_i`. Complexité en $\mathcal{O}(n)$ $(*)$.
- `d[key]` renvoie la valeur associée à la clé `key` dans `d` et lève une erreur si la clé `key` n'est pas présente. Complexité en $\mathcal{O}(1)$ $(*)$.
- `d[key] = v` modifie `d` pour associer la valeur `v` à la clé `key`, même si la clé `key` n'est pas présente dans `d` initialement. Complexité en $\mathcal{O}(1)$ $(*)$.
- `key in d` teste si la clé `key` est présente dans `d`. Complexité en $\mathcal{O}(1)$ $(*)$.

Aux concours : coûts rappelés en préambule

Rappels sur Python. L'utilisation de toute fonction Python sur les listes ou sur les dictionnaires autre que celles mentionnées dans ce paragraphe **est interdite**. Sur les **listes**, on autorise les opérations suivantes, *dont la complexité est en* $O(1)$:

- `len(l)` renvoie la longueur de la liste `l`.
- `l[i]` désigne l'élément d'indice `i` de la liste `l`, pour $0 \leq$ `i` $<$ `len(l)`.
- `l.append(e)` ajoute en place l'élément `e` à la fin de la liste `l`.

Les opérations suivantes sont également autorisées et leur *complexité est en* $O(n)$:

- `l1 + l2` renvoie une nouvelle liste (de longueur n) qui est la concaténation des listes `l1` et `l2`.
- `range(n)` renvoie la liste `[0,1, ..., n-1]`.
- `range(n-1,-1,-1)` renvoie la liste `[n-1, ..., 0]`.
- `l.pop(0)` retire le premier élément `e` de la liste `l` (de longueur n) et renvoie `e`.
- `(e in l)` renvoie `True` si l'élément `e` est dans la liste `l` (de longueur n), et `False` sinon.
- `l[:]` renvoie une *copie* de la liste `l` (de longueur n).

On autorise également les constructions suivantes :

- La *construction* `for e in l` parcourt (itère sur) les éléments de la liste `l` du premier élément (d'indice 0), au dernier élément (d'indice `len(l)-1`) avec la complexité $O($`len(l)`$)$.
- La *construction* `[f(e) for e in l]` produit la même liste que le code suivant, et avec la complexité `len(l)` fois la complexité de `f` :

```
result = []
for e in l:
  result.append(f(e))
return result
```

Sur les **dictionnaires**, on autorise uniquement les opérations suivantes, dont *la complexité est en* $O(1)$:

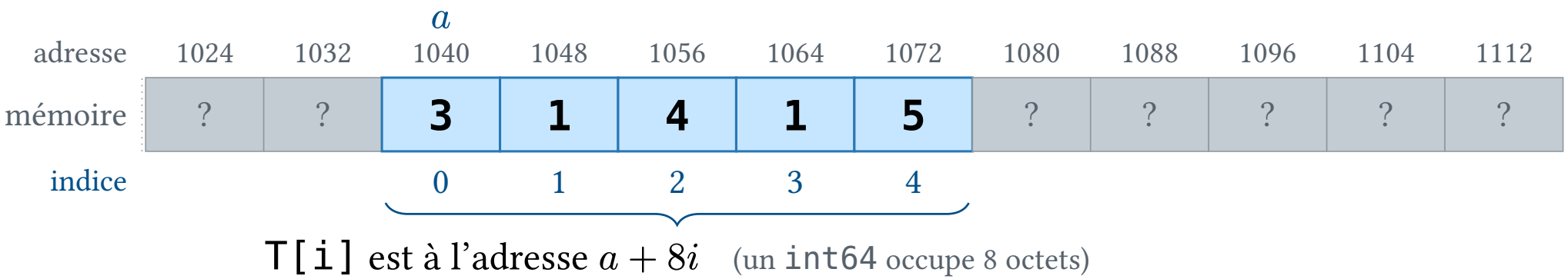
- Le test `(e in d)` renvoie `True` si `e` est une clé du dictionnaire `d`, et `False` sinon.
- L'accès `d[e]` à l'élément associé à la clé `e` dans le dictionnaire `d`.

On autorise également les constructions suivantes sur les dictionnaires :

- La *construction* `for (k,v) in d` parcourt les éléments du dictionnaire `d`.
- La *construction* `d[k] = v` affecte la valeur `v` à la clé `k`.

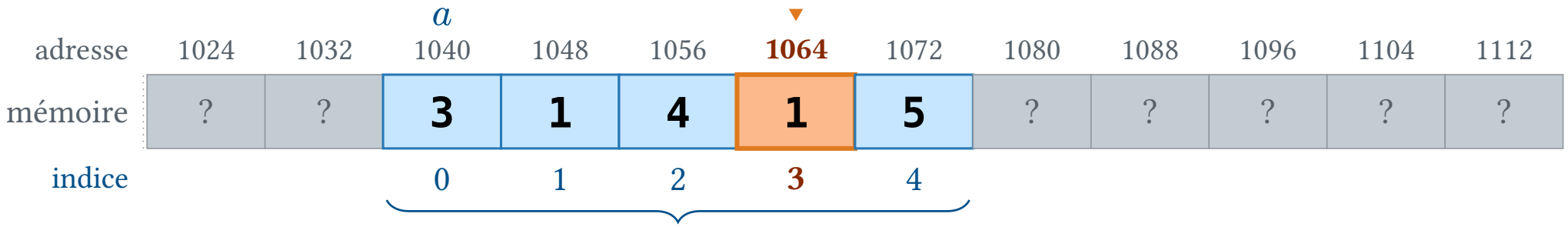
Structures de données et leurs coûts

La mémoire et les tableaux



- zone **contiguë**, de taille **fixe**, éléments **de même type** (taille t) : `T[i]` à l'adresse $a + it$
- accès, modification : $O(1)$, on ne peut pas « ajouter » : nouveau tableau + recopie, $O(n)$

La mémoire et les tableaux



`T[i]` est à l'adresse $a + 8i$ (un `int64` occupe 8 octets)

- **`T[3]`** : $1040 + 8 \times 3 = 1064$

- zone **contiguë**, de taille **fixe**, éléments **de même type** (taille t) : `T[i]` à l'adresse $a + it$
- accès, modification : $O(1)$, on ne peut pas « ajouter » : nouveau tableau + recopie, $O(n)$

Les tableaux : coûts

Opération (taille n)	**Coût**	**Pourquoi**
création	$O(n)$	réservation et initialisation des n cases
accès `T[i]`	$O(1)$	calcul d'adresse, puis lecture
modification `T[i] = x`	$O(1)$	calcul d'adresse, puis écriture
recherche d'une valeur	$O(n)$	parcours
ajout d'un élément	impossible	taille fixe : nouveau tableau + recopie, $O(n)$

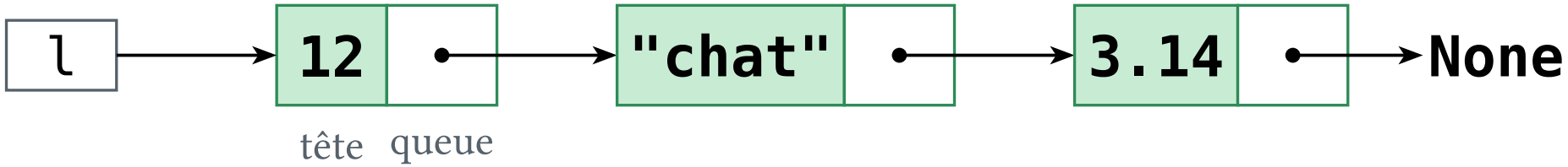
Les tableaux `numpy` (Python)

- `ndarray` = ce modèle : bloc contigu, `dtype` unique, taille fixe
- opérations **vectorisées** (toujours $O(n)$), tranche `T[1:4]` = **vue** (listes : **copie**)

```
T = np.array([3, 1, 4, 1, 5])
T * 2  # array([6, 2, 8, 2, 10])
V = T[1:4]
V[0] = 100
T  # array([3, 100, 4, 1, 5])
```

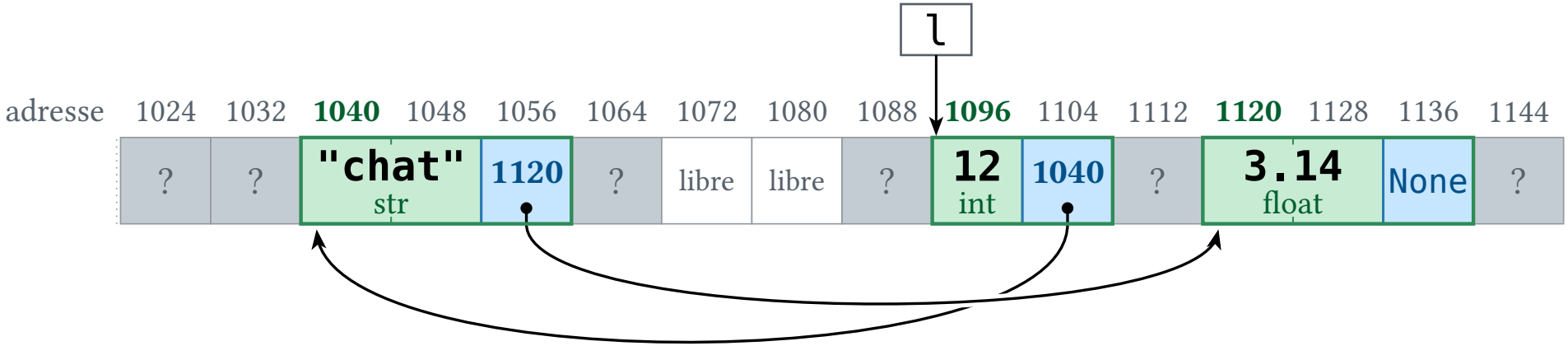
```
L = [3, 1, 4, 1, 5]
L * 2  # [3, 1, 4, 1, 5, 3, 1, …]
W = L[1:4]        # copie
W[0] = 100
L  # [3, 1, 4, 1, 5]
```

Les listes chaînées



- **pas de taille fixe** : chaque élément est rangé dans une **cellule**, placée n'importe où en mémoire
- cellule = la **valeur** (la *tête*) + l'**adresse** de la cellule suivante (la *queue*)
- une valeur spéciale (`None`) marque la fin de la liste

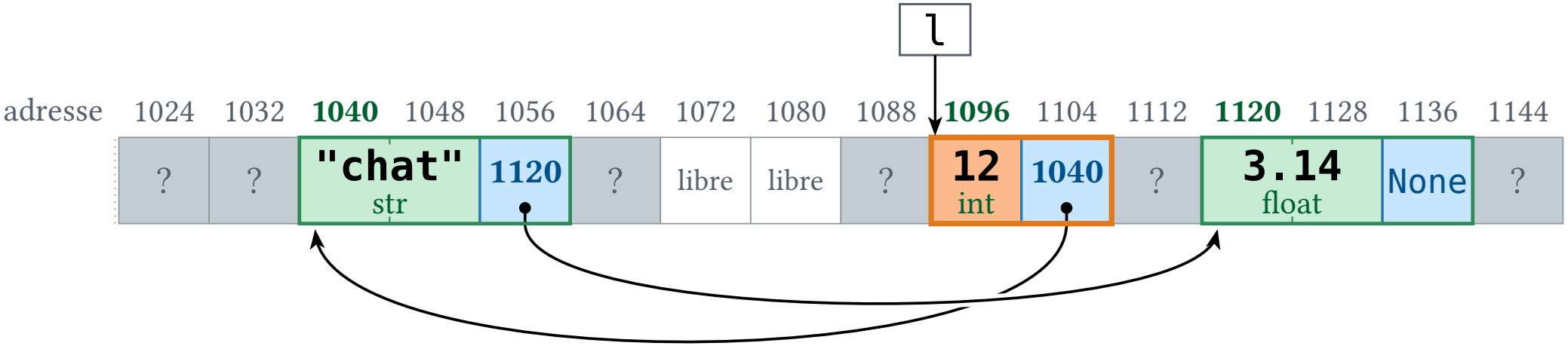
Une liste chaînée dans la mémoire



une cellule = la **valeur** (vert) + l'**adresse** de la cellule suivante (bleu)

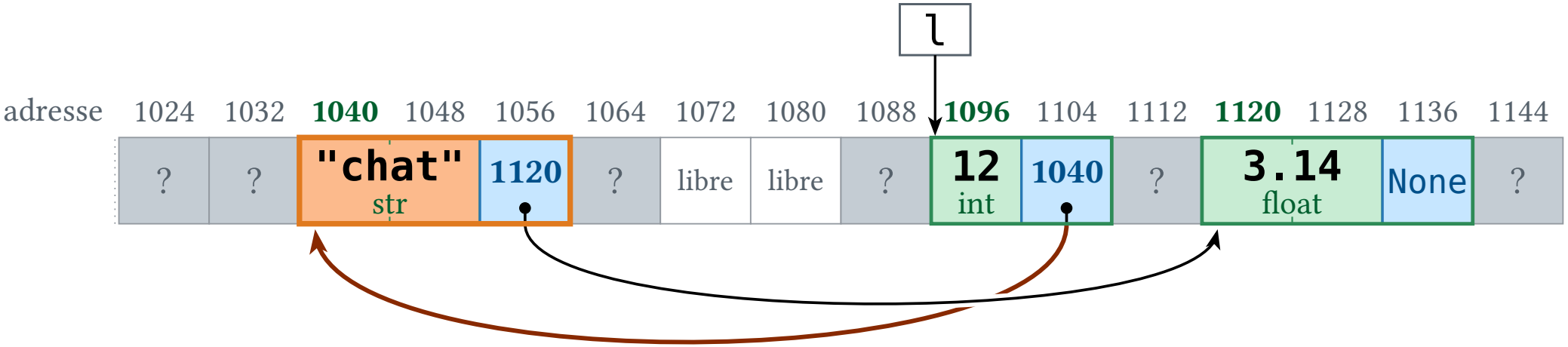
- cellules **dispersées**, dans n'importe quel ordre, de tailles différentes
- l'ordre de la liste est donné par les **adresses** : accès au i-ème = suivre i adresses

Lecture : `acces(l, 2)`



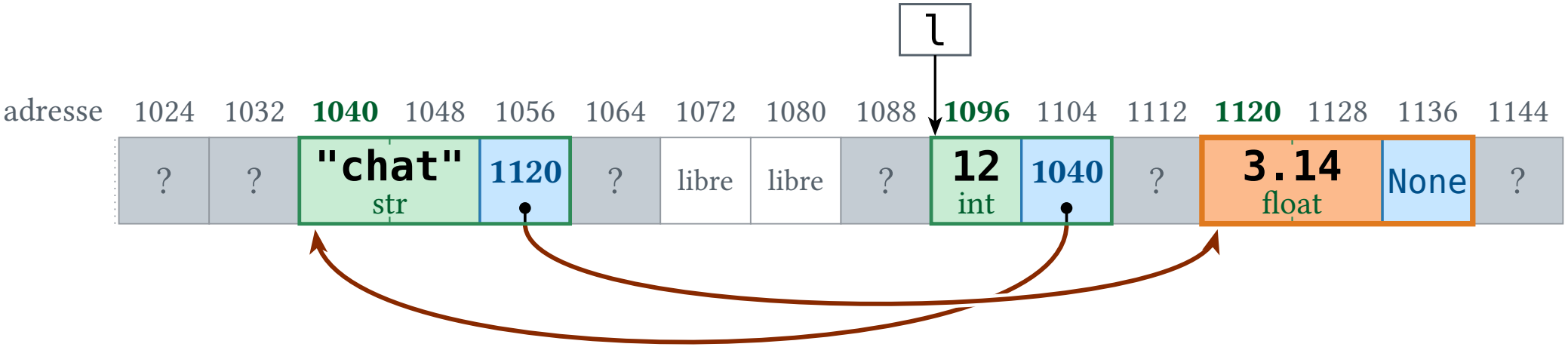
1. `acces(l, 2)` : `l` contient 1096, la cellule d'indice 0 (valeur `12`)

Lecture : `acces(l, 2)`



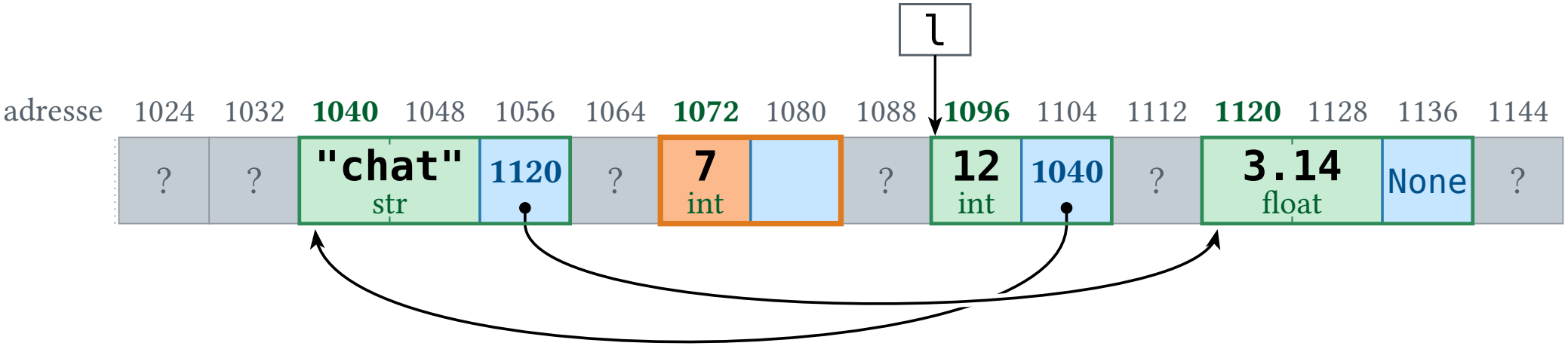
1. `acces(l, 2)` : `l` contient 1096, la cellule d'indice 0 (valeur `12`)
2. on lit son champ suivant, 1040 : cellule d'indice 1 (valeur `"chat"`)

Lecture : `acces(l, 2)`



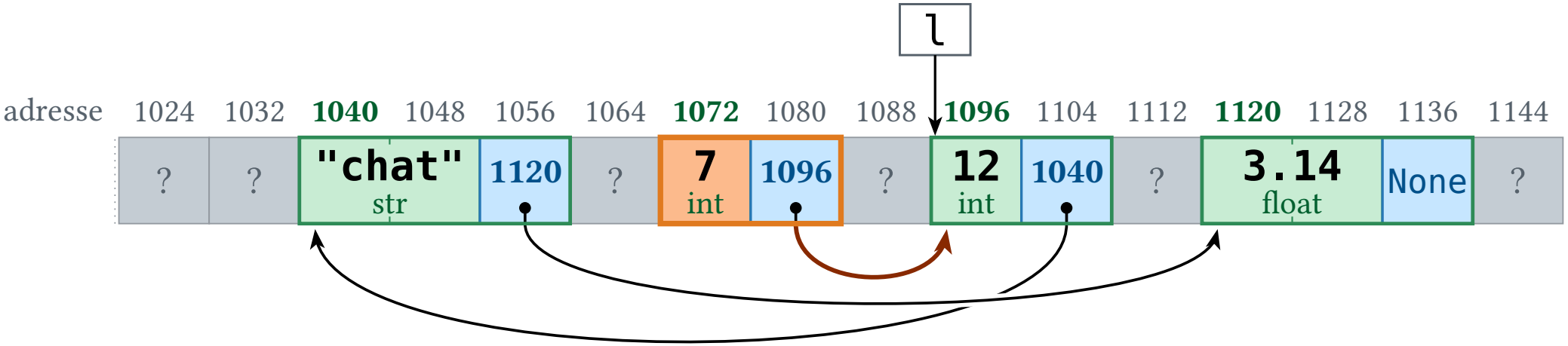
1. `acces(l, 2)` : `l` contient 1096, la cellule d'indice 0 (valeur `12`)
2. on lit son champ suivant, 1040 : cellule d'indice 1 (valeur `"chat"`)
3. on lit 1120 : cellule d'indice 2, on renvoie ▸ **`3.14`**, 2 sauts : $O(i)$

Ajout en tête : `ajout_tete(l, 7)`



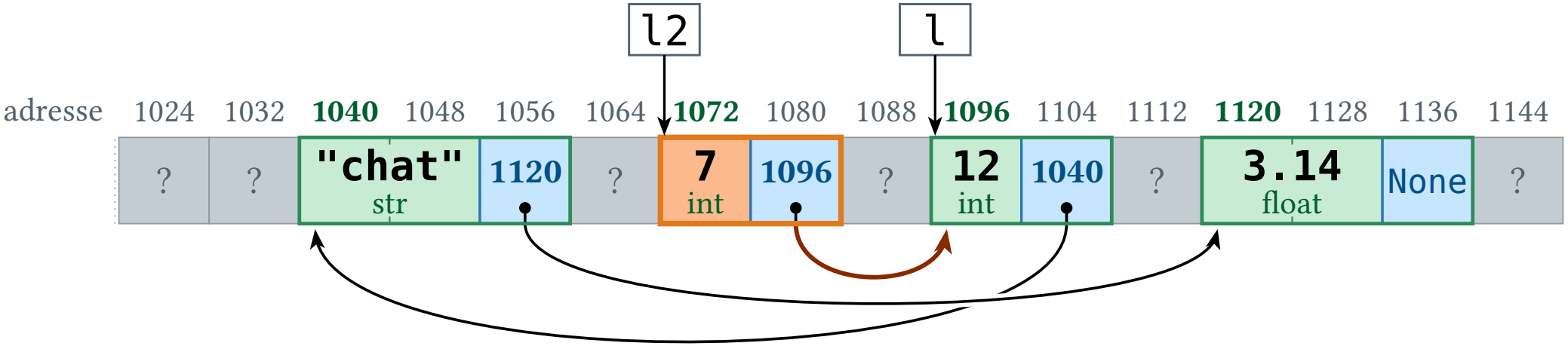
1. `ajout_tete(l, 7)` : on crée une cellule dans des cases libres (1072) et on y écrit `7`

Ajout en tête : `ajout_tete(l, 7)`



1. `ajout_tete(l, 7)` : on crée une cellule dans des cases libres (1072) et on y écrit `7`
2. son champ suivant reçoit l'adresse de la tête de `l` : 1096

Ajout en tête : `ajout_tete(l, 7)`

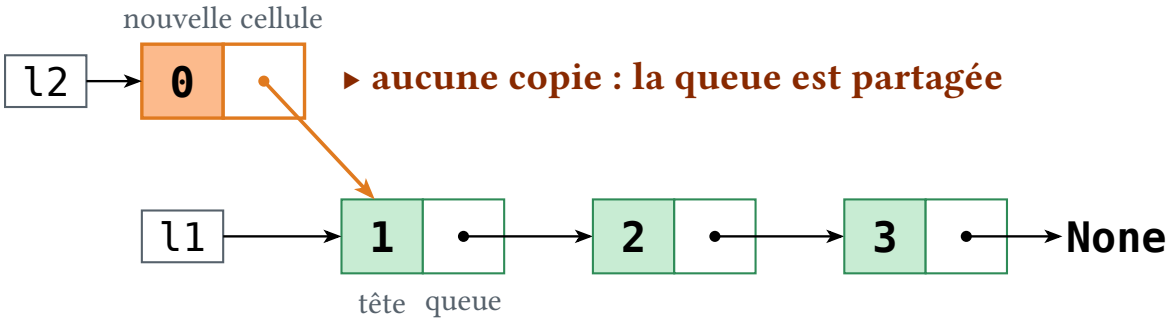


1. `ajout_tete(l, 7)` : on crée une cellule dans des cases libres (1072) et on y écrit `7`
2. son champ suivant reçoit l'adresse de la tête de `l` : 1096
3. `l2` contient 1072, **aucune cellule recopiée : $O(1)$**, `l` est inchangée

Les listes chaînées en Python (Python)

```
l1 = (1, (2, (3, None)))  # 1, 2, 3
l1[0]     # 1 : la tête
l1[1]     # (2, (3, None)) : la queue
l2 = (0, l1)   # ajout de 0 en tête
```

- Python n'en fournit pas : on les **modélise**
- liste vide : `None`
- cellule : couple `(valeur, suivant)`



- un tuple est **immuable** : une cellule n'est jamais modifiée, le partage de la queue est sans danger

Question : accès et ajouts

Question (poly § 2.2) Écrire les fonctions :

- `acces(l, i)`, renvoyant l'élément d'indice `i` de la liste chaînée `l` (et `None` si `i` n'est pas un indice valide),
- `ajout_tete(l, x)`, renvoyant la liste obtenue en ajoutant `x` en tête,
- `ajout_fin(l, x)`, renvoyant la liste obtenue en ajoutant `x` en fin.

