Informatique tronc commun

Structures de données et dictionnaires

Poly de cours à compléter

1 Introduction

1.1 Le programme

En **première année**, les dictionnaires sont utilisés « en boîte noire », et le choix entre liste et dictionnaire doit être « éclairé » :

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.* extrait programme ITC, 1re année

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. extrait programme ITC, 1re année

En **deuxième année**, il faut en comprendre le fonctionnement :

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. extrait programme ITC, 2e année

Enfin, l'annexe du programme fixe les opérations Python exigibles sur les listes et les dictionnaires. Les autres (`insert`, `pop(i)`, `del`, `index`…) ne sont utilisables aux concours que si l'énoncé les introduit explicitement :

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`. extrait programme ITC, annexe A

On en profite pour retravailler :

- les **structures de données** usuelles et leurs coûts,
- l'**implémentation** de structures de données,
- les **calculs de complexité**,
- le **choix** d'une structure de données adaptée à un problème.

1.2 Aux concours

Le fonctionnement interne d'une table de hachage est rarement demandé explicitement. Il suffit de savoir **utiliser selon le contexte** une liste, un tableau ou un dictionnaire. Plus précisément, les rapports de jury insistent sur la nécessité de :

- **choisir**, par exemple un dictionnaire, pour obtenir la bonne complexité,
- **raisonner avec les coûts** des opérations sur les listes, qui sont rappelés (ou restreints !) en préambule,
- **copier** et créer correctement une liste : c'est l'une des erreurs les plus citées dans les rapports.

Il est aussi important de savoir travailler avec un nombre restreint d'opérations et de s'y ramener.

Et comme toujours, dans le doute, *faire un schéma !*

Voici par exemple le préambule du sujet X-ENS 2025 :

> **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`.
>
> extrait X-ENS 2025

> Remarque : ce préambule contient une erreur : en Python, `for (k, v) in d` parcourt les **clés** de `d` (et tente de décomposer chacune en couple), pour parcourir les couples, il faut écrire `for (k, v) in d.items()`. Mais **l'énoncé a toujours raison**.

Le sujet X-ENS 2023 précise que le $O(1)$ des dictionnaires est une « hypothèse simplificatrice » :

> **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 $\mathcal{O}(1)$ $(*)$.
>
> extrait X-ENS 2023

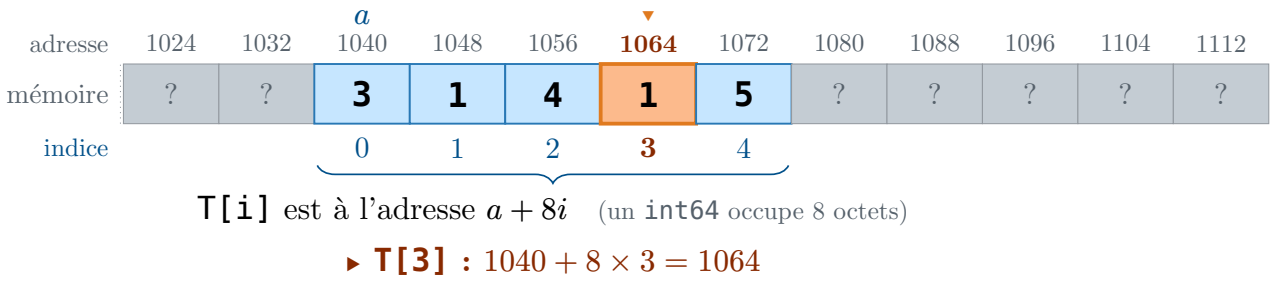
Même s'il n'est donc pas forcément nécessaire de justifier ces complexités, il est important de comprendre les mécanismes et les choix sous-jacents (et c'est une bonne excuse pour réviser les structures de données et la complexité).

2 Structures de données et leurs coûts

2.1 La mémoire et les tableaux

De manière simplifiée, la mémoire d'un ordinateur se comporte comme un **immense tableau d'octets**, chaque case étant repérée par un entier, son **adresse**. Accéder à une case dont on connaît l'adresse prend un temps constant.

Définition. Un **tableau** est une zone _____ de la mémoire, de taille _____, contenant des éléments _____. Le type fixe la taille t (en octets) de chaque élément, donc la taille totale du tableau. Si le tableau commence à l'adresse a, l'élément d'indice i se trouve à l'adresse $a + it$.



Un tableau de 5 entiers sur 64 bits (8 octets chacun) commençant à l'adresse 1040.

Pour un tableau de taille n :

Opération	Coût	Pourquoi
création (de taille n)	_____	réservation et initialisation des n cases
accès `T[i]`	_____	calcul d'adresse, puis lecture
modification `T[i] = x`	_____	calcul d'adresse, puis écriture
recherche d'une valeur	_____	parcours

Attention. La taille d'un tableau est _____ à sa création : la case qui suit le tableau est peut-être occupée par autre chose. On ne peut donc pas « ajouter » un élément à un tableau : on peut seulement **modifier** ses éléments, ou **créer un nouveau tableau** (plus grand) et y recopier l'ancien, en $O(n)$.