Quelles sont les complexités de ces fonctions en fonction de la longueur n de la liste ?

Question : accès et ajouts (correction)

```python
def acces(l, i):   # O(i)
    if i < 0:
        return None
    while l is not None and i > 0:
        l = l[1]   # lien suivant
        i = i - 1
    if l is None:  # i trop grand
        return None
    return l[0]
```

```python
def ajout_tete(l, x):  # O(1)
    return (x, l)      # partage

def ajout_fin(l, x):   # O(n)
    if l is None:
        return (x, None)
    return (l[0], ajout_fin(l[1], x))
```

`ajout_fin` recopie les n cellules : la liste de départ n'est pas modifiée.

Question : accès et ajouts (correction)

```python
def acces(l, i):   # O(i)
    if i < 0:
        return None
    while l is not None and i > 0:
        l = l[1]   # lien suivant
        i = i - 1
    if l is None:  # i trop grand
        return None
    return l[0]
```

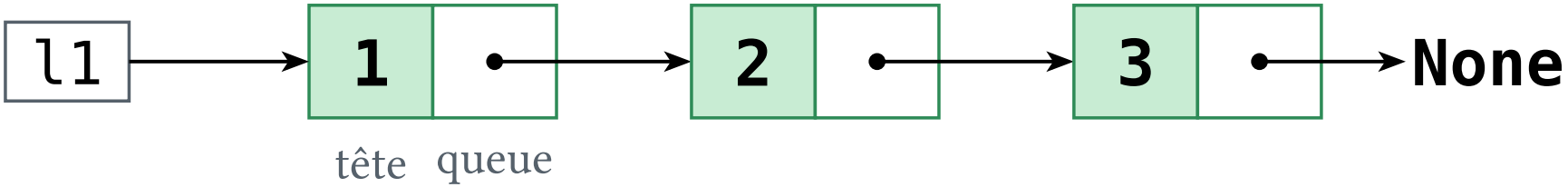
```python
def ajout_tete(l, x):  # O(1)
    return (x, l)      # partage

def ajout_fin(l, x):   # O(n)
    if l is None:
        return (x, None)
    return (l[0], ajout_fin(l[1], x))
```

`ajout_fin` recopie les n cellules : la liste de départ n'est pas modifiée.

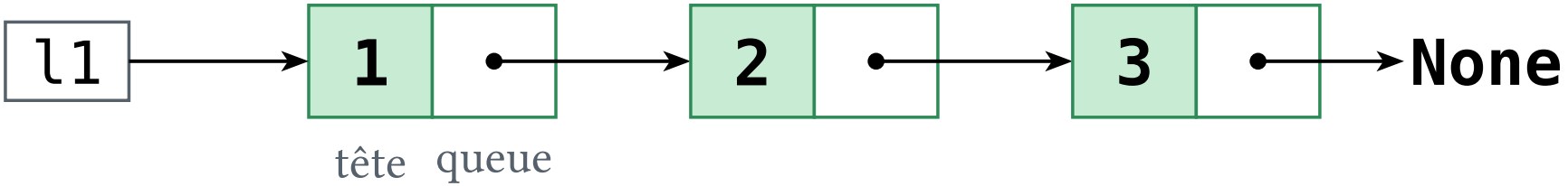
`acces` : $O(i)$ `ajout_tete` : $O(1)$ (queue partagée) `ajout_fin` : $O(n)$ (tout est recopié)

Question : insertion



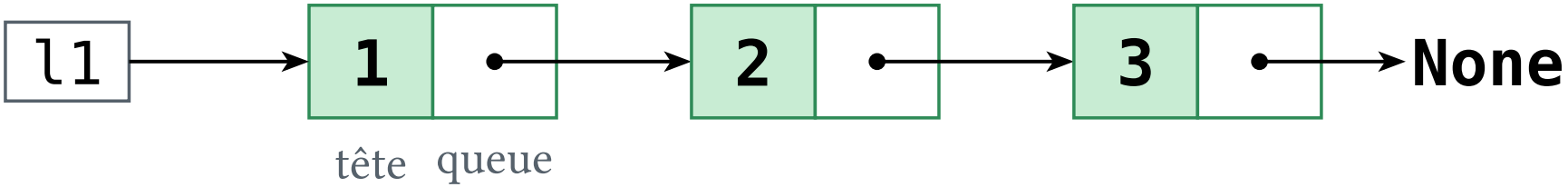
Question (poly § 2.2) Écrire `inserer(l, i, x)`, prenant en argument une liste chaînée `l`, un indice valide `i` ($0 \leq i \leq$ longueur) et un élément `x`, et renvoyant la liste obtenue en insérant `x` de sorte qu'il devienne l'élément d'indice i. Quelle est sa complexité ?

Question : insertion (correction)




```
def inserer(l, i, x):
    if i == 0:
        return (x, l)
    return (l[0], inserer(l[1], i - 1, x))
```

Question : insertion (correction)




```python
def inserer(l, i, x):
    if i == 0:
        return (x, l)
    return (l[0], inserer(l[1], i - 1, x))
```

$O(i)$: les i premières cellules sont recopiées, la suite est partagée

Question : concaténation

```
def concatener(l1, l2):
    if l1 is None:
        return l2                    # l2 est partagée, pas recopiée
    return (l1[0], concatener(l1[1], l2))
```

Question (poly § 2.2) Quelle est la complexité de `concatener(l1, l2)` en fonction des longueurs n_1 et n_2 de `l1` et `l2` ?

Question : concaténation

```
def concatener(l1, l2):
    if l1 is None:
        return l2                    # l2 est partagée, pas recopiée
    return (l1[0], concatener(l1[1], l2))
```

Question (poly § 2.2) Quelle est la complexité de `concatener(l1, l2)` en fonction des longueurs n_1 et n_2 de `l1` et `l2` ?

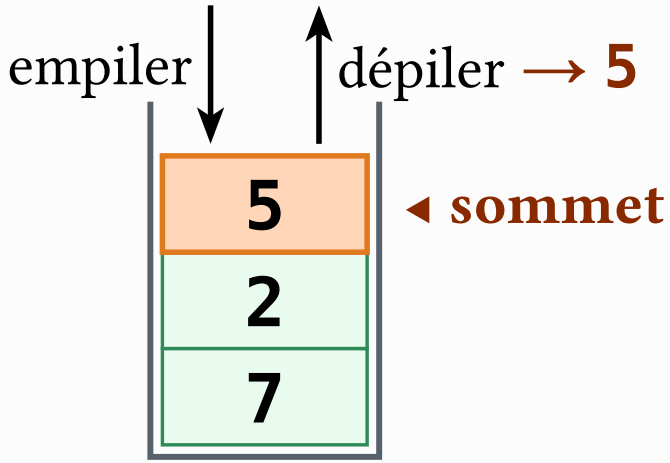
$O(n_1)$: les cellules de `l1` sont recopiées, `l2` est partagée, ni parcourue ni recopiée

Les listes chaînées : coûts

Opération (longueur n)	**Coût**	**Pourquoi**
accès au i-ème	$O(i)$	on suit i liens
ajout, suppression en tête	$O(1)$	une cellule créée ou oubliée
insertion à l'indice i	$O(i)$	on recopie les i premières cellules
ajout en fin, concaténation	$O(n)$	on recopie toutes les cellules (de `l1`)
longueur, recherche d'une valeur	$O(n)$	parcours

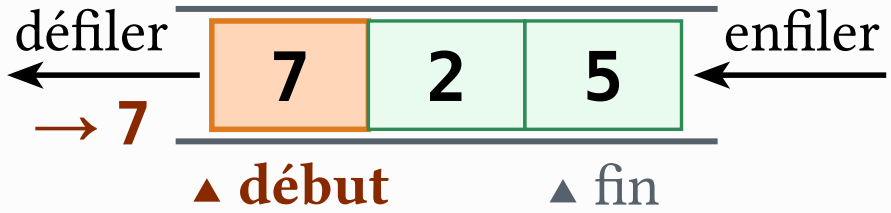
Rappel : piles, files et deque

(a) pile (LIFO)



dernier entré, premier sorti

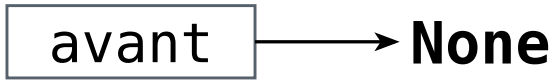
(b) file (FIFO)



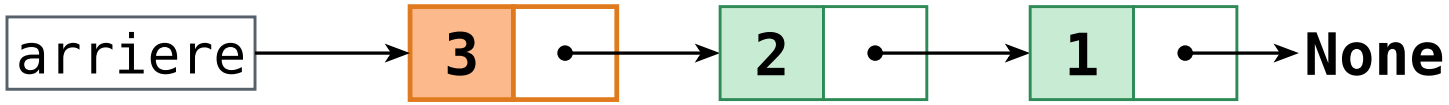
premier entré, premier sorti

- **pile** (*LIFO*) : on **empile** et on **dépile** au sommet, une liste chaînée est une pile naturelle
- **file** (*FIFO*) : on **enfile** à la fin, on **défile** au début

Une file avec deux listes immuables



on **défile** en tête

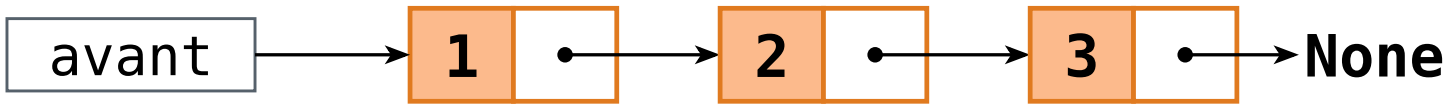


on **enfile** en tête

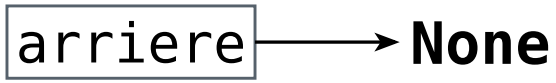
1. enfiler 1, 2, 3 : ajouts **en tête** de `arriere`, en $O(1)$

   chaque élément est retourné au plus une fois : enfiler et défiler en $O(1)$ **amorti**

Une file avec deux listes immuables



on **défile** en tête

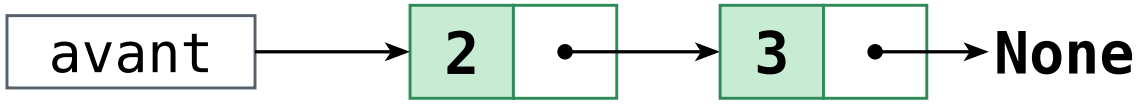


on **enfile** en tête

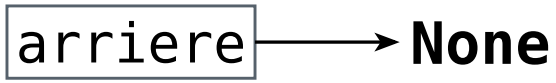
2. défiler, `avant` est **vide** : on y met `arriere` **retournée** ($O(n)$), puis…

chaque élément est retourné au plus une fois : enfiler et défiler en $O(1)$ **amorti**

Une file avec deux listes immuables



on **défile** en tête

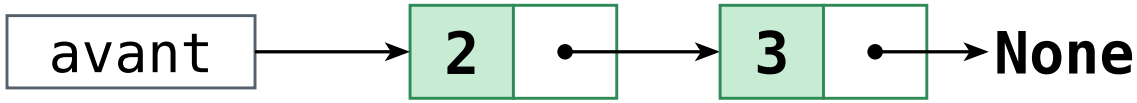


on **enfile** en tête

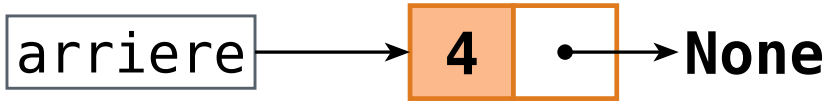
3. … on renvoie la tête de `avant` : `1`

chaque élément est retourné au plus une fois : enfiler et défiler en $O(1)$ **amorti**

Une file avec deux listes immuables



on **défile** en tête

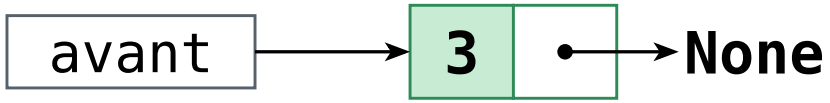


on **enfile** en tête

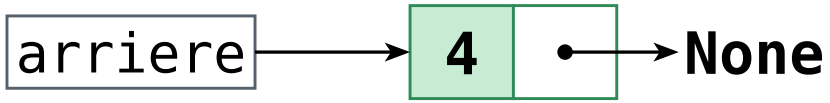
4. enfiler 4 : ajout en tête de `arriere`, en $O(1)$

chaque élément est retourné au plus une fois : enfiler et défiler en $O(1)$ **amorti**

Une file avec deux listes immuables



on **défile** en tête



on **enfile** en tête

5. défiler, `avant` n'est **pas vide** : on renvoie `2` directement, en $O(1)$

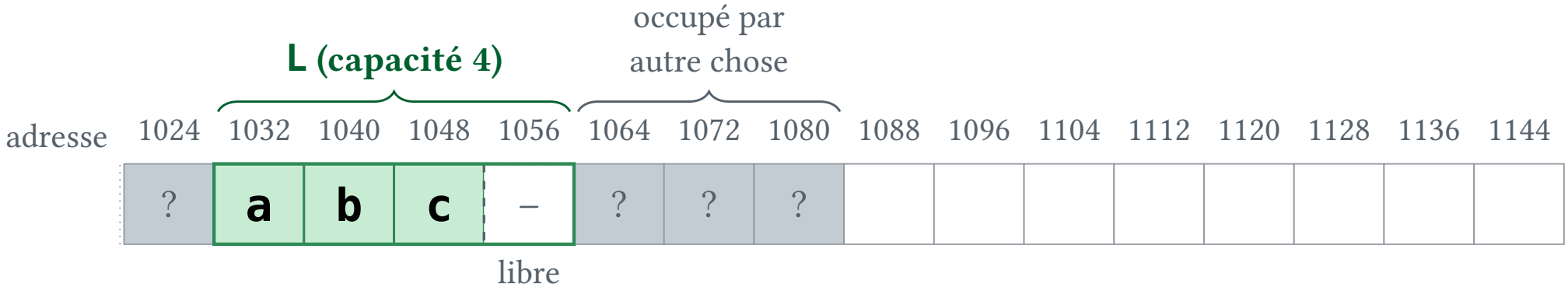
chaque élément est retourné au plus une fois : enfiler et défiler en $O(1)$ **amorti**

Piles, files : coûts (Python)

Structure	Opération	Liste `L`	Coût	`deque d`	Coût
pile	empiler	`L.append(x)`	$O(1)$ amorti	`d.append(x)`	$O(1)$
pile	dépiler	`L.pop()`	$O(1)$ amorti	`d.pop()`	$O(1)$
file	enfiler	`L.append(x)`	$O(1)$ amorti	`d.append(x)`	$O(1)$
file	défiler	`L.pop(0)`	$O(n)$!	`d.popleft()`	$O(1)$

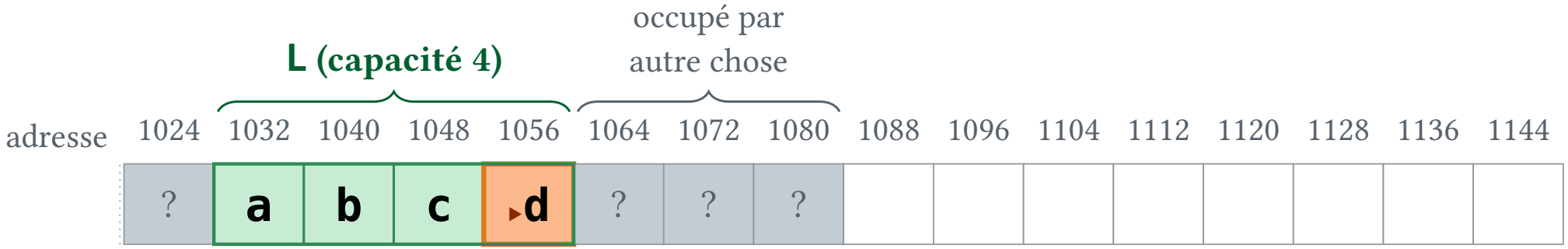
```
from collections import deque
d = deque()
```

Les tableaux dynamiques



- **taille** $n \leq$ **capacité** c, si $n < c$: ajout en $O(1)$
- si $n = c$: **réallocation** + recopie, $O(n)$

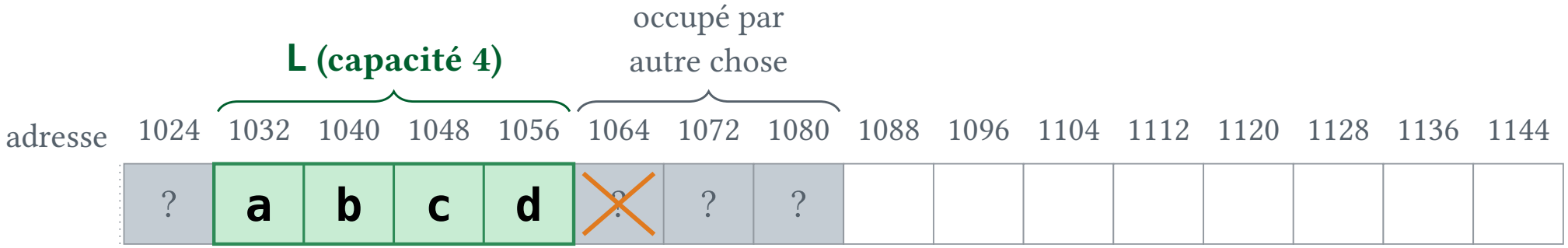
Les tableaux dynamiques



1. `L.append(d)` : on écrit dans la case réservée : $O(1)$

- **taille** $n \leq$ **capacité** c, si $n < c$: ajout en $O(1)$
- si $n = c$: **réallocation** + recopie, $O(n)$

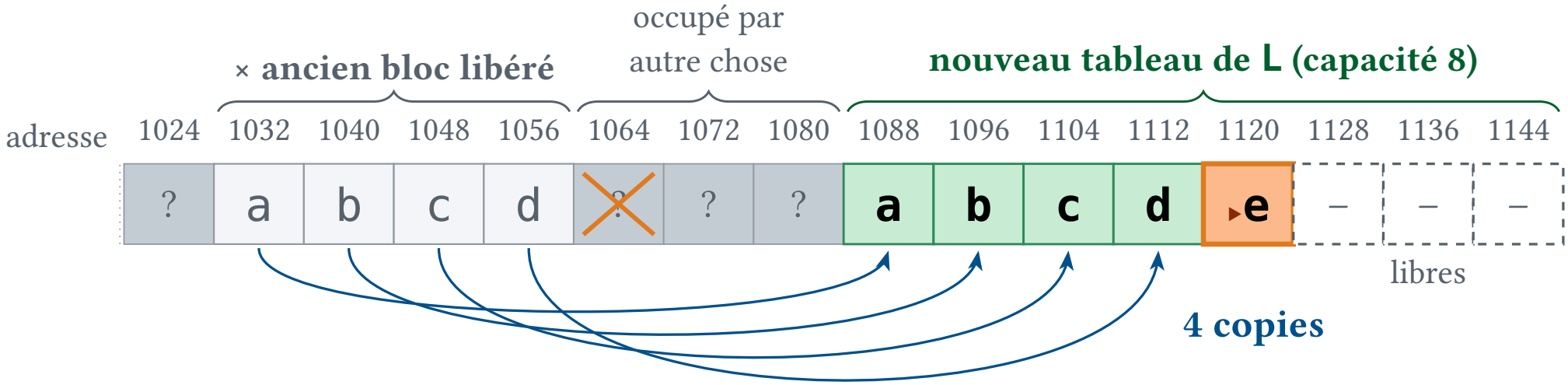
Les tableaux dynamiques



1. `L.append(d)` : on écrit dans la case réservée : $O(1)$
2. `L.append(e)` : la case suivante (1064) est occupée × : impossible de grandir sur place

- **taille** $n \leq$ **capacité** c, si $n < c$: ajout en $O(1)$
- si $n = c$: **réallocation** + recopie, $O(n)$

Les tableaux dynamiques



1. `L.append(d)` : on écrit dans la case réservée : $O(1)$
2. `L.append(e)` : la case suivante (1064) est occupée × : impossible de grandir sur place
3. nouveau bloc de capacité 8 : 4 copies, puis ▸ **écriture de `e`**, l'ancien bloc est libéré

- **taille $n \leq$ capacité c**, si $n < c$: ajout en $O(1)$
- si $n = c$: **réallocation** + recopie, $O(n)$

Question : quelle nouvelle capacité ?

$$C(N) = N + \sum_{j\,:\,c_j < N} c_j$$

Question (poly § 2.3) Calculer $C(N)$ dans les deux cas suivants :

1. on agrandit le tableau de manière arithmétique : on rajoute r cases à chaque réallocation ($c_j = rj$, r fixé),
2. on agrandit le tableau de manière géométrique : on multiplie la capacité par $q > 1$ à chaque réallocation ($c_{j+1} \geq qc_j$).