2.1.1 Les tableaux `numpy`

Un tableau `numpy` (`ndarray`) est exactement ce modèle : le type commun des éléments est son `dtype`, et sa taille est fixe.

```
import numpy as np
T = np.zeros(5, dtype=np.int64)  # création : O(n), le paramètre dtype est HP
T[2] = 4                         # modification : O(1)
```

En plus des différences usuelles entre listes et tableaux (coût pour accéder à un élément, possibilité ou non de changer la taille, type des éléments, etc.), les tableaux `numpy` ont quelques particularités par rapport aux « listes » Python :

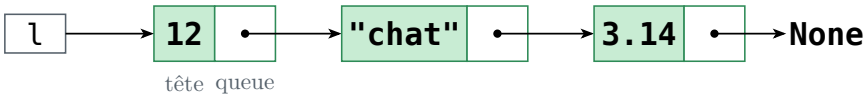
- une opération sur un tableau `numpy` est souvent **vectorisée** : elle agit sur tous les éléments en une seule fois (le coût reste en $O(n)$),
- lors d'un **slicing** `T[1:4]` d'un tableau `numpy`, on obtient une *vue* sur les mêmes cases, et non une copie.

```
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])
```

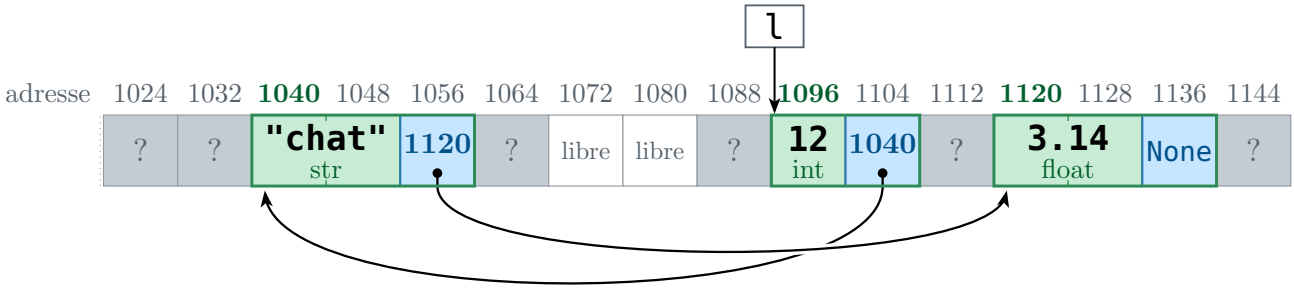
```
L = [3, 1, 4, 1, 5]
L * 2   # [3, 1, 4, 1, 5, 3, 1, 4, 1, 5]
W = L[1:4]       # copie
W[0] = 100
L   # [3, 1, 4, 1, 5]
```

2.2 Les listes chaînées

Une **liste chaînée** (parfois simplement appelée liste) n'est pas de taille fixe. La structure est la suivante : chaque élément est rangé dans une **cellule** placée n'importe où en mémoire, qui contient la valeur (la *tête*) et l'adresse de la cellule suivante (la *queue*). Une valeur spéciale marque la fin de la liste.



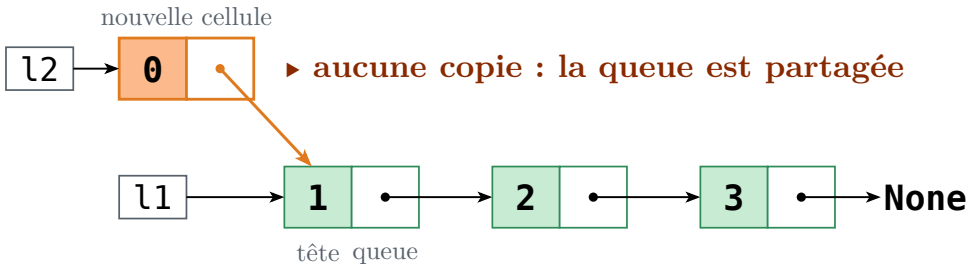
Dans la mémoire, les cellules sont dispersées, dans n'importe quel ordre, et leurs valeurs n'ont pas forcément la même taille : c'est l'adresse contenue dans chaque cellule qui donne l'ordre de la liste. Accéder à l'élément d'indice i, c'est suivre i adresses.



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

Python ne fournit pas de listes chaînées. Pour étudier leur fonctionnement, on peut en construire une, de façon purement théorique (on ne s'en servira pas en pratique) : la liste vide est `None` et une cellule est un couple `(valeur, suivant)`. Un tuple étant **immuable**, une cellule n'est jamais modifiée : c'est ce qui rend sans danger le partage de la queue entre `l1` et `l2` ci-dessous.

```
l1 = (1, (2, (3, None)))        # la liste 1, 2, 3
l2 = (0, l1)                    # ajout de 0 en tête : une seule cellule créée
```



Question. Écrire les fonctions :

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

La concaténation de deux listes chaînées s'écrit comme `ajout_fin` : on recopie les cellules de `l1`, et la dernière copie pointe vers `l2`.