Question : quelle nouvelle capacité ?

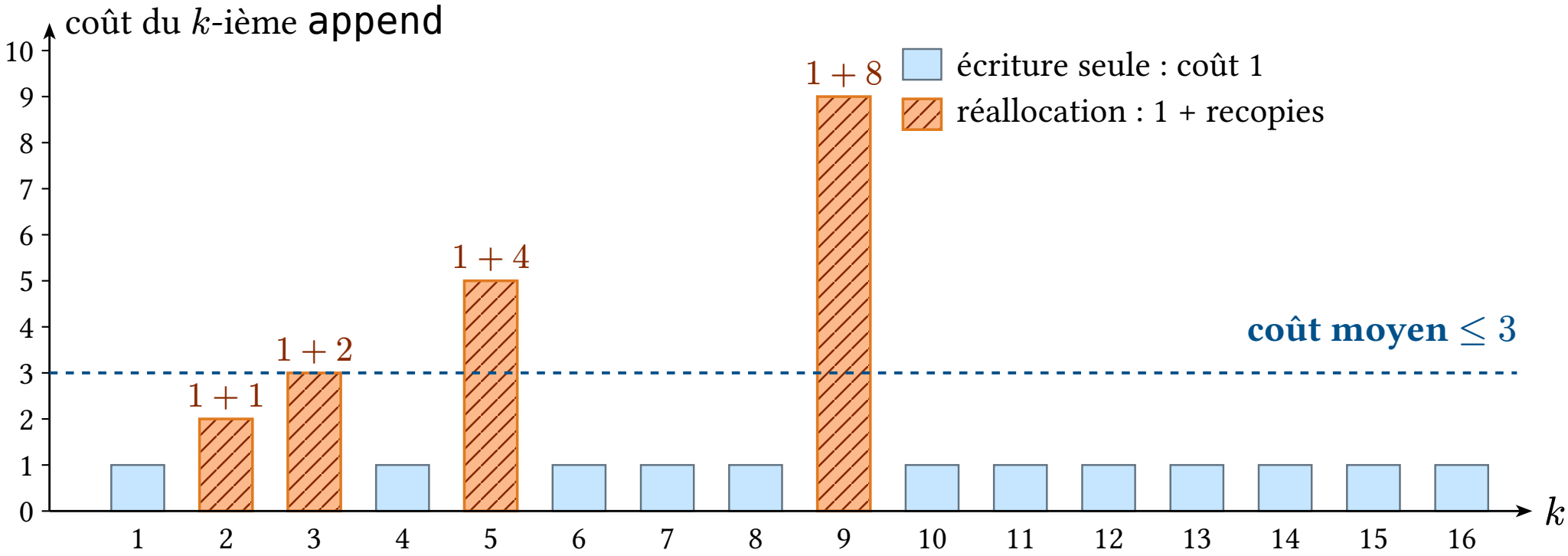
$$C(N) = N + \sum_{j\,:\,c_j < N} c_j$$

Question (poly § 2.3) Calculer $C(N)$ dans les deux cas suivants :

1. on agrandit le tableau de manière arithmétique : on rajoute r cases à chaque réallocation ($c_j = rj$, r fixé),
2. on agrandit le tableau de manière géométrique : on multiplie la capacité par $q > 1$ à chaque réallocation ($c_{j+1} \geq qc_j$).

- arithmétique : $C(N) = O(N^2)$, quadratique
- géométrique : $C(N) = O(N)$

Coût amorti



Coût amorti. toute suite de N opérations coûte $O(N)$.

Ce que fait Python

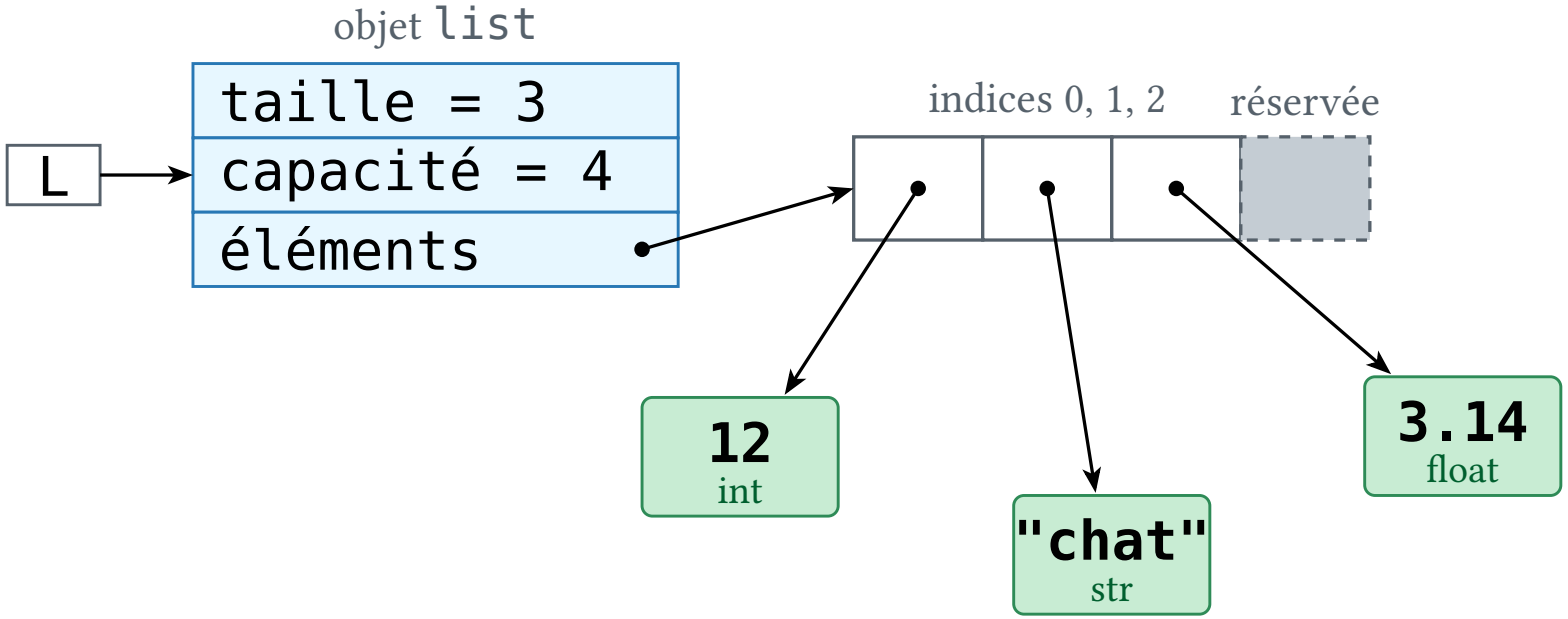
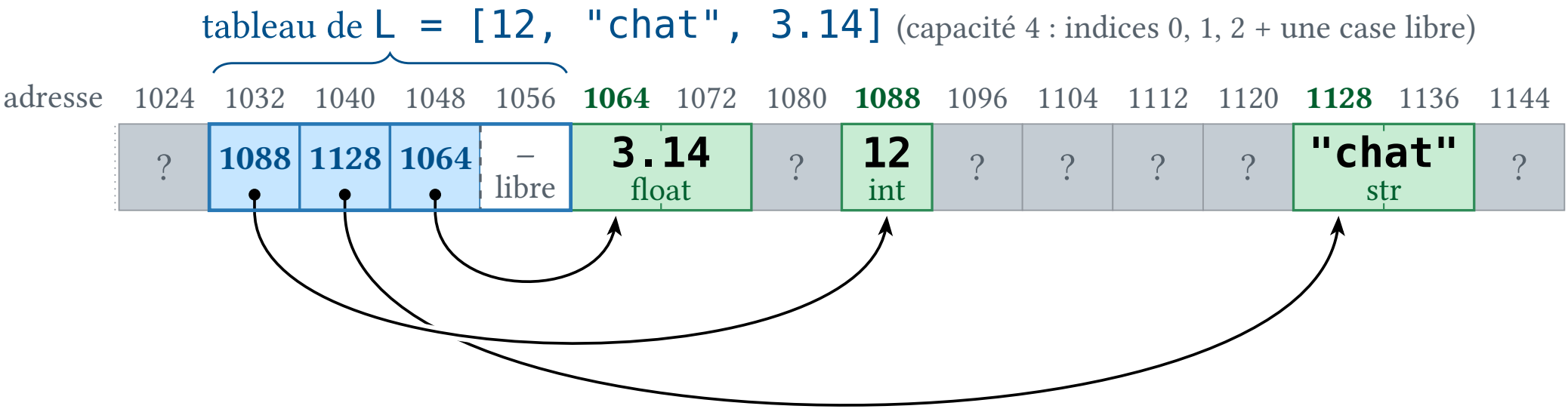


tableau de références, **contigu**, objets **n'importe où** en mémoire

- `list` = tableau dynamique de **références** (8 octets)
- capacités $0, 4, 8, 16, 24, 32, 40, 52, 64, \ldots$: $q \approx 1{,}125$

Ce que fait Python



- `list` = tableau dynamique de **références** (8 octets)
- capacités $0, 4, 8, 16, 24, 32, 40, 52, 64, \ldots$: $q \approx 1{,}125$

Tableau récapitulatif (Python)

Opération (liste Python)	Coût	Commentaire
`L[i]`, `L[i] = x`	$O(1)$	calcul d'adresse
`len(L)`	$O(1)$	la taille est stockée
`L.append(x)`, `L.pop()`	$O(1)$	amorti
`x in L`	$O(n)$	parcours séquentiel
`min(L)`, `max(L)`, `sum(L)`	$O(n)$	parcours (HP)
`L[i:j]`	$O(j-i)$	**copie** de la tranche
`L + M`	$O(n+m)$	nouvelle liste
`L.copy()`, `L[:]`	$O(n)$	copie superficielle
`L.sort()`, `sorted(L)`	$O(n \log n)$	HP : tri à savoir implémenter

Attention : `x in L`

- `x in L` sur une **liste** : parcours, $O(n)$, dans une boucle $\Rightarrow$ quadratique
- par précaution : **ne jamais l'utiliser** (souvent interdit) et écrire la boucle
- `k in d` sur un **dictionnaire** : $O(1)$ en moyenne
- `L = L + [x]` : même contenu que `L.append(x)`, mais $O(n)$ au lieu de $O(1)$ amorti

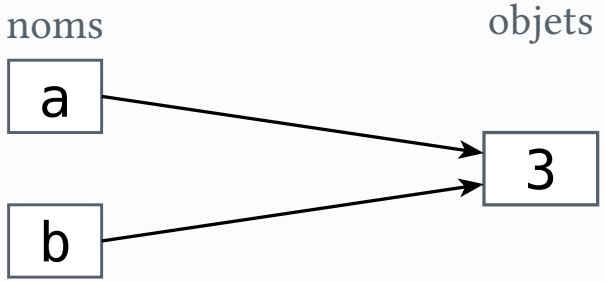
Tableau récapitulatif

	accès i-ème	ajout fin	suppr. fin	concaténation	recherche
tableau	$O(1)$	impossible*	impossible*	impossible*	$O(n)$
liste chaînée	$O(i)$	$O(n)$	$O(n)$	$O(n_1)$	$O(n)$
tableau dynamique	$O(1)$	$O(1)$ am.	$O(1)$ am.	$O(n_1+n_2)$	$O(n)$
▸ **liste Python**	`L[i]`	`L.append(x)`	`L.pop()`	`L1 + L2`	`x in L`

(*) nouveau tableau, $O(n)$. Aux concours : `append`, `pop()` annoncés en $O(1)$.

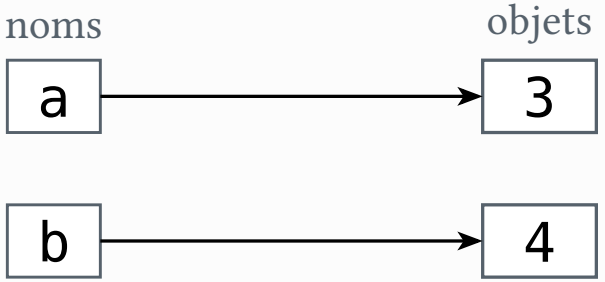
Rappel : références et copies

`a = 3` puis `b = a`



un seul objet, deux noms

puis `b = b + 1`

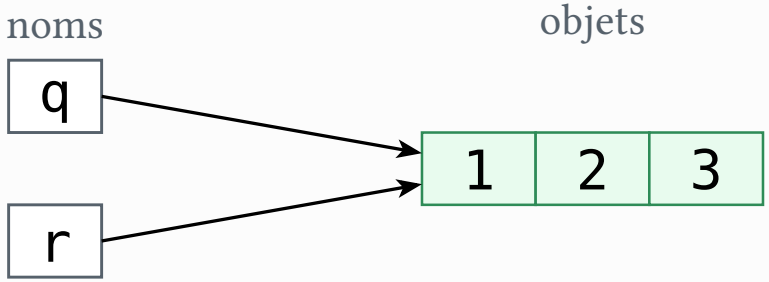


nouvel objet pour `b`, `a` vaut toujours 3

- variable = **étiquette** : elle contient une **référence**, `b = a` ne copie jamais l'objet
- immuable : `b = b + 1` crée un nouvel objet, liste : `r = q` partage, `r = q.copy()` copie

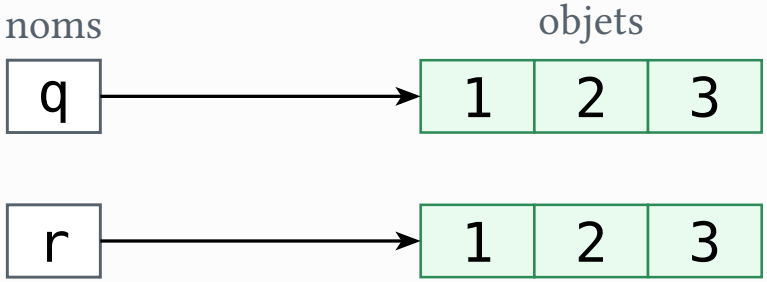
Rappel : références et copies

`r = q`



▸ **`r.append(4)` modifie aussi `q`**

`r = q.copy()`

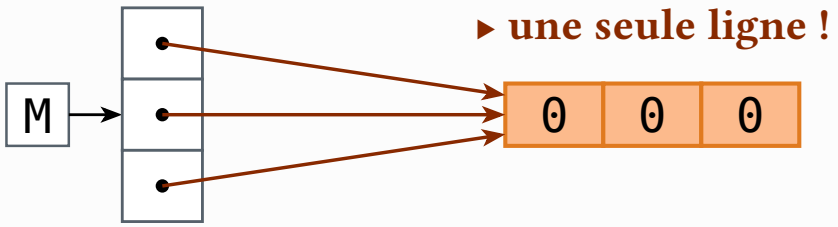


`r.append(4)` ne modifie pas `q`

- variable = **étiquette** : elle contient une **référence**, `b = a` ne copie jamais l'objet
- immuable : `b = b + 1` crée un nouvel objet, liste : `r = q` partage, `r = q.copy()` copie

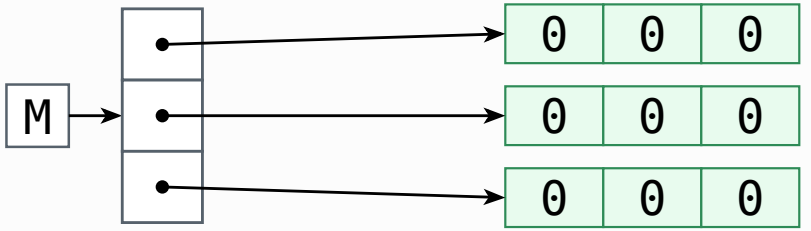
Attention : listes de listes

```
M = [[0] * 3] * 3
```



`M[0][0] = 1` donne
`[[1,0,0],[1,0,0],[1,0,0]]`

```
M = [[0 for _ in range(3)]
     for _ in range(3)]
```



`M[0][0] = 1` donne
`[[1,0,0],[0,0,0],[0,0,0]]`

- `[x] * n`, `[[]] * m` : n fois **la même** référence
- toujours : `[[0 for _ in range(m)] for _ in range(n)]`, `[[] for _ in range(m)]`

Question : en place ou nouvelle liste ?

Question (poly § 2.5) Pour chacune des lignes suivantes, dire si `L` est modifiée **en place** ou si une **nouvelle liste** est créée, et donner le coût si `L` a n éléments.

```
L.append(x)    L = L + [x]    L += [x]    L = L.append(x)
```

Question : en place ou nouvelle liste ?

Question (poly § 2.5) Pour chacune des lignes suivantes, dire si `L` est modifiée **en place** ou si une **nouvelle liste** est créée, et donner le coût si `L` a n éléments.

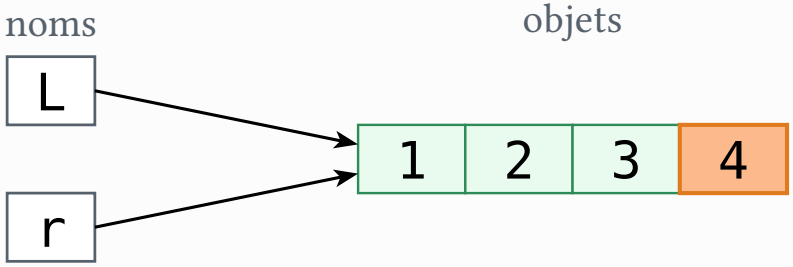
```
L.append(x)    L = L + [x]    L += [x]    L = L.append(x)
```

- `L.append(x)` : en place, $O(1)$ amorti,

-`L = L + [x]` : nouvelle liste, $O(n)$
- `L += [x]` : en place, $O(1)$ amorti,
- `L = L.append(x)` : `L` vaut `None` !

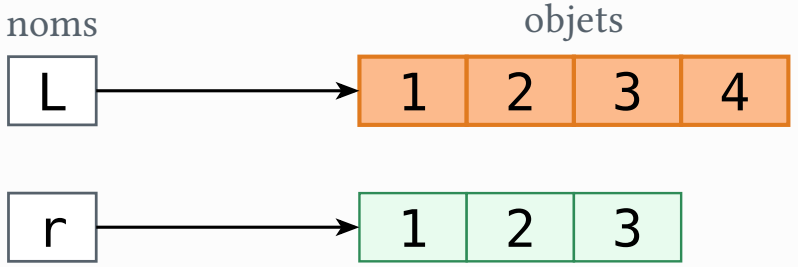
`L.append(x)` contre `L = L + [x]`

`L.append(4)`



en place, $O(1)$ amorti : `r` voit aussi le 4

`L = L + [4]`

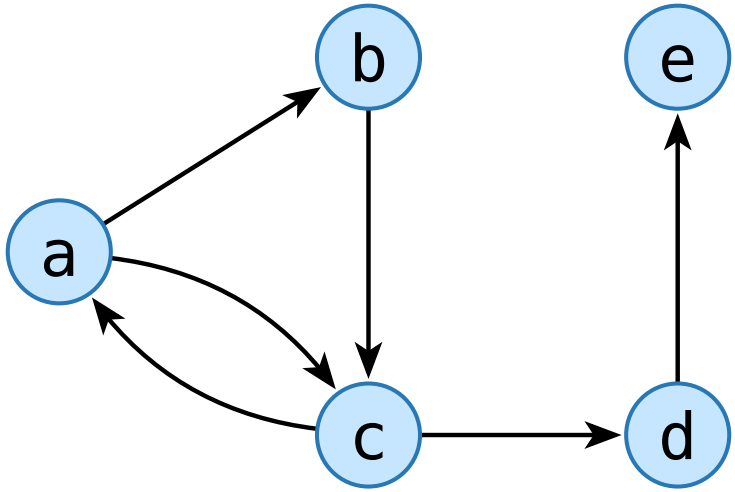


▸ **nouvelle liste, $O(n)$: `r` est inchangée**

- `L.append(x)` : **en place**, $O(1)$ amorti : tous les noms de la liste voient l'ajout
- `L = L + [x]` : **nouvelle liste**, $O(n)$: l'étiquette `L` est déplacée

Dictionnaires et hachage

Le problème



`voisins`

clé	valeur
`"a"`	`["b", "c"]`
`"b"`	`["c"]`
`"c"`	`["a", "d"]`
`"d"`	`["e"]`
`"e"`	`[]`

- sommets désignés par des **noms** : pas de tableau indexé par $0, \ldots, n-1$
- de même pour compter les animaux de chaque type d'un zoo

Dictionnaire : définition

Dictionnaire. structure qui stocke des couples (**clé**, **valeur**), chaque clé apparaissant **au plus une fois**. Opérations :

- **ajouter** un couple ou modifier la valeur associée à une clé,
- **rechercher** si une clé est présente et, le cas échéant, la valeur associée,
- **supprimer** une clé.

$$d : \mathcal{D} \to \mathcal{V}$$
$$k \mapsto d(k)$$

$\mathcal{D} \subset \mathcal{C}$ fini : clés présentes

$$\text{voisins} : S \to \mathcal{P}(S)$$
$$s \mapsto \{t \in S \mid (s,t) \in A\}$$

$$\text{compte} : T \to \mathbb{N}$$
$$t \mapsto |\{a \in Z \mid \text{type}(a) = t\}|$$

Question : une implémentation naïve

Question (poly § 3.1.1) On représente un dictionnaire par une liste Python de couples `D = [(c1, v1), (c2, v2), ...]`. Écrire :

- `rechercher_naif(D, cle)`, qui renvoie la valeur associée à `cle` si la clé est présente, `None` sinon,
- `ajouter_naif(D, cle, valeur)`, qui modifie `D` en place : si `cle` est présente, sa valeur est remplacée par `valeur`, sinon le couple `(cle, valeur)` est ajouté.

Quelle est la complexité de ces fonctions en fonction de la longueur n de `D` ?

Question : une implémentation naïve (correction)

```
def rechercher_naif(D, cle):
    for (c, v) in D:
        if c == cle:
            return v
    return None

def ajouter_naif(D, cle, valeur):
    for i in range(len(D)):
        if D[i][0] == cle:        # clé déjà présente : on écrase
            D[i] = (cle, valeur)
            return
    D.append((cle, valeur))
```

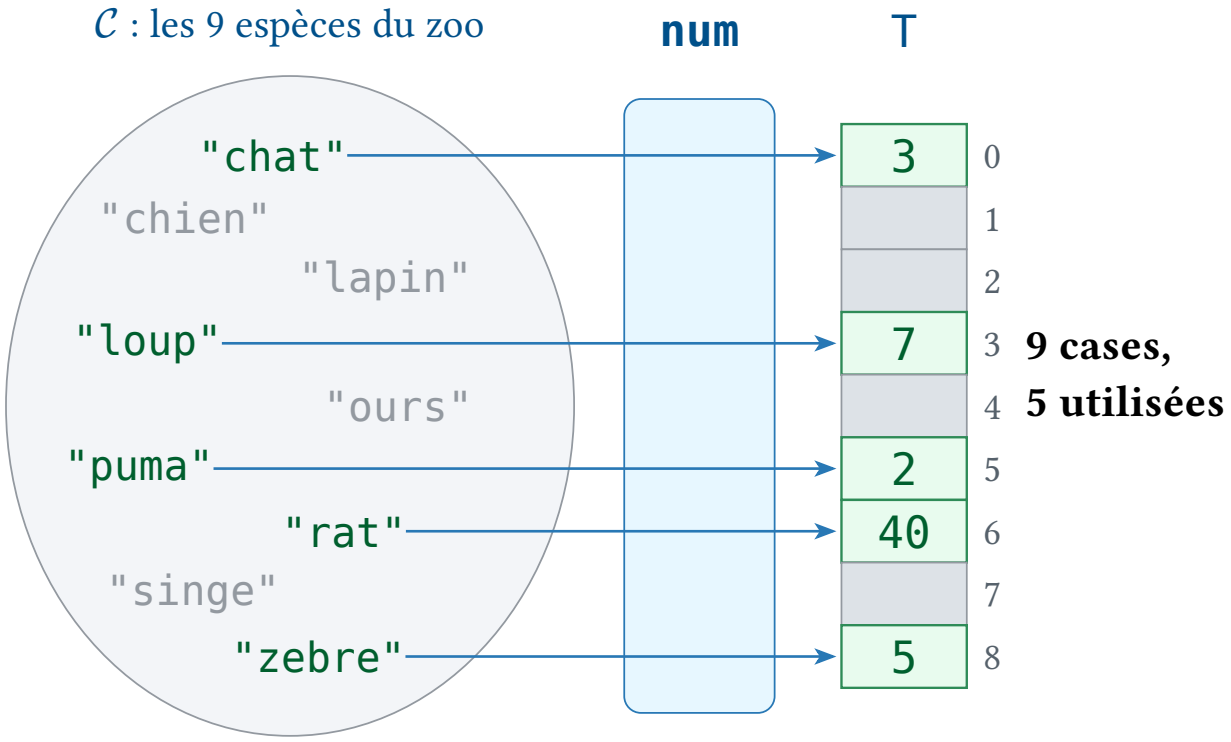
Question : une implémentation naïve (correction)

```
def rechercher_naif(D, cle):
    for (c, v) in D:
        if c == cle:
            return v
    return None

def ajouter_naif(D, cle, valeur):
    for i in range(len(D)):
        if D[i][0] == cle:        # clé déjà présente : on écrase
            D[i] = (cle, valeur)
            return
    D.append((cle, valeur))
```

$O(n)$: il faut parcourir toute la liste, ne serait-ce que pour vérifier que la clé est absente. Objectif : $O(1)$!

Adressage direct



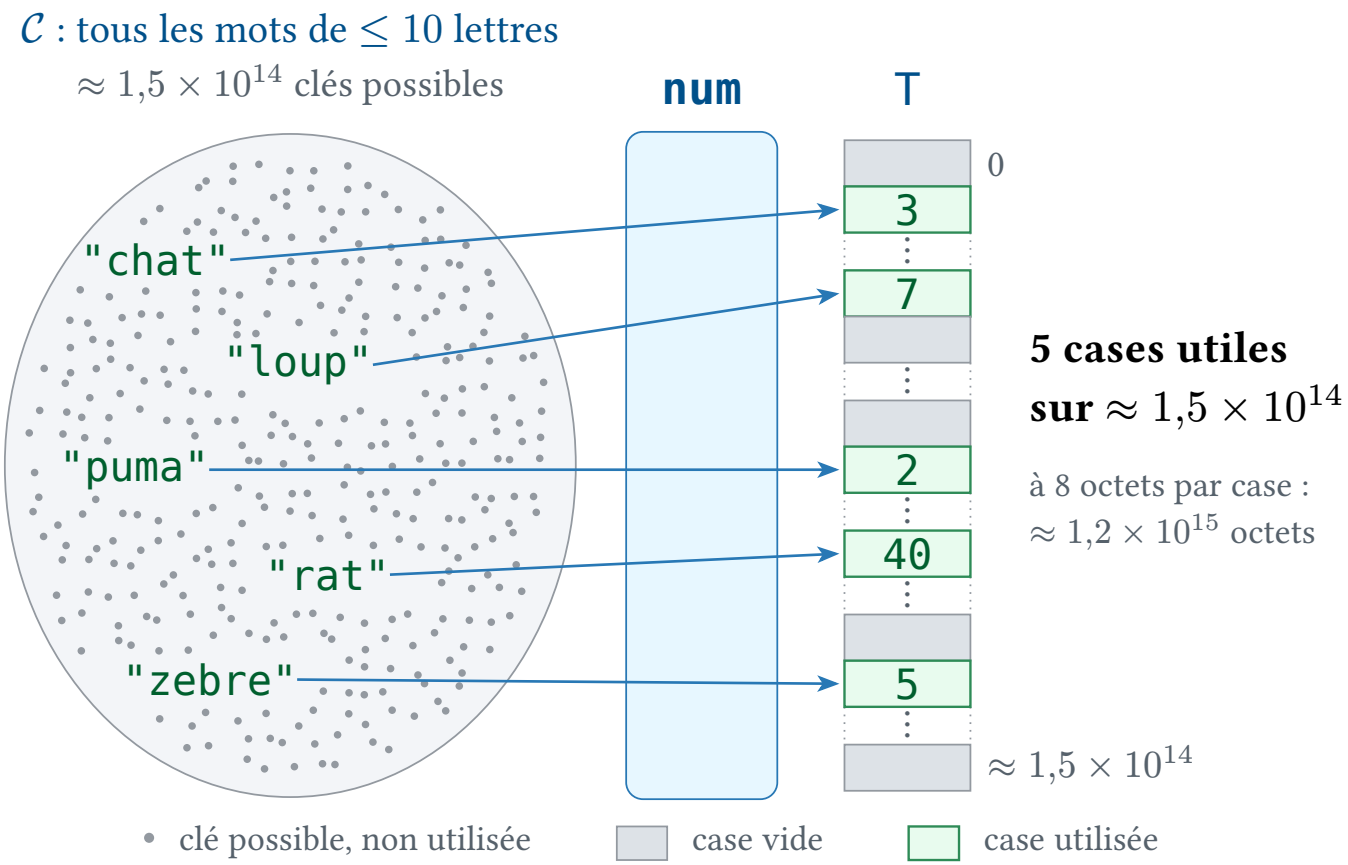
`num("chat") = 0`, `num("chien") = 1`, …, `num("zebre") = 8`

`"ours"` clé possible, absente — case vide — case utilisée

`T[num(k)]` : effectif de l'espèce `k` s'il est dans le dictionnaire, case vide sinon

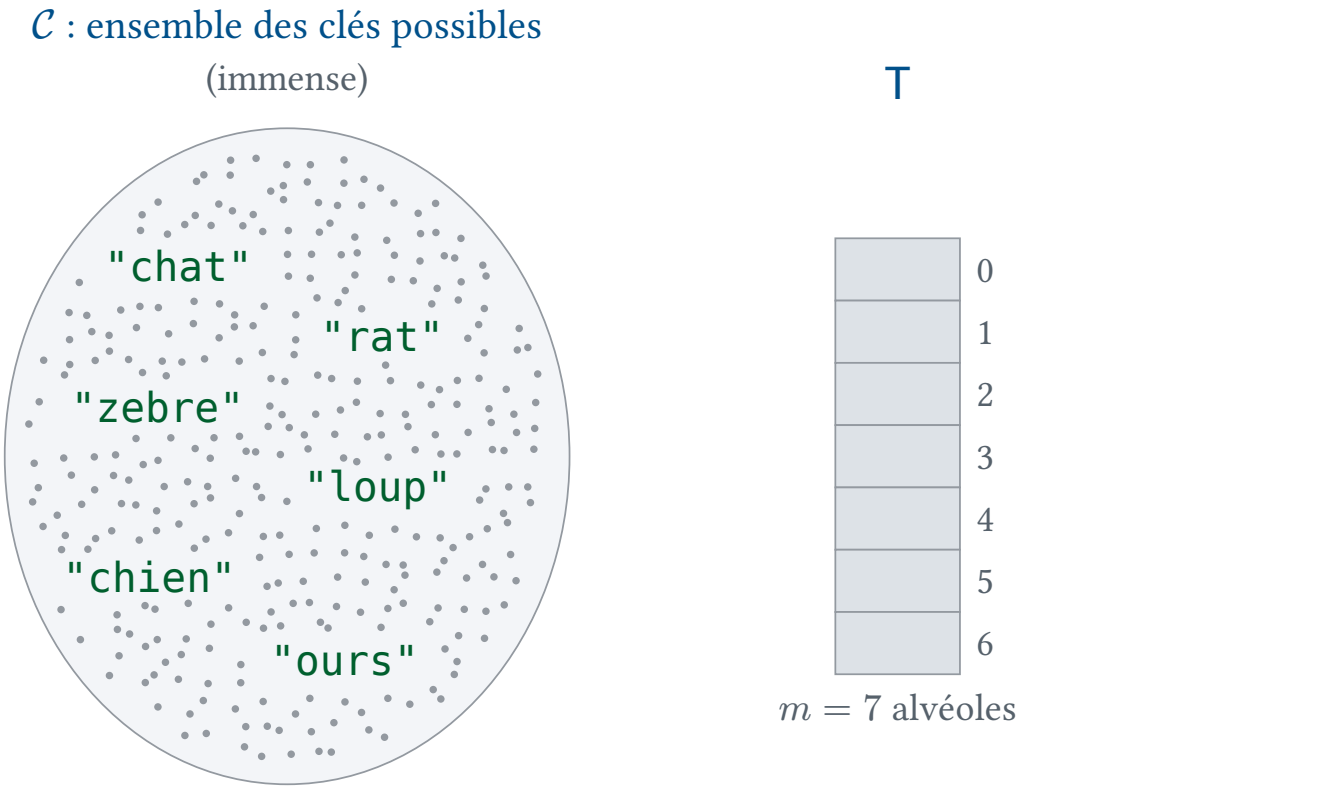
- num : $\mathcal{C} \to [\![0, m-1]\!]$ **injective**
- tout en $O(1)$
- mais mémoire $O(m)$, $m \geq |\mathcal{C}|$: clés **possibles**
- utile si $\mathcal{C}$ est petit : lettres, tri par comptage, matrice d'adjacence, crible

Adressage direct



- num : $\mathcal{C} \to [\![0, m-1]\!]$ **injective**
- tout en $O(1)$
- mais mémoire $O(m)$, $m \geq |\mathcal{C}|$: clés **possibles**
- utile si $\mathcal{C}$ est petit : lettres, tri par comptage, matrice d'adjacence, crible

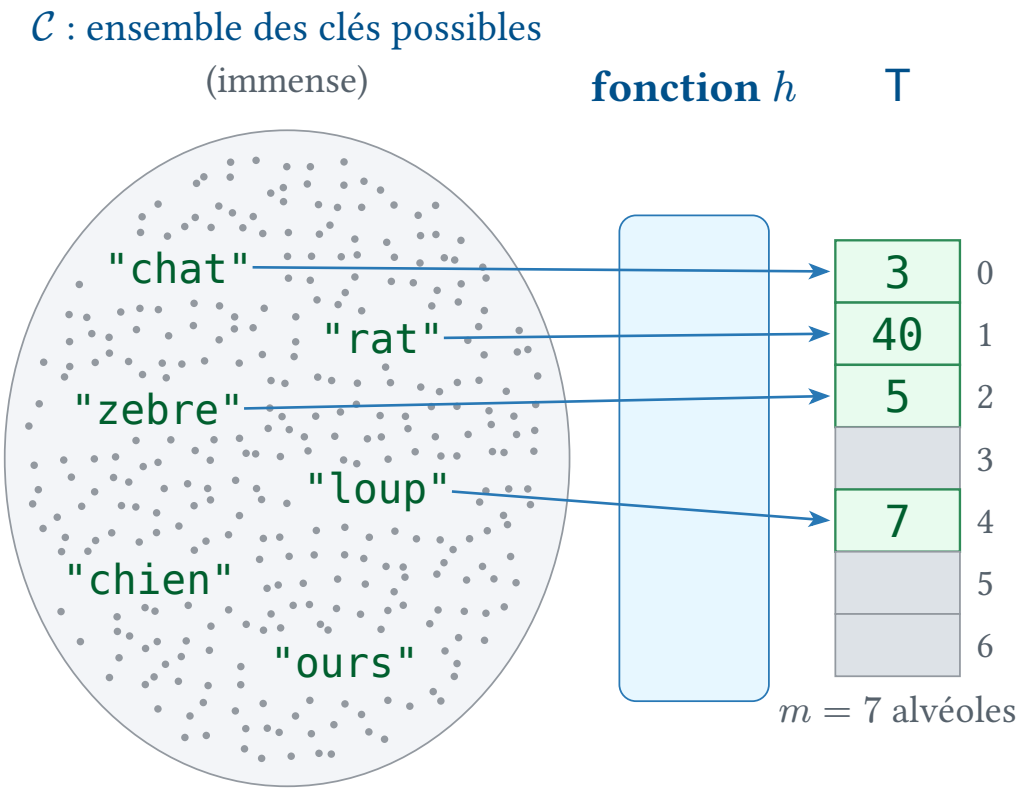
Fonctions de hachage et collisions



- clé possible, non utilisée
- alvéole vide
- alvéole utilisée
- collision

- $h : \mathcal{C} \to [\![0, m-1]\!]$
- $h(k) =$ `hache`$(k)\%m$
- **collision** : $k \neq k', h(k) = h(k')$

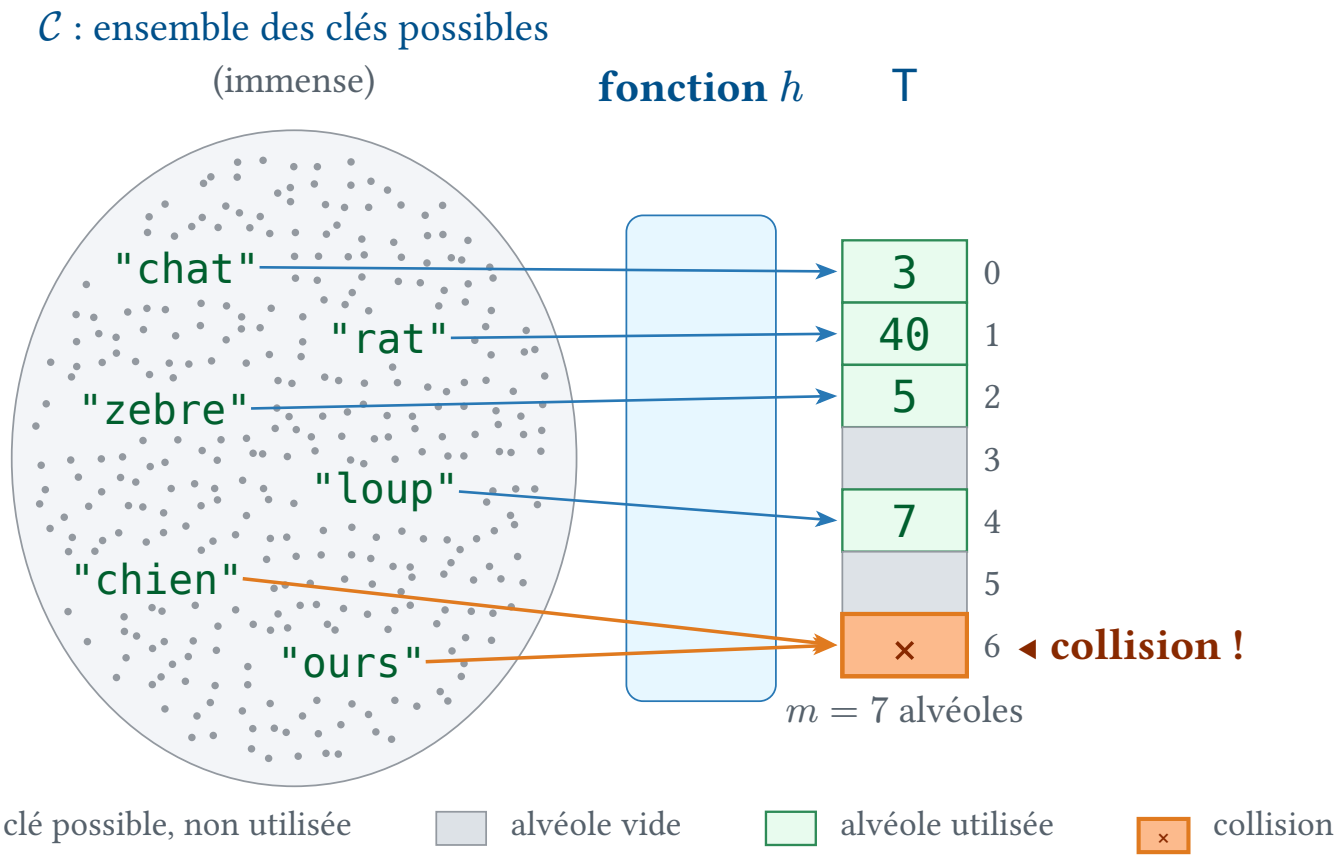
Fonctions de hachage et collisions



- clé possible, non utilisée
- alvéole vide
- alvéole utilisée
- collision

- $h : \mathcal{C} \to [\![0, m-1]\!]$
- $h(k) =$ `hache`$(k) \% m$
- **collision** : $k \neq k', h(k) = h(k')$

Fonctions de hachage et collisions



- clé possible, non utilisée
- alvéole vide
- alvéole utilisée
- collision

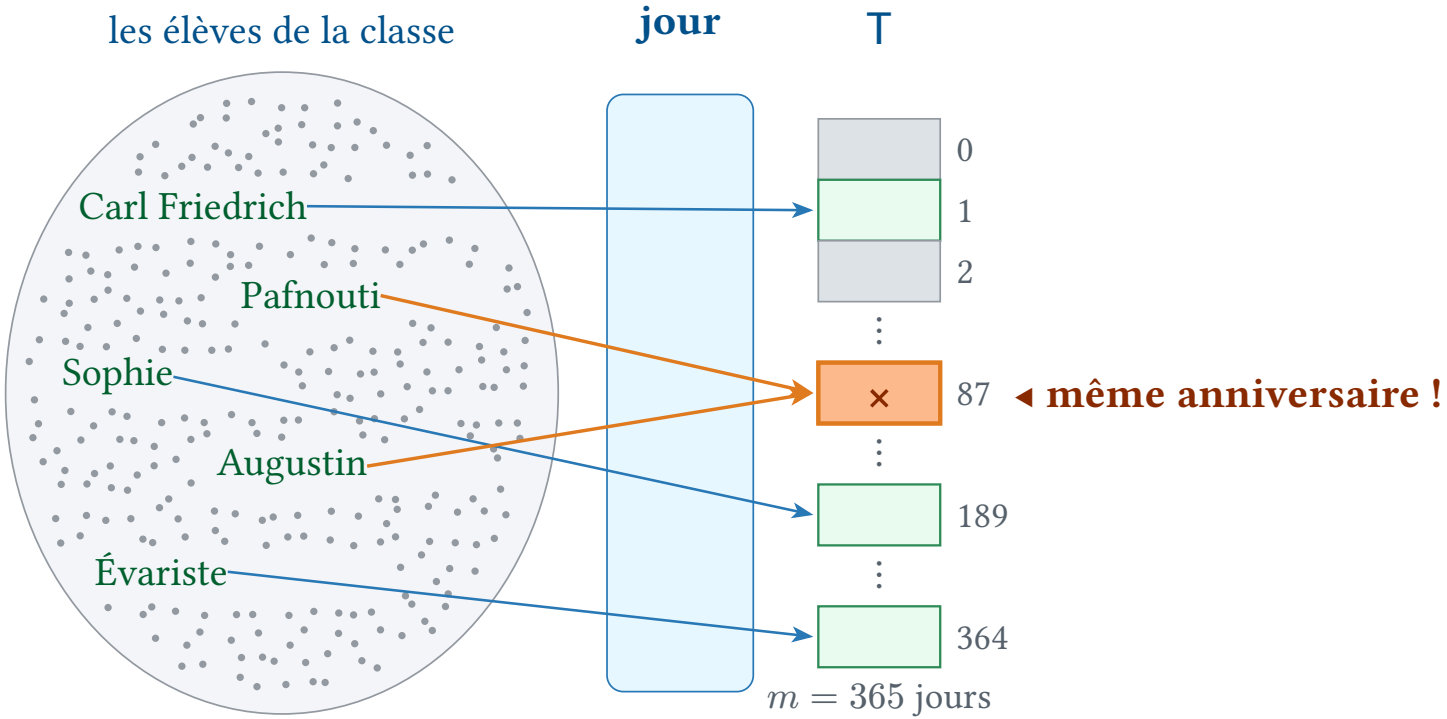
- $h : \mathcal{C} \to [\![0, m-1]\!]$
- $h(k) =$ `hache`$(k)\%m$
- **collision** : $k \neq k', h(k) = h(k')$

Facteur de charge

Facteur de charge. $\alpha = n/m$, où $n = |\mathcal{D}|$ est le nombre de clés présentes et m le nombre d'alvéoles.

- α grand : beaucoup de collisions, si $\alpha > 1$, au moins une (principe des **tiroirs**)
- α petit : mémoire gaspillée (n/α alvéoles pour n clés)
- on le garde **borné** (et < 1 en adressage ouvert)

Le paradoxe des anniversaires



- 23 élèves : une chance sur deux d'avoir deux anniversaires identiques, 42 élèves : 0,91

Question : le paradoxe des anniversaires

n clés envoyées **uniformément** et **indépendamment** dans m alvéoles.

Question (poly § 3.3.1) On note p_n la probabilité qu'il n'y ait aucune collision.

- Calculer p_n en fonction de n et m.
- Déterminer un équivalent de $\ln(p_n)$ lorsque $n \to +\infty$ avec $m \sim \alpha n$, où $\alpha \in]0, 1[$ est constant.
- Déterminer un équivalent de $\ln(p_n)$ lorsque $n \to +\infty$ avec $m \sim Cn^\beta$, où $C > 0$ et $\beta > 1$ sont constants.
- Pour quelles valeurs de β a-t-on $p_n \to 0$? Commenter.

Question : le paradoxe des anniversaires

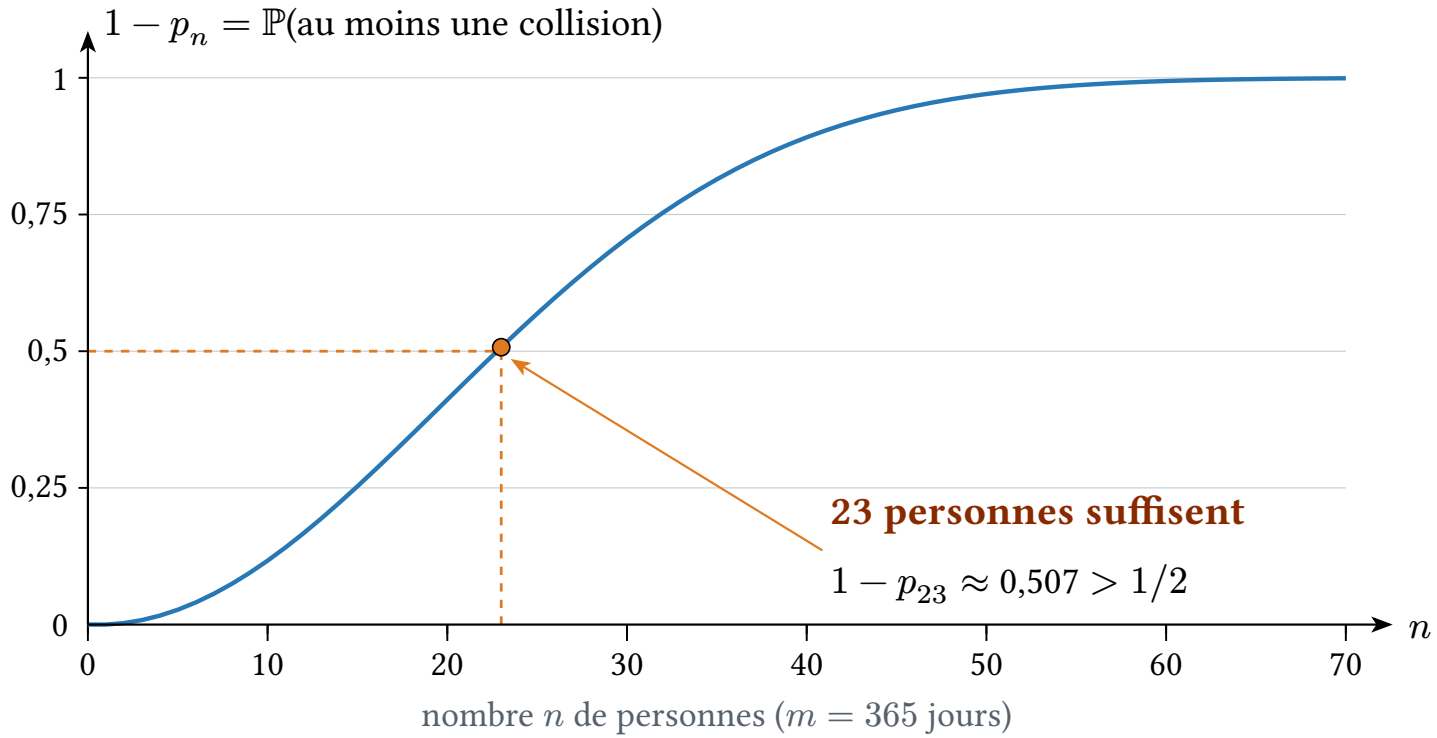
n clés envoyées **uniformément** et **indépendamment** dans m alvéoles.

Question (poly § 3.3.1) On note p_n la probabilité qu'il n'y ait aucune collision.

- Calculer p_n en fonction de n et m.
- Déterminer un équivalent de $\ln(p_n)$ lorsque $n \to +\infty$ avec $m \sim \alpha n$, où $\alpha \in]0, 1[$ est constant.
- Déterminer un équivalent de $\ln(p_n)$ lorsque $n \to +\infty$ avec $m \sim Cn^\beta$, où $C > 0$ et $\beta > 1$ sont constants.
- Pour quelles valeurs de β a-t-on $p_n \to 0$? Commenter.

- $p_n = \prod_{i=0}^{n-1}(1 - i/m)$,
- si $m \sim \alpha n$: $\ln p_n \sim -C(\alpha)n$
- si $m \sim Cn^\beta$: $\ln p_n \sim -n^{2-\beta}/(2C)$,
- $p_n \to 0$ ssi $\beta < 2$: il faudrait m de l'ordre de n^2

Le paradoxe des anniversaires : la courbe



- $\mathbb{P}(\text{collision}) > 1/2$ dès $n \approx 1{,}18\sqrt{m}$: 23 pour $m = 365$, 1178 pour $m = 10^6$
- éviter les collisions demanderait m de l'ordre de n^2 : il faut les **gérer**

Coûts moyens

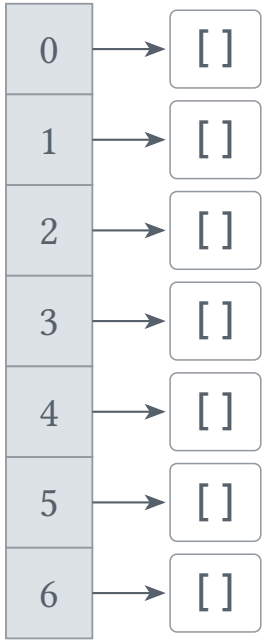
- recherche **fructueuse** ($k \in \mathcal{D}$, choisie uniformément) : coût X_{F}
- recherche **infructueuse** ($k \notin \mathcal{D}$) : coût X_{I}
- ajouter, supprimer = une recherche + $O(1)$, on étudie $\mathbb{E}[X_{\mathrm{I}}]$ et $\mathbb{E}[X_{\mathrm{F}}]$ en fonction de α

Hachage uniforme. $h(k_1), \ldots, h(k_n)$ indépendantes, uniformes sur $[\![0, m-1]\!]$, pour k absente, $h(k)$ aussi, indépendante des précédentes.

Chaînage : principe

- alvéole $T[i]$ = **liste** des couples (k, v) tels que $h(k) = i$ (petit dictionnaire naïf)
- **rechercher** k : parcourir la liste $T[h(k)]$
- **ajouter** (k, v) : rechercher k, présente : on remplace la valeur, sinon : ajout **en fin** de $T[h(k)]$
- **supprimer** k : rechercher k, retirer le couple de la liste

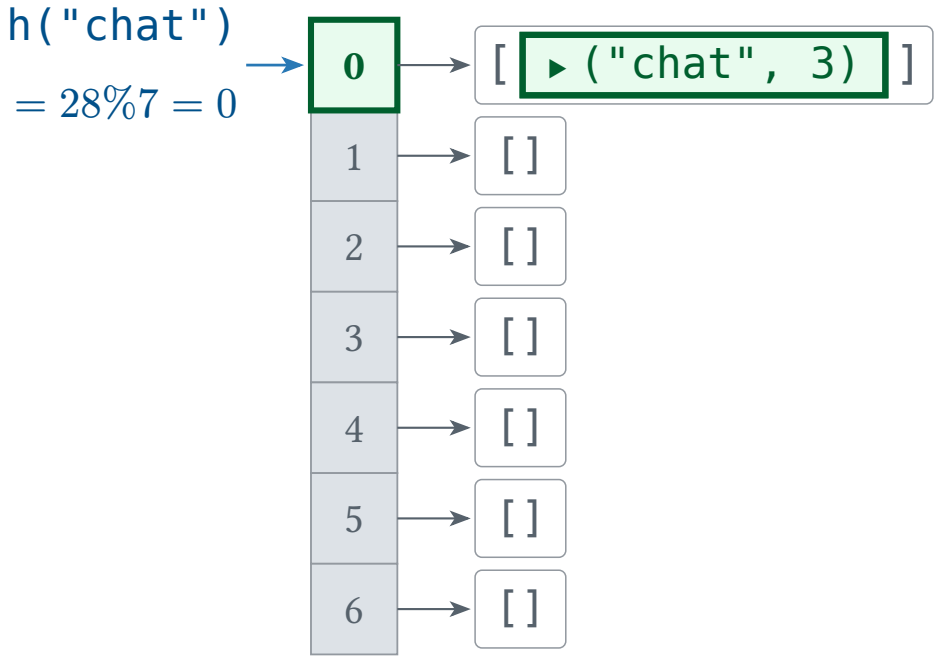
Chaînage : exemple



alvéole = liste de couples (clé, valeur)

- $m = 7$, même ordre que pour l'adressage ouvert : collisions pour `puma`, `singe`, `ours`, ajoutés **en fin** de liste

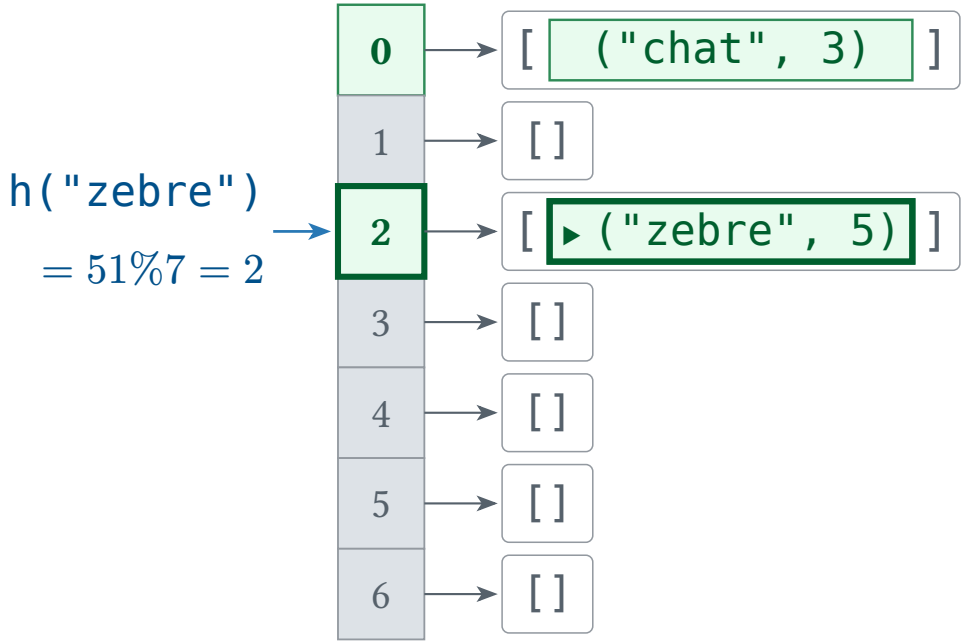
Chaînage : exemple



alvéole = liste de couples (clé, valeur)

- $m = 7$, même ordre que pour l'adressage ouvert : collisions pour `puma`, `singe`, `ours`, ajoutés **en fin** de liste

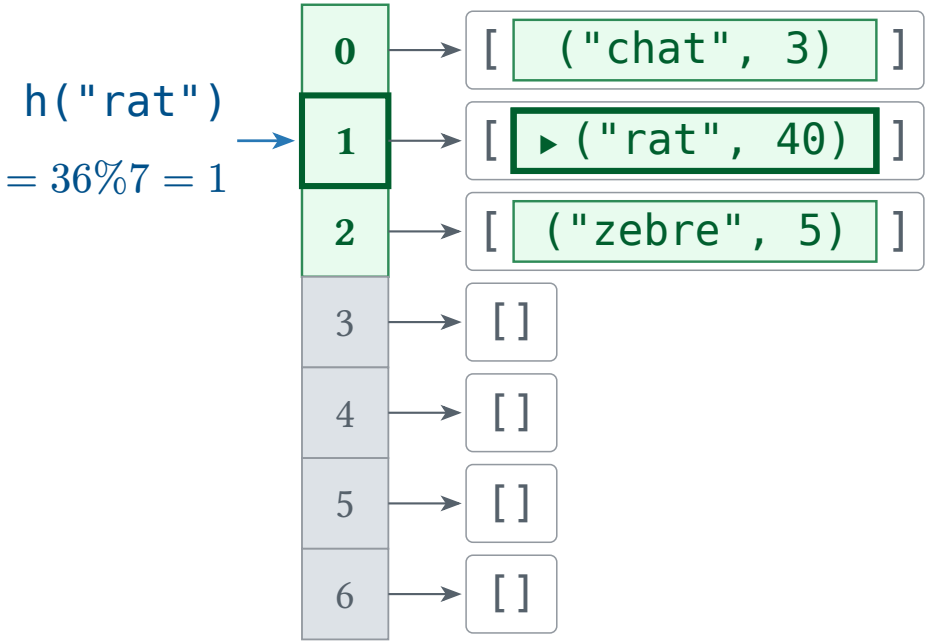
Chaînage : exemple



alvéole = liste de couples (clé, valeur)

- $m = 7$, même ordre que pour l'adressage ouvert : collisions pour `puma`, `singe`, `ours`, ajoutés **en fin** de liste

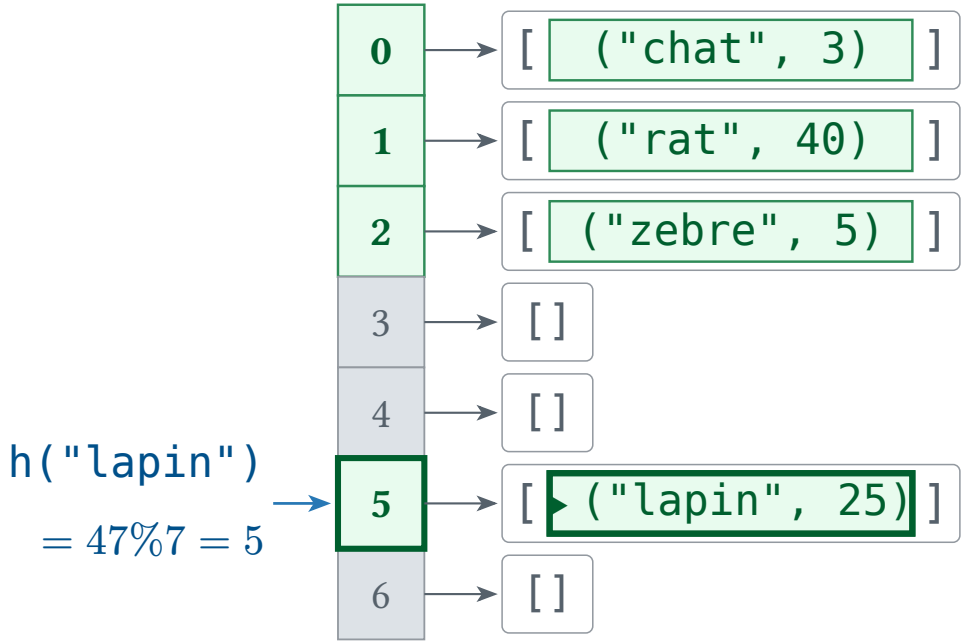
Chaînage : exemple



alvéole = liste de couples (clé, valeur)

- $m = 7$, même ordre que pour l'adressage ouvert : collisions pour `puma`, `singe`, `ours`, ajoutés **en fin** de liste

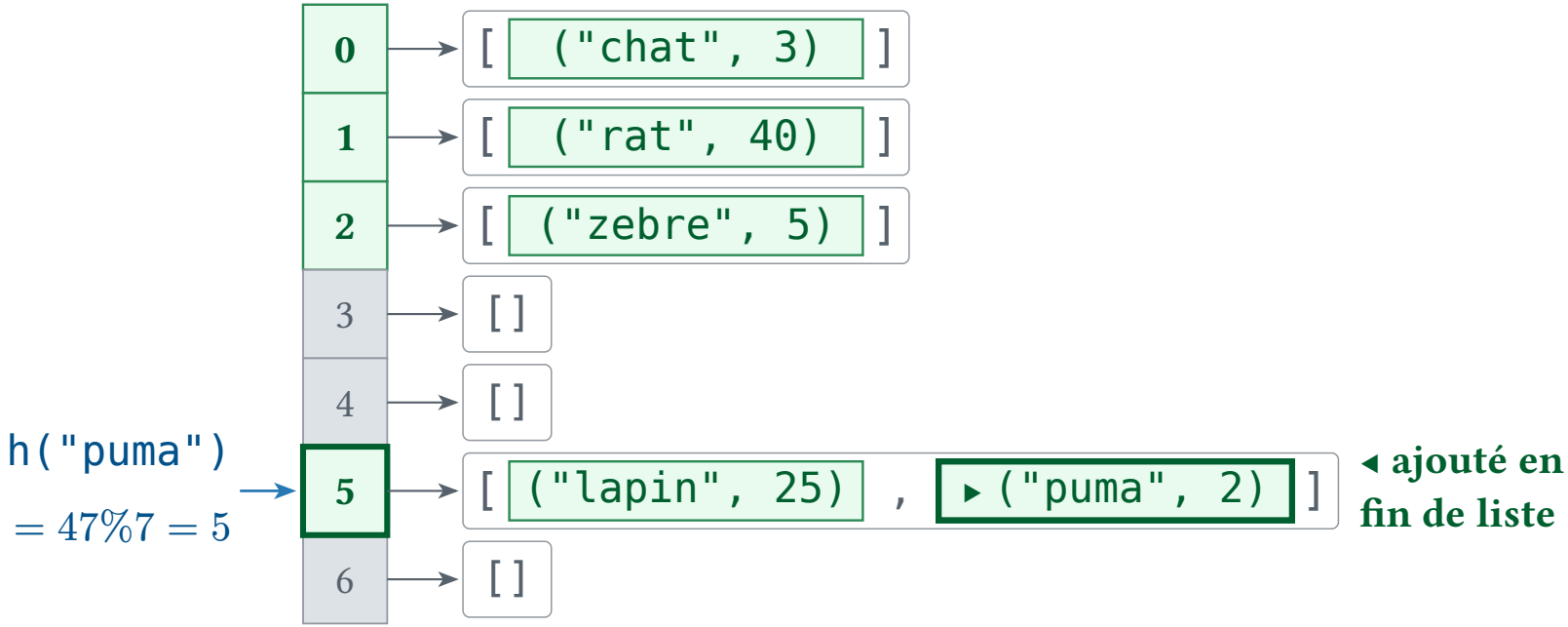
Chaînage : exemple



alvéole = liste de couples (clé, valeur)

- $m = 7$, même ordre que pour l'adressage ouvert : collisions pour `puma`, `singe`, `ours`, ajoutés **en fin** de liste

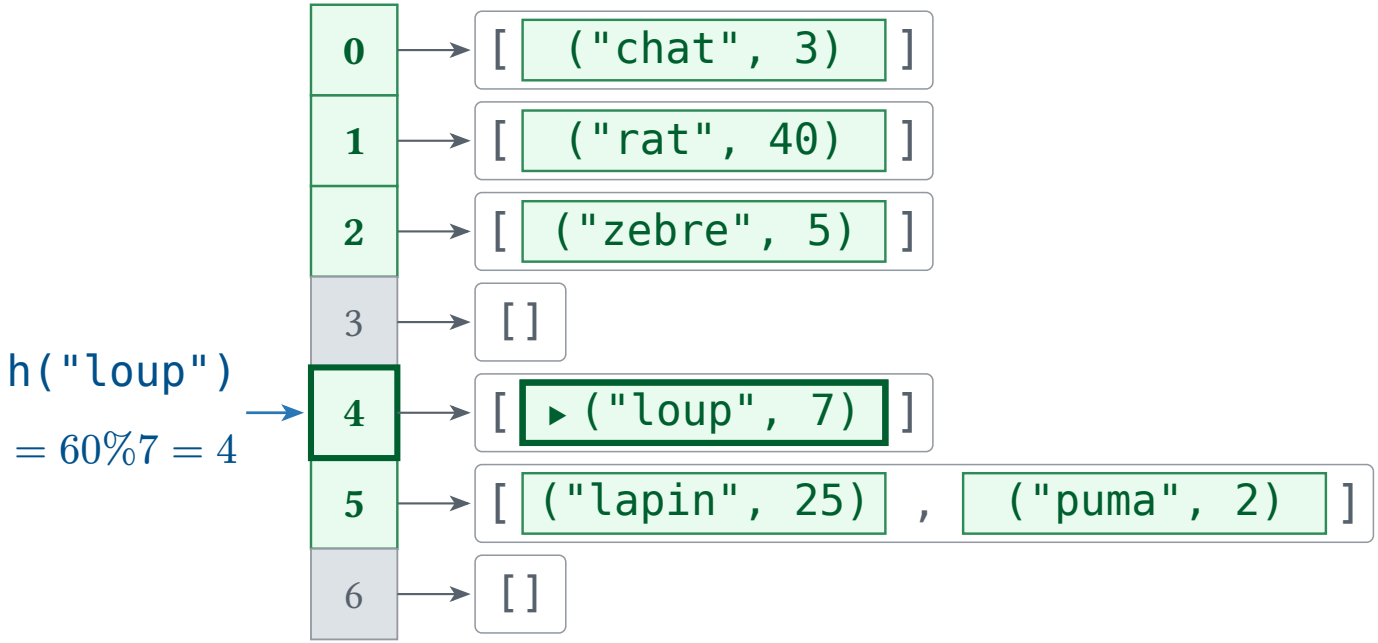
Chaînage : exemple



alvéole = liste de couples (clé, valeur)

- $m = 7$, même ordre que pour l'adressage ouvert : collisions pour `puma`, `singe`, `ours`, ajoutés **en fin** de liste

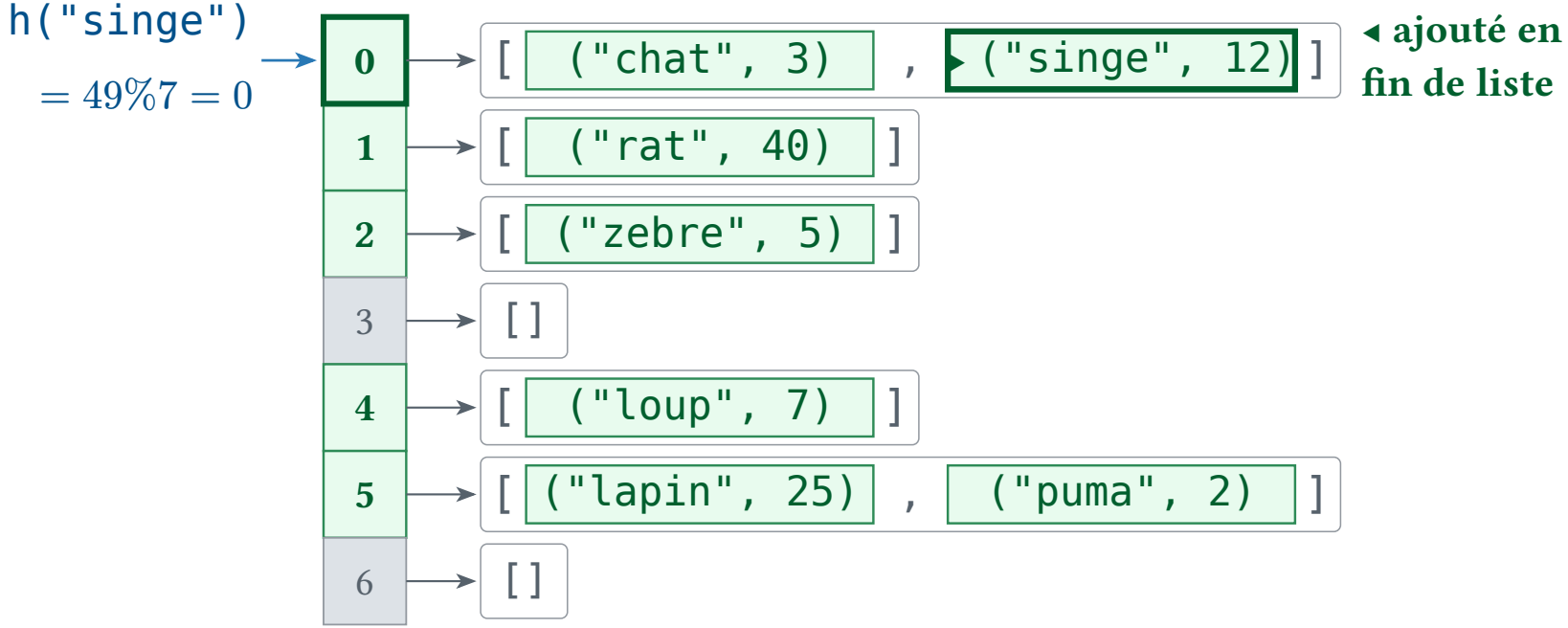
Chaînage : exemple



alvéole = liste de couples (clé, valeur)

- $m = 7$, même ordre que pour l'adressage ouvert : collisions pour `puma`, `singe`, `ours`, ajoutés **en fin** de liste

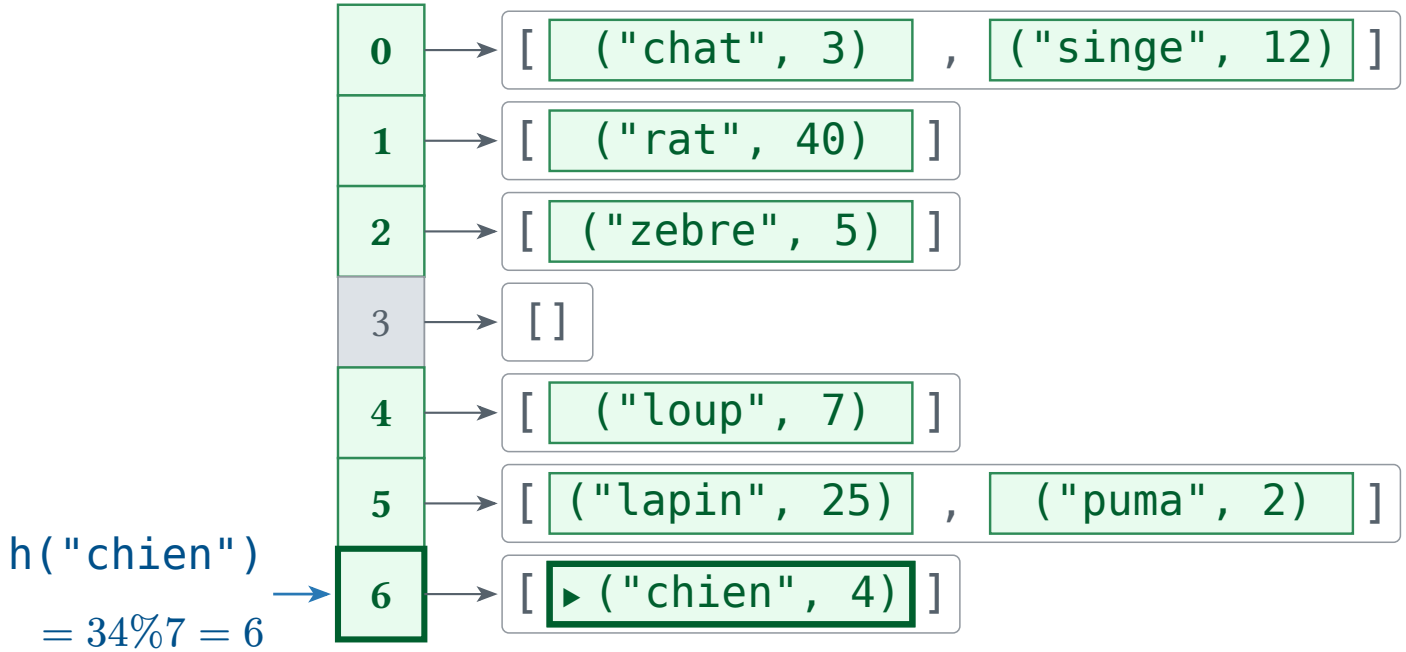
Chaînage : exemple



alvéole = liste de couples (clé, valeur)

- $m = 7$, même ordre que pour l'adressage ouvert : collisions pour `puma`, `singe`, `ours`, ajoutés **en fin** de liste

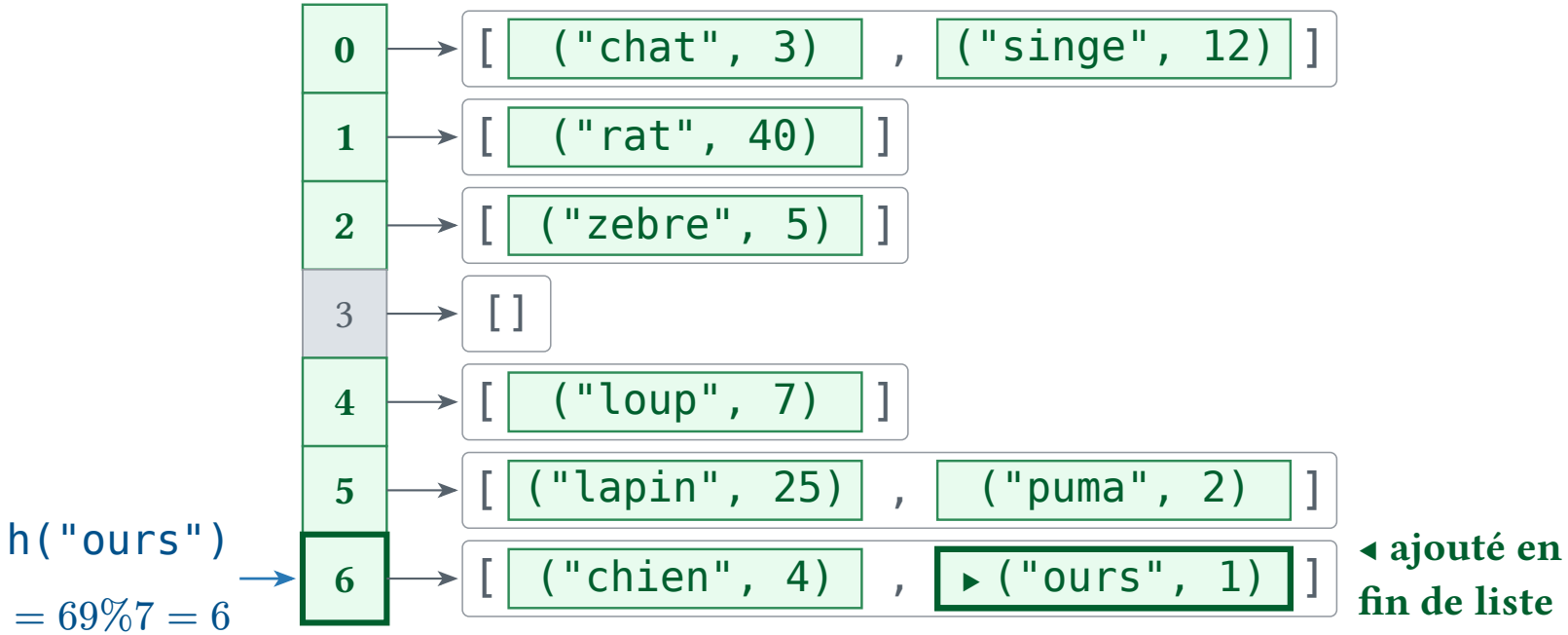
Chaînage : exemple



alvéole = liste de couples (clé, valeur)

- $m = 7$, même ordre que pour l'adressage ouvert : collisions pour `puma`, `singe`, `ours`, ajoutés **en fin** de liste

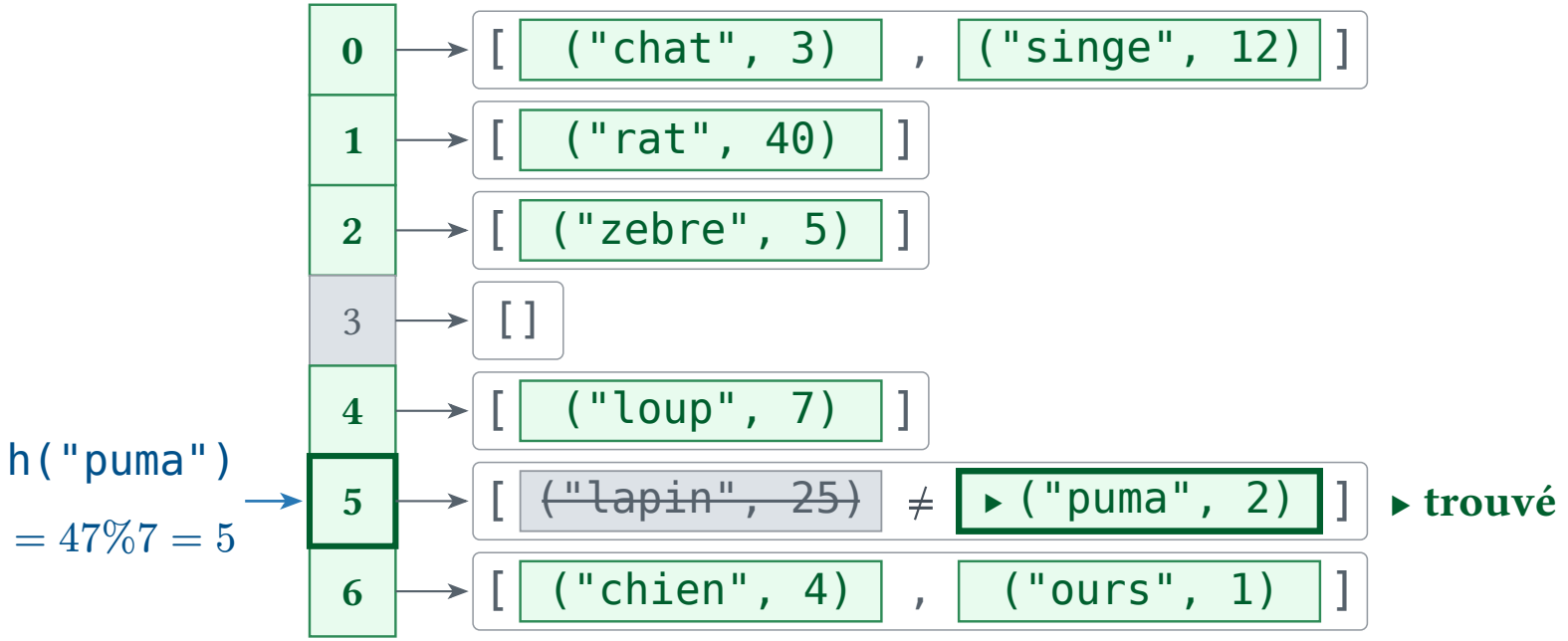
Chaînage : exemple



alvéole = liste de couples (clé, valeur)

- $m = 7$, même ordre que pour l'adressage ouvert : collisions pour `puma`, `singe`, `ours`, ajoutés **en fin** de liste

Chaînage : recherche



alvéole = liste de couples (clé, valeur)

- `"puma"` : alvéole 5, on compare à `"lapin"`, puis `"puma"` : trouvé
- `"girafe"` : $40 \% 7 = 5$, on compare à `"lapin"`, `"puma"`, fin de liste : absente

Question : chaînage en Python

Question (poly § 3.4.3) Avec `hache` et m alvéoles, écrire `creer_table(m)`, `rechercher(T, cle)`, `ajouter(T, cle, valeur)` et `supprimer(T, cle)` pour le chaînage, en réutilisant `rechercher_naif` et `ajouter_naif`. Complexités ?

Question : chaînage en Python

Question (poly § 3.4.3) Avec `hache` et m alvéoles, écrire `creer_table(m)`, `rechercher(T, cle)`, `ajouter(T, cle, valeur)` et `supprimer(T, cle)` pour le chaînage, en réutilisant `rechercher_naif` et `ajouter_naif`. Complexités ?

```
def creer_table(m):
    return [[] for _ in range(m)]

def rechercher(T, cle):
    alveole = T[hache(cle) % len(T)]
    return rechercher_naif(alveole, cle)

def ajouter(T, cle, valeur):
    alveole = T[hache(cle) % len(T)]
    ajouter_naif(alveole, cle, valeur)
```

```
def supprimer(T, cle):
    alveole = T[hache(cle) % len(T)]
    for i in range(len(alveole)):
        if alveole[i][0] == cle:
            # on écrase par le dernier
            alveole[i] = alveole[-1]
            alveole.pop()     # O(1)
            return
```

`creer_table` : $O(m)$ (et surtout pas `[[]] * m`), les trois autres : $O(1+\alpha)$ en moyenne

Chaînage : analyse

- coût = nombre de clés **examinées** dans l'alvéole $h(k)$
- pire cas (une seule alvéole) : $O(n)$
- chaque alvéole reste petite : 42 élèves $\rightarrow$ environ 1,06 élève par jour occupé

Question : chaînage, recherche infructueuse

Question (poly § 3.4.4) (Recherche infructueuse.) Soit $k \notin \{k_1, \ldots, k_n\}$. Calculer $\mathbb{E}[X_{\mathrm{I}}]$, où X_{I} est le nombre de clés examinées lors de sa recherche.

Question : chaînage, recherche infructueuse

Question (poly § 3.4.4) (Recherche infructueuse.) Soit $k \notin \{k_1, \ldots, k_n\}$. Calculer $\mathbb{E}[X_{\mathrm{I}}]$, où X_{I} est le nombre de clés examinées lors de sa recherche.

$$\mathbb{E}[X_{\mathrm{I}}] = \alpha$$

Question : chaînage, recherche fructueuse

Question (poly § 3.4.4) (Recherche fructueuse.) Soit $k \in \{k_1, \ldots, k_n\}$.
Calculer $\mathbb{E}[X_{\mathrm{F}}]$, où X_{F} est le nombre de clés examinées.

Question : chaînage, recherche fructueuse

Question (poly § 3.4.4) (Recherche fructueuse.) Soit $k \in \{k_1, \ldots, k_n\}$. Calculer $\mathbb{E}[X_{\mathrm{F}}]$, où X_{F} est le nombre de clés examinées.

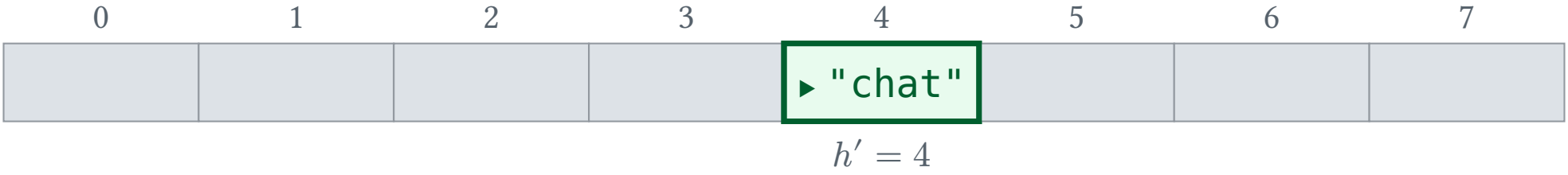
$$\mathbb{E}[X_{\mathrm{F}}] = 1 + \frac{n-1}{2m} \leq 1 + \frac{\alpha}{2}$$

coût moyen $O(1+\alpha)$: $O(1)$ **en moyenne** si α borné

Adressage ouvert : principe

- une case = **au plus un** couple, suite de **sondage** : linéaire, $s(k, i) = (h(k) + i)\%m$
- **rechercher** k : sonder jusqu'à la clé ou une case **vide**
- **ajouter** (k, v) : rechercher k, présente : on remplace la valeur, sinon : première case **libre** rencontrée
- **supprimer** k : marqueur « supprimé » (on ne peut pas vider la case)
- $\alpha < 1$, et même (clés + marqueurs) $< m$

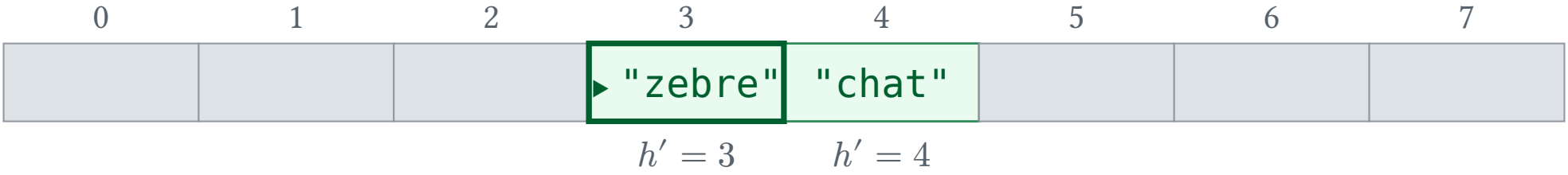
Adressage ouvert : exemple



facteur de charge $\alpha = n/m = 1/8$

- $m = 8$: deux cases vides d'abord, puis `rat` et `puma` en collision : rangés dans la première case **libre** qui suit

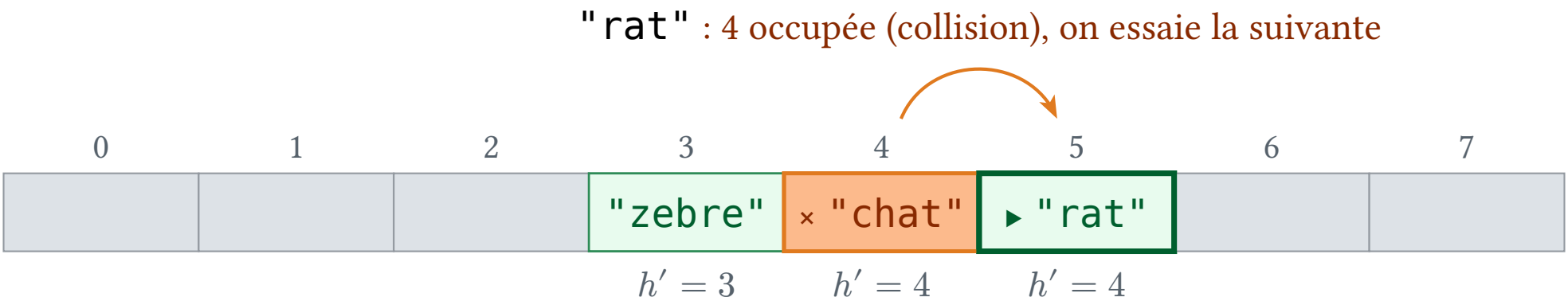
Adressage ouvert : exemple



facteur de charge $\alpha = n/m = 2/8$

- $m = 8$: deux cases vides d'abord, puis `rat` et `puma` en collision : rangés dans la première case **libre** qui suit

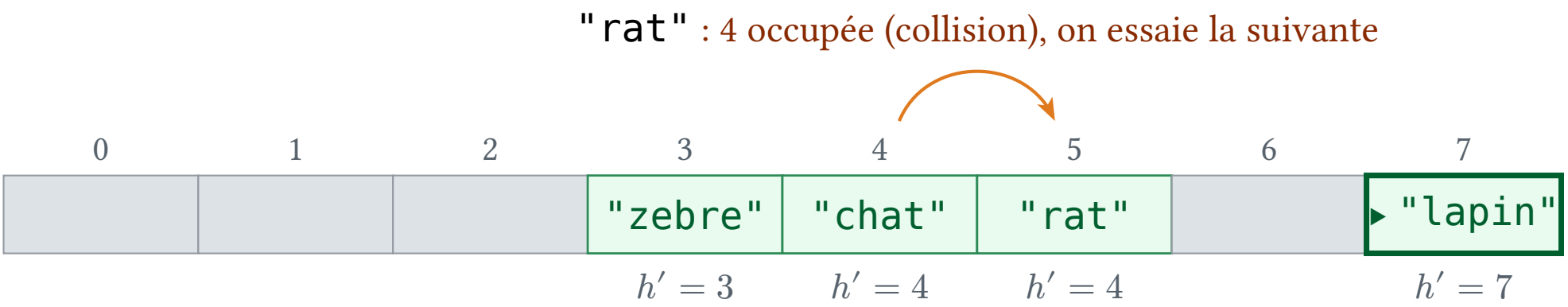
Adressage ouvert : exemple



facteur de charge $\alpha = n/m = 3/8$

- $m = 8$: deux cases vides d'abord, puis `rat` et `puma` en collision : rangés dans la première case **libre** qui suit

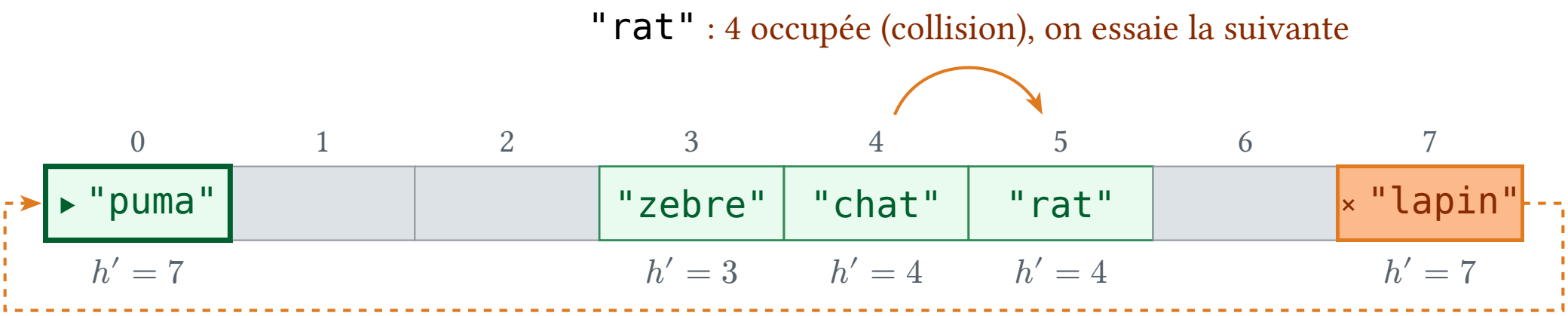
Adressage ouvert : exemple



facteur de charge $\alpha = n/m = 4/8$

- $m = 8$: deux cases vides d'abord, puis `rat` et `puma` en collision : rangés dans la première case **libre** qui suit

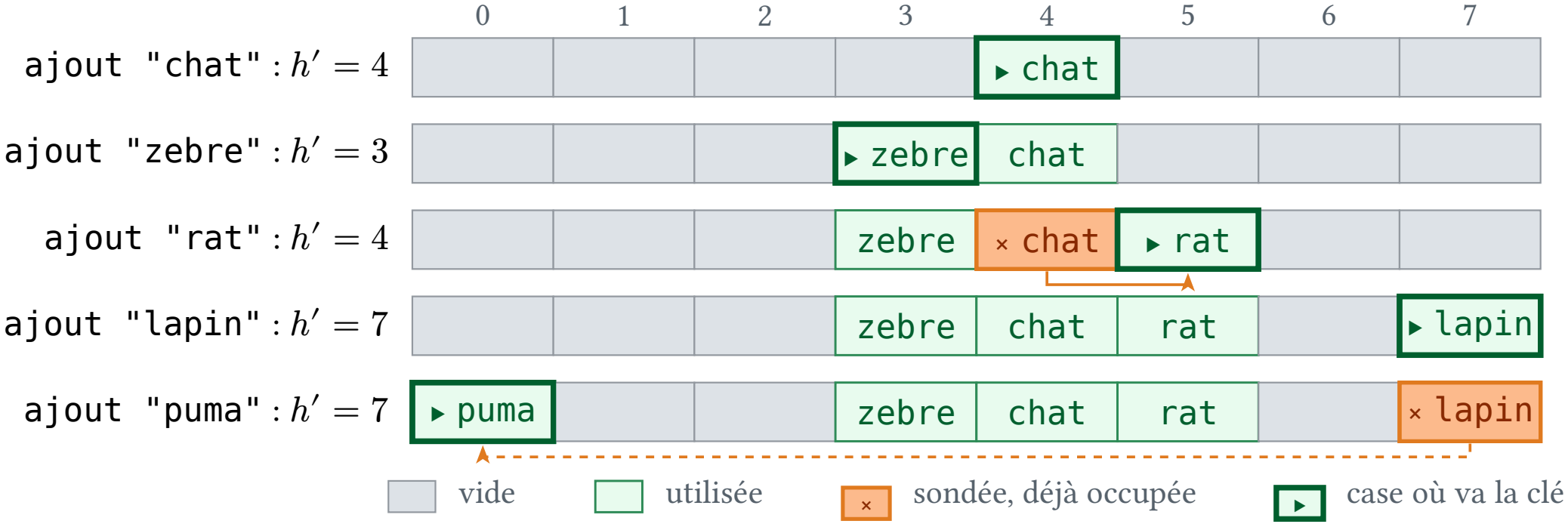
Adressage ouvert : exemple



facteur de charge $\alpha = n/m = 5/8$

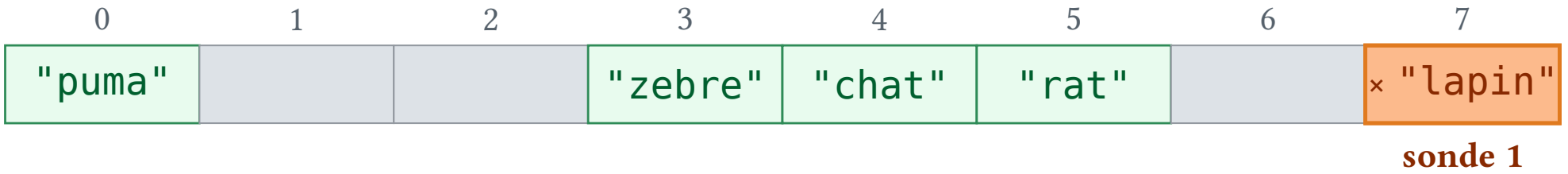
- $m = 8$: deux cases vides d'abord, puis `rat` et `puma` en collision : rangés dans la première case **libre** qui suit

Adressage ouvert : récapitulatif des insertions



Adressage ouvert : recherche

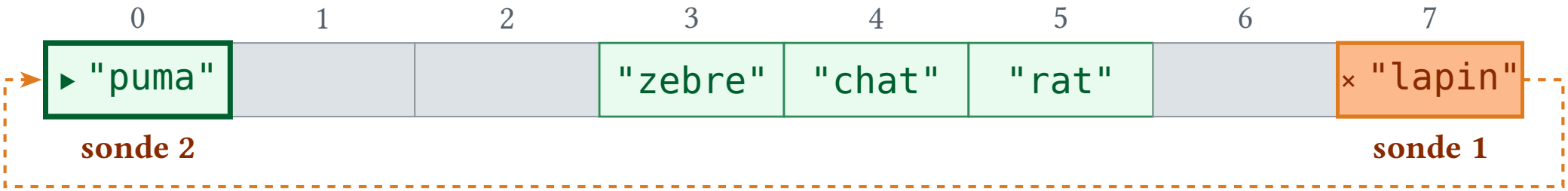
recherche de `"puma"` : $h' = \texttt{hache}(\texttt{"puma"})\%8 = 47\%8 = 7$



1. case 7 : `"lapin"` $\neq$ `"puma"`, on essaie la suivante, $(7+1)\%8 = 0$

Adressage ouvert : recherche

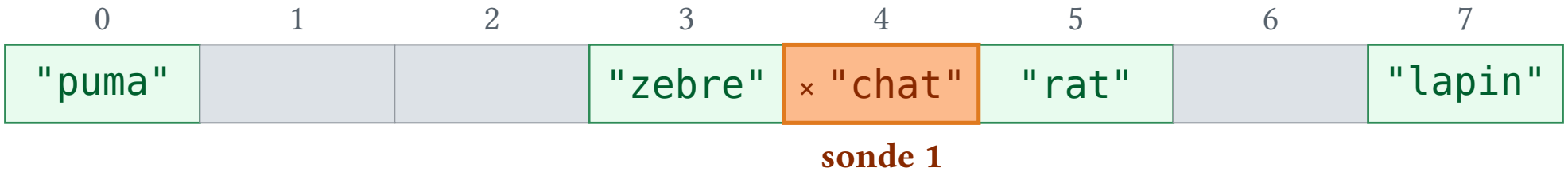
recherche de `"puma"` : $h' = \texttt{hache}(\texttt{"puma"}) \% 8 = 47 \% 8 = 7$



1. case 7 : `"lapin"` $\neq$ `"puma"`, on essaie la suivante, $(7+1)\%8 = 0$
2. case 0 : ▸ **trouvée**, recherche fructueuse, 2 sondages

Adressage ouvert : recherche

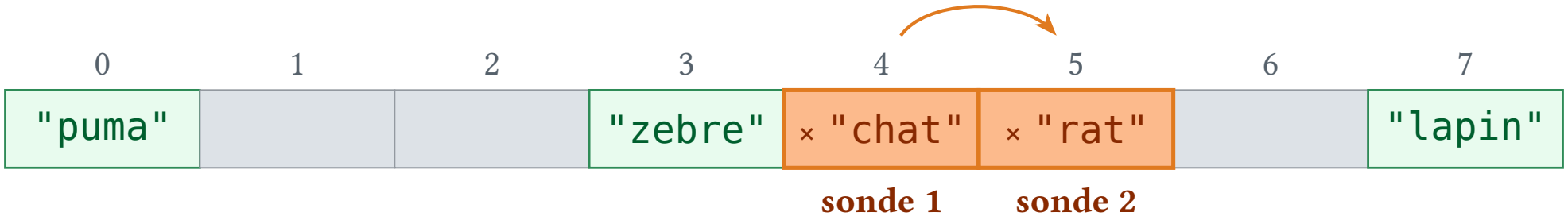
recherche de `"loup"` : $h' = \texttt{hache}(\texttt{"loup"}) \% 8 = 60 \% 8 = 4$



1. case 4 : `"chat"` $\neq$ `"loup"`, on essaie la suivante

Adressage ouvert : recherche

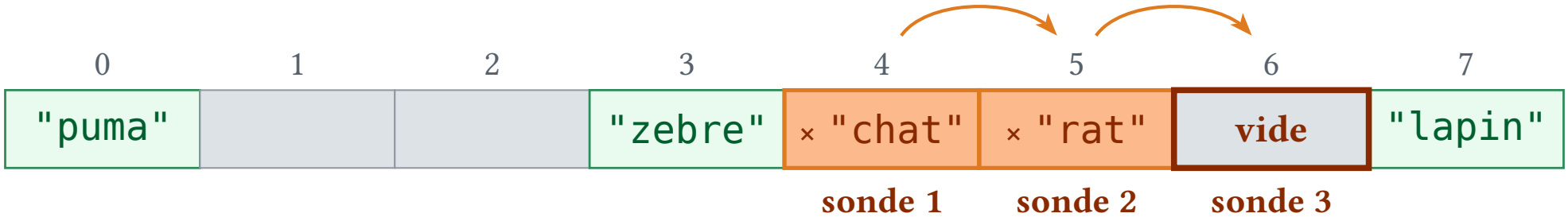
recherche de `"loup"` : $h' = \texttt{hache}(\texttt{"loup"}) \% 8 = 60 \% 8 = 4$



1. case 4 : `"chat"` $\neq$ `"loup"`, on essaie la suivante
2. case 5 : `"rat"` $\neq$ `"loup"`, on essaie la suivante

Adressage ouvert : recherche

recherche de `"loup"` : $h' = \texttt{hache}(\texttt{"loup"}) \% 8 = 60 \% 8 = 4$

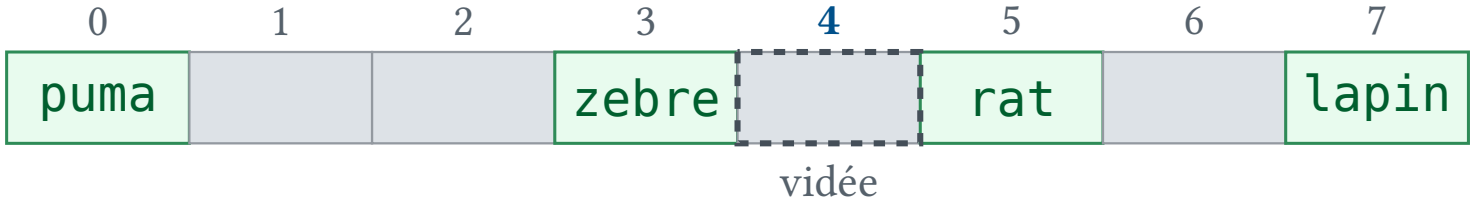


1. case 4 : `"chat"` $\neq$ `"loup"`, on essaie la suivante
2. case 5 : `"rat"` $\neq$ `"loup"`, on essaie la suivante
3. case 6 : **vide** : la clé est absente, recherche infructueuse, 3 sondages

Adressage ouvert : suppression

(a) si on vide la case

▾ recherche de `"rat"` : $h' = 4$

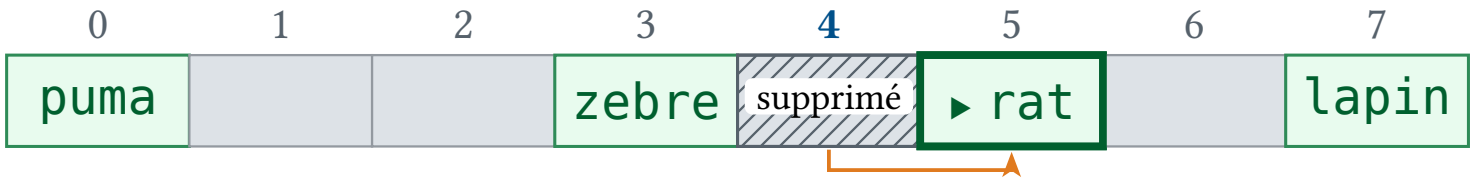


4 : vide, on s'arrête

× introuvable (faux !)

(b) avec un marquer

▾ recherche de `"rat"` : $h' = 4$



4 : marqueur, on continue

▸ 5 : trouvé

après suppression de `"chat"` (case 4), case hachurée : marqueur « supprimé », la recherche le saute

- si l'on vidait la case 4, la recherche de `"rat"` s'arrêterait à tort
- marqueur « supprimé » : la recherche le **saute**, l'ajout peut le **réutiliser**

Question : adressage ouvert en Python

Question (poly § 3.5.3) Avec le sondage linéaire, sans suppression, et `None` pour une case vide, écrire `creer_table(m)`, `rechercher(T, cle)` (valeur associée ou `None`) et `ajouter(T, cle, valeur)` (table supposée non pleine).

Question : adressage ouvert en Python

Question (poly § 3.5.3) Avec le sondage linéaire, sans suppression, et `None` pour une case vide, écrire `creer_table(m)`, `rechercher(T, cle)` (valeur associée ou `None`) et `ajouter(T, cle, valeur)` (table supposée non pleine).

```
def creer_table(m):
    return [None for _ in range(m)]  # vide

def rechercher(T, cle):
    m = len(T)
    i = hache(cle) % m
    for _ in range(m):        # ≤ m sondages
        if T[i] is None:      # vide : absente
            return None
        if T[i][0] == cle:
            return T[i][1]
        i = (i + 1) % m       # case suivante
    return None
```

```
def ajouter(T, cle, valeur):
    m = len(T)
    i = hache(cle) % m
    while (T[i] is not None
           and T[i][0] != cle):
        i = (i + 1) % m
    T[i] = (cle, valeur)
```

Question : adressage ouvert, recherche infructueuse

- aucune clé supprimée, coût = nombre de cases **sondées**

Sondage uniforme. suite de sondage = permutation uniforme de $[\![0, m-1]\!]$, indépendante des autres clés.

Question (poly § 3.5.4) (Recherche infructueuse.) Soit $k \notin \{k_1, \ldots, k_n\}$, avec $n < m$. Calculer $\mathbb{E}[X_{\mathrm{I}}]$, où X_{I} est le nombre de cases sondées lors de sa recherche, puis montrer que $\mathbb{E}[X_{\mathrm{I}}] \leq 1/(1-\alpha)$. *Indication :* $\mathbb{E}[X_{\mathrm{I}}] = \sum_{i \geq 1} \mathbb{P}(X_{\mathrm{I}} \geq i)$.

Question : adressage ouvert, recherche infructueuse

- aucune clé supprimée, coût = nombre de cases **sondées**

Sondage uniforme. suite de sondage = permutation uniforme de $[\![0, m-1]\!]$, indépendante des autres clés.

Question (poly § 3.5.4) (Recherche infructueuse.) Soit $k \notin \{k_1, \ldots, k_n\}$, avec $n < m$. Calculer $\mathbb{E}[X_{\mathrm{I}}]$, où X_{I} est le nombre de cases sondées lors de sa recherche, puis montrer que $\mathbb{E}[X_{\mathrm{I}}] \leq 1/(1-\alpha)$. *Indication :* $\mathbb{E}[X_{\mathrm{I}}] = \sum_{i \geq 1} \mathbb{P}(X_{\mathrm{I}} \geq i)$.

$$\mathbb{E}[X_{\mathrm{I}}] = \frac{m+1}{m-n+1} \leq \frac{1}{1-\alpha}$$

Question : adressage ouvert, recherche fructueuse

Question (poly § 3.5.4) (Recherche fructueuse.) Soit $k \in \{k_1, \ldots, k_n\}$, avec $n < m$. Calculer $\mathbb{E}[X_{\mathrm{F}}]$, où X_{F} est le nombre de cases sondées lors de sa recherche, puis montrer que $\mathbb{E}[X_{\mathrm{F}}] \leq \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$.

Question : adressage ouvert, recherche fructueuse

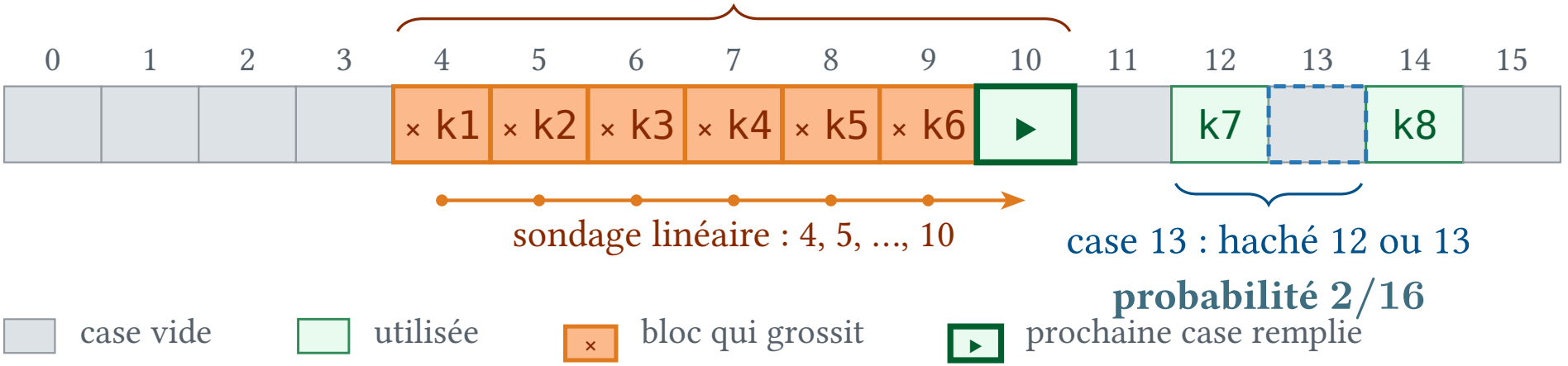
Question (poly § 3.5.4) (Recherche fructueuse.) Soit $k \in \{k_1, \ldots, k_n\}$, avec $n < m$. Calculer $\mathbb{E}[X_{\mathrm{F}}]$, où X_{F} est le nombre de cases sondées lors de sa recherche, puis montrer que $\mathbb{E}[X_{\mathrm{F}}] \leq \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$.

$$\mathbb{E}[X_{\mathrm{F}}] = \frac{m+1}{n} \sum_{t=m-n+2}^{m+1} \frac{1}{t} \leq \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

Regroupement primaire

probabilité d'allonger ce bloc : 7/16

haché dans $\{4, \ldots, 10\}$ (7 valeurs) $\Rightarrow$ clé rangée en 10



un bloc occupé grossit d'autant plus vite qu'il est long : c'est le *regroupement primaire*

- sondage linéaire : les blocs **grossissent**
- Knuth : $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$ sondages (infructueuse)

Chaînage ou adressage ouvert : aucun n'est meilleur

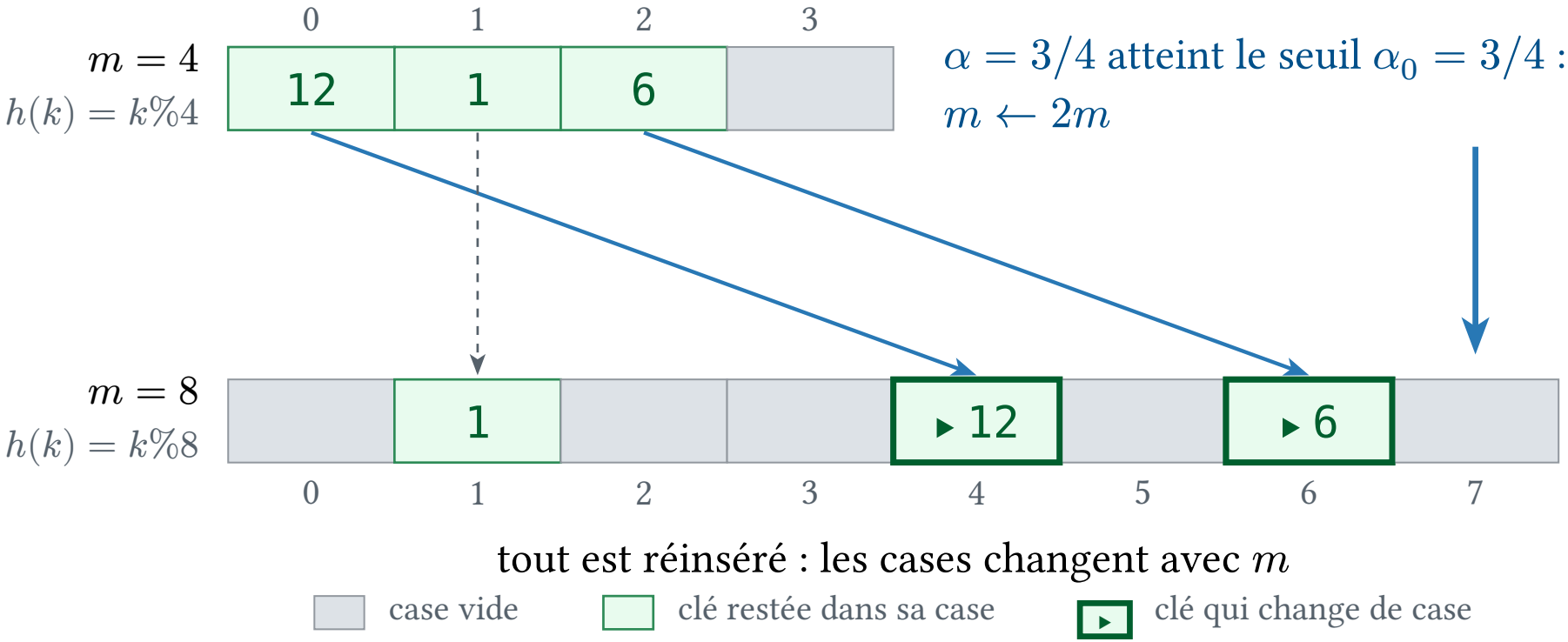
	Chaînage	Adressage ouvert
facteur de charge	+ $\alpha > 1$ possible, dégradation douce ($1 + \alpha$)	− $\alpha < 1$ obligatoire, dégradation brutale quand $\alpha \to 1$
mémoire	− une liste par alvéole : surcoût par clé	+ un seul tableau : à mémoire égale, α plus petit
collisions	+ une collision ne gêne que son alvéole	− regroupements, sensible au hachage et au sondage
suppression	+ simple : on retire le couple	− marqueur « supprimé », qui allonge les recherches
rapidité réelle	− listes dispersées en mémoire	+ cases voisines (*cache*), pas d'allocation
simplicité	+ très simple	+ simple sans suppression, plus délicat avec

Les deux : $O(1)$ en moyenne si α reste borné.

Dans les langages courants

Langage	Structure	Stratégie
Java	`HashMap`	chaînage
C++	`std::unordered_map`	chaînage
Python	`dict`	adressage ouvert
Rust	`HashMap`	adressage ouvert
Julia	`Dict`	adressage ouvert (sondage linéaire)
Lua	tables	mixte : listes chaînées rangées dans les cases libres

Redimensionnement (HP)



- $\alpha \geq \alpha_0$: table de taille $2m$, on **réinsère** tout ($O(n)$ en moyenne)
- doublement : ajout en $O(1)$ **amorti** (comme `append`)

Qu'est-ce qu'une bonne fonction de hachage ? (HP)

1. **déterministe**
2. **rapide**
3. bien **répartie**, même pour des clés qui se ressemblent

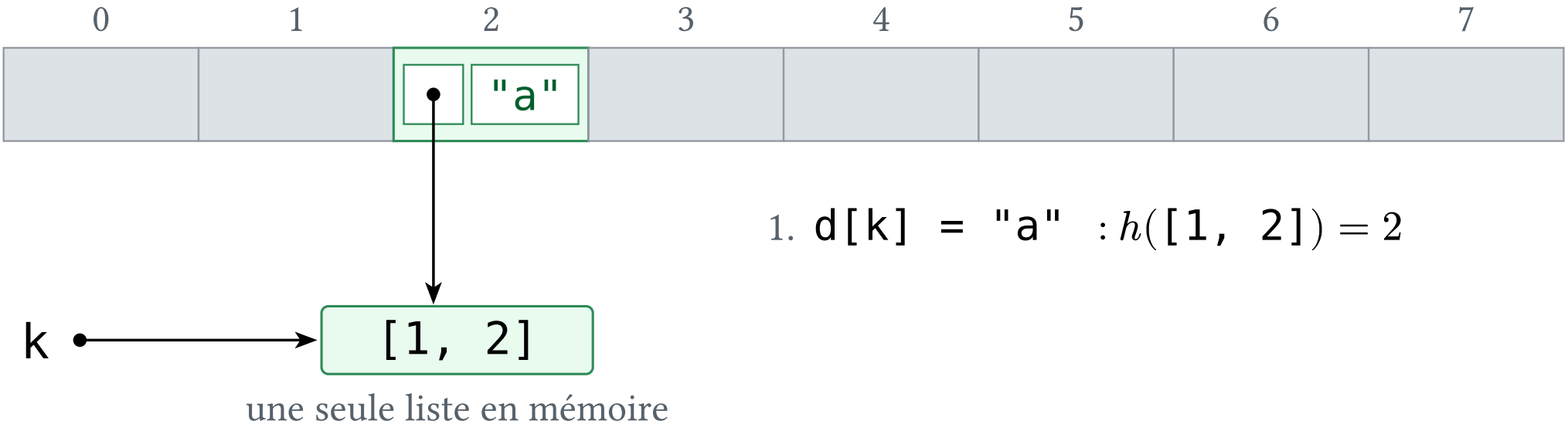
- entiers : $k\%m$ (m premier), $\lfloor m\{k\theta\}\rfloor$
- chaînes : `hache_poly`$(c_0 ... c_{\ell-1}) = \left(c_0 b^{\ell-1} + ... + c_{\ell-1}\right)\%M$, en $O(\ell)$ par Horner
- universel : $h_{a,b}(k) = ((ak+b)\%p)\%m$ tirée au hasard
- à éviter : 1re lettre (26 valeurs mal réparties), somme des rangs `hache` (petites valeurs, anagrammes en collision)

Les dictionnaires de Python : la fonction `hash` (Python)

```
>>> hash(12), hash(2**61 + 42), hash(-3)
(12, 43, -3)
>>> hash("chat")        # change à chaque lancement de Python !
3192665300061180367
>>> hash([1, 2])
TypeError: unhashable type: 'list'
```

- entiers : $n \% (2^{61} - 1)$, non randomisés, curiosité : `hash(-1) == -2`
- chaînes : SipHash à clé secrète **aléatoire**

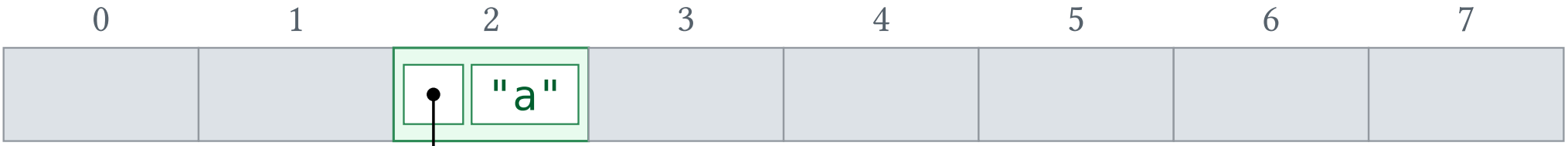
Quelles clés sont autorisées ?

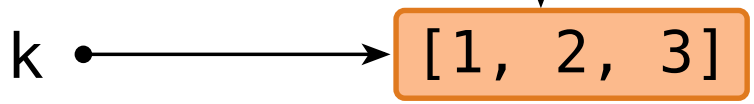


1. `d[k] = "a"` : $h(\texttt{[1, 2]}) = 2$

- clé **hachable** = **immuable** : `int`, `float`, `bool`, `str`, `tuple`, pas `list`, `dict`, `set`
- `a == b` $\Rightarrow$ `hash(a) == hash(b)` : `1`, `1.0`, `True` sont la même clé

Quelles clés sont autorisées ?



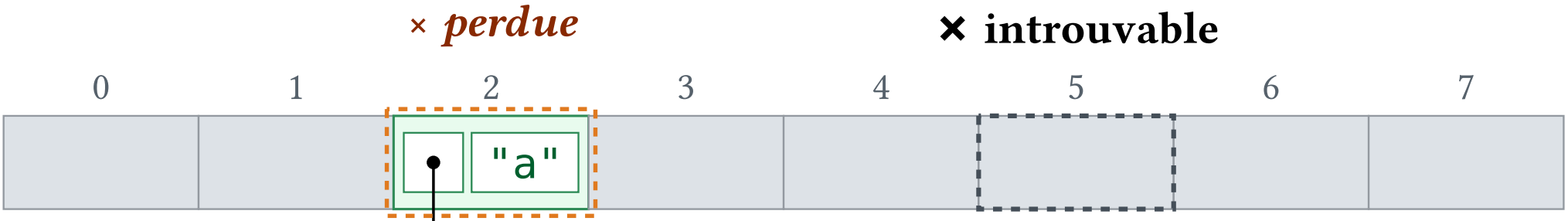


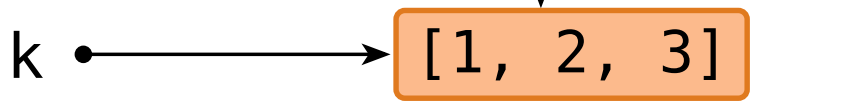
une seule liste en mémoire

1. `d[k] = "a"` : $h(\texttt{[1, 2]}) = 2$
2. `k.append(3)` : la clé rangée change aussi

- clé **hachable** = **immuable** : `int`, `float`, `bool`, `str`, `tuple`, pas `list`, `dict`, `set`
- `a == b` $\Rightarrow$ `hash(a) == hash(b)` : `1`, `1.0`, `True` sont la même clé

Quelles clés sont autorisées ?



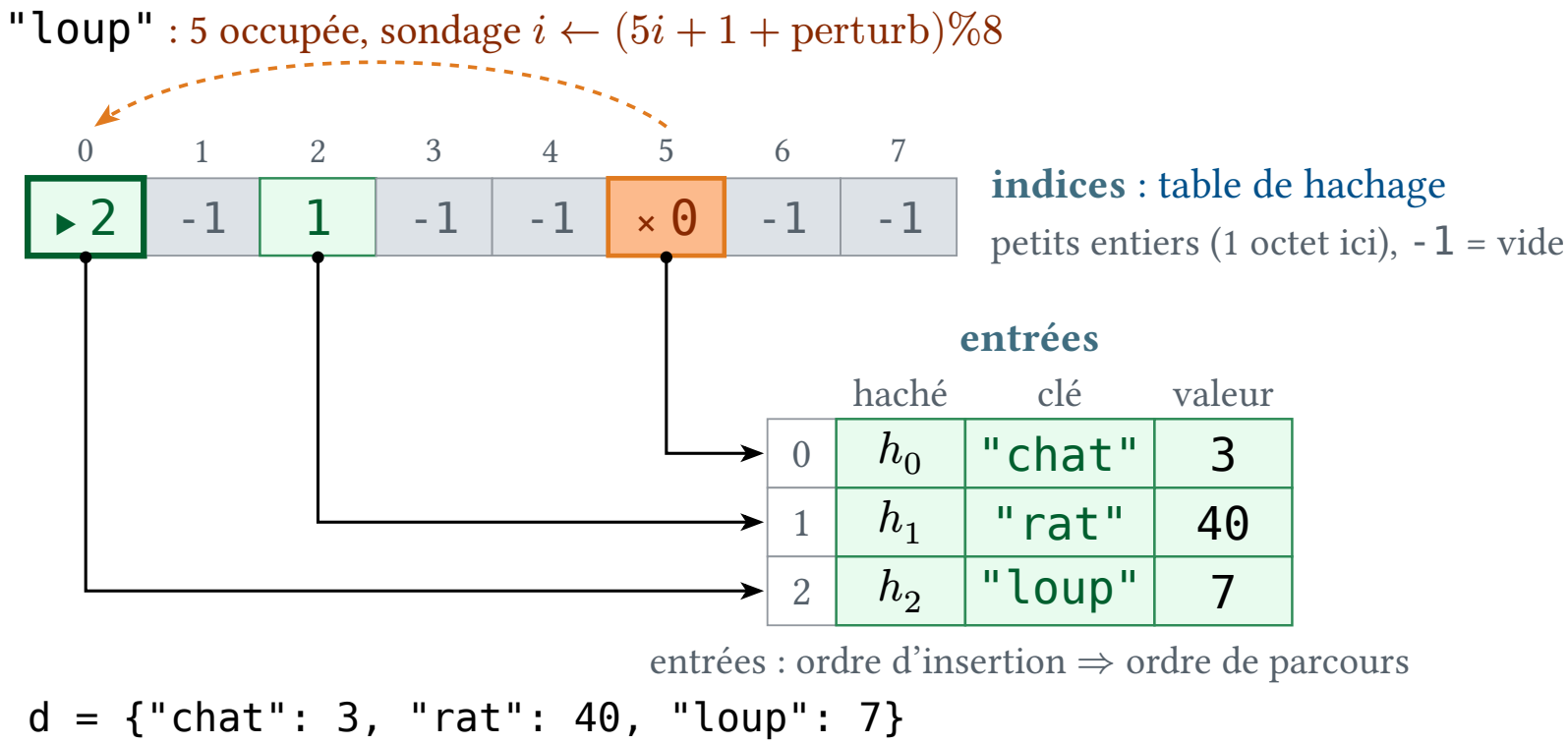


une seule liste en mémoire

1. `d[k] = "a"` : $h(\texttt{[1, 2]}) = 2$
2. `k.append(3)` : la clé rangée change aussi
3. `k in d` : $h(\texttt{[1, 2, 3]}) = 5$, alvéole vide

- clé **hachable** = **immuable** : `int`, `float`, `bool`, `str`, `tuple`, pas `list`, `dict`, `set`
- `a == b` $\Rightarrow$ `hash(a) == hash(b)` : `1`, `1.0`, `True` sont la même clé

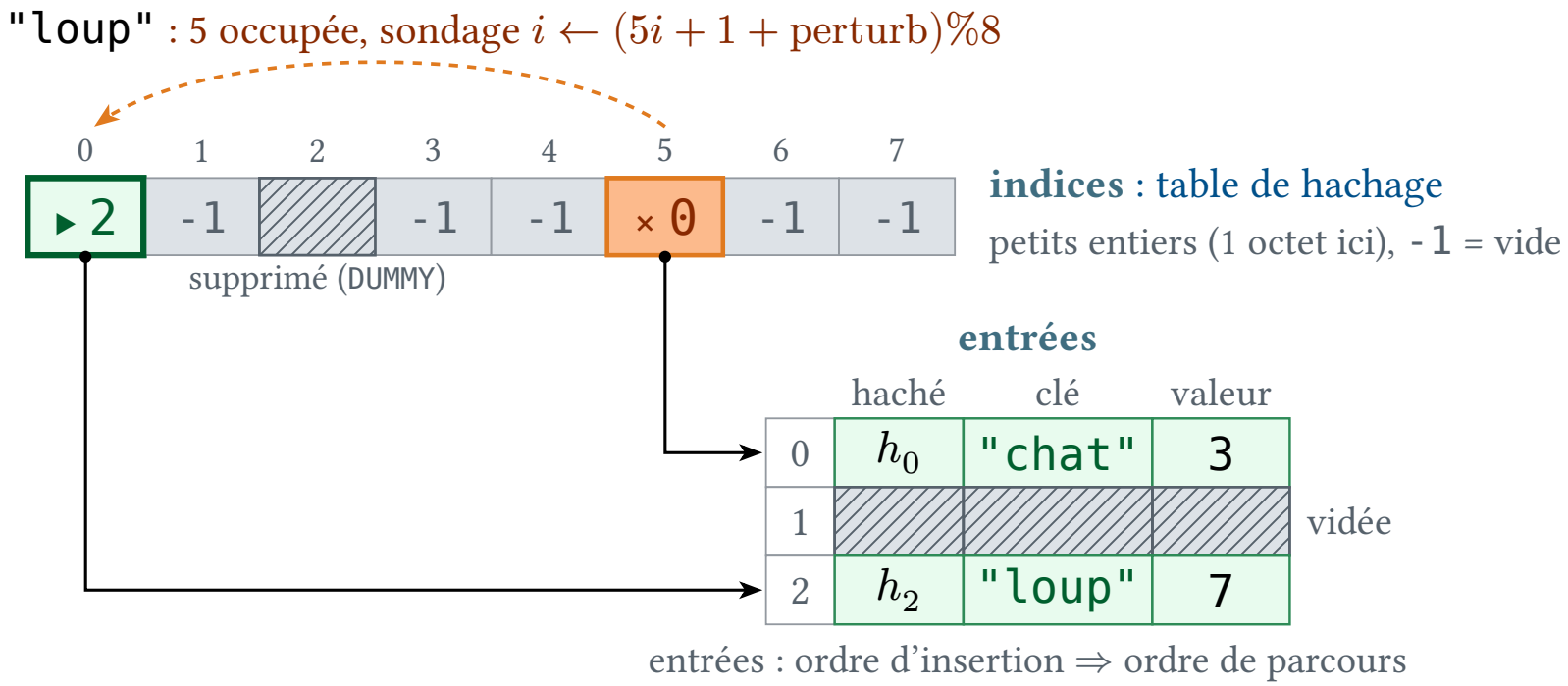
L'implémentation de `dict` (HP)




```
d = {"chat": 3, "rat": 40, "loup": 7}
```

- adressage ouvert, $m = 2^p$, $\alpha \leq 2/3$, sondage « perturbé »
- entrées dans l'**ordre d'insertion**, « vide » $= -1$, « supprimé » $= -2$

L'implémentation de `dict` (HP)




```
d = {"chat": 3, "rat": 40, "loup": 7}
```

- adressage ouvert, $m = 2^p$, $\alpha \leq 2/3$, sondage « perturbé »
- entrées dans l'**ordre d'insertion**, « vide » $= -1$, « supprimé » $= -2$

Utiliser les dictionnaires (Python)

Syntaxe	Effet	Coût moyen
`d = {}`	dictionnaire vide	$O(1)$
`d = {"chat": 3, "rat": 40}`	création de n couples	$O(n)$
`d = {x: 0 for x in L}`	création par compréhension	$O(n)$
`d[k] = v`	ajout ou modification	$O(1)$
`d[k]`	valeur (`KeyError` si absente)	$O(1)$
`k in d`	test sur les **clés**	$O(1)$
`len(d)`	nombre de couples	$O(1)$
`for k in d.keys():` (ou `for k in d:`)	parcours des clés	$O(n)$
`for k, v in d.items():`	parcours des couples	$O(n)$
`d.copy()`	copie superficielle	$O(n)$
`v in d.values()`	test sur les **valeurs** (hors annexe)	$O(n)$

en **moyenne**, et **amorti** pour l'ajout, motif `if k in d: … else: …`

Exercices : programmes usuels

Travail personnel : programmes à savoir écrire

- `occurrences(L)`
- `max_argmax(L)`, `deux_max(L)`
- `point_le_plus_bas(P)`
- `plus_frequent(L)`
- `distincts(L)`, `a_un_doublon(L)`
- `paire_de_somme(L, s)`
- `est_auto_evitante(chemin)`
- `plus_petit_absent(L)`
- `cycle(f, u0)`
- `intersection(L1, L2)`
- `positions(L)`, `inverse(d)`
- `anagrammes(mots)`
- `jointure(T1, T2, a, b)`
- `adjacence(sommets, aretes)`
- `degres(g)`, `voisins(s, sommets, aretes)`
- `moyenne_par_categorie(couples)`
- `somme_poly(p, q)`, `produit_poly(p, q)`
- `decoder(bits, code)`

Document séparé : `cours_dico_applications`. Conventions : pas de `x in L`, `if k in d: … else: …`, ensemble = dictionnaire à valeurs `True`