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

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

Pour une liste chaînée de longueur n :

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

HORS PROGRAMME · listes d'OCaml

Les listes d'OCaml (langage de l'option informatique) sont des listes chaînées immuables, exactement comme nos tuples : `[]` est notre `None`, `x :: l` notre `ajout_tete`, `l1 @ l2` notre `concatener` (en $O(n_1)$). L'accès à l'élément d'indice i s'écrit par filtrage, comme notre `acces` (la bibliothèque fournit aussi `List.nth l i`) :

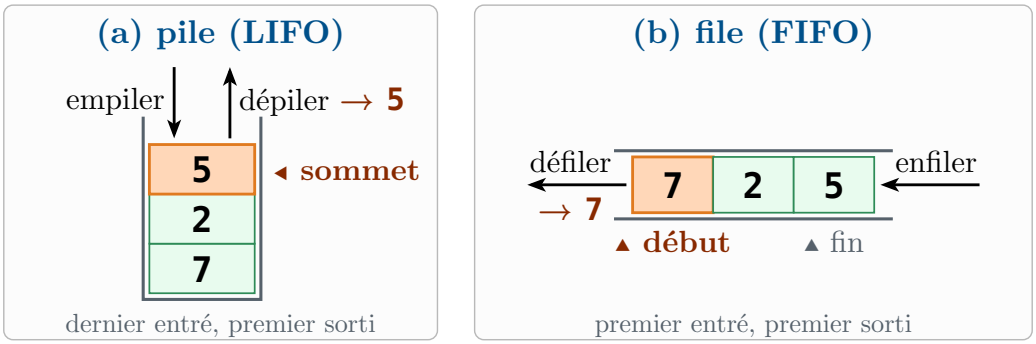
```
let rec acces l i = match l with
  | [] -> failwith "indice invalide"
  | x :: q -> if i = 0 then x else acces q (i - 1)
```

2.3 Rappel : piles, files et `deque`

On remarque qu'une liste chaînée implémente de manière naturelle une structure de _____.

Rappel de première année :

- une **pile** (*LIFO* : dernier entré, premier sorti) : on **empile** et on **dépile** au sommet,
- une **file** (*FIFO* : premier entré, premier sorti) : on **enfile** à la fin et on **défile** au début.



Pour une file, avec des cellules **modifiables**, il suffirait de conserver en plus l'adresse de la **dernière** cellule pour enfiler en fin en $O(1)$.

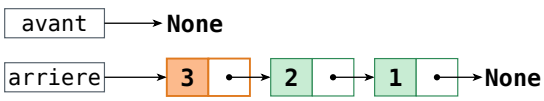
Avec nos cellules immuables, on représente une file par **deux** listes chaînées :

- `avant` contient les premiers éléments de la file, dans l'ordre : on **défile** en tête de `avant`, en $O(1)$,
- `arriere` contient les derniers arrivés, dans l'ordre inverse : on **enfile** en tête de `arriere`, en $O(1)$.

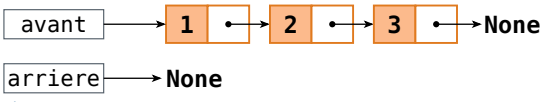
Pour défiler :

- si `avant` n'est pas vide, on renvoie sa tête, en $O(1)$,
- si `avant` est vide, on la remplace d'abord par `arriere` **retournée** (et `arriere` par la liste vide), puis on renvoie sa tête.

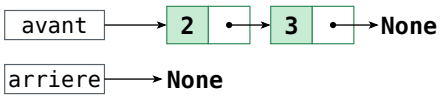
Le retournement n'a donc lieu que lorsque `avant` est vide, et non à chaque défilement. Il coûte un temps proportionnel à la longueur de `arriere`, mais chaque élément n'est retourné qu'une seule fois (il passe de `arriere` à `avant`, puis sort de la file) : enfiler et défiler sont en $O(1)$ **amorti**.



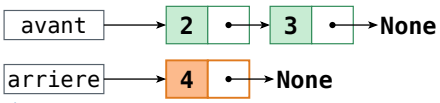
Étape 1. enfiler 1, 2, 3



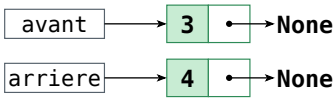
Étape 2. défiler (`avant` vide) : on y met `arriere` retournée…



Étape 3. … et on renvoie `1`



Étape 4. enfiler 4



Étape 5. défiler (`avant` non vide) : on renvoie `2`

Une file représentée par deux listes chaînées (étapes 1 à 5, dans l'ordre de lecture) : `arriere` n'est retournée que si l'on défile alors que `avant` est vide (étape 2).

En Python, on utilise la classe `deque` (*double-ended queue*) du module `collections` : ajout et retrait en $O(1)$ aux deux extrémités.

```
from collections import deque
d = deque()                    # file vide
```

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

2.4 Les tableaux dynamiques

On a vu que :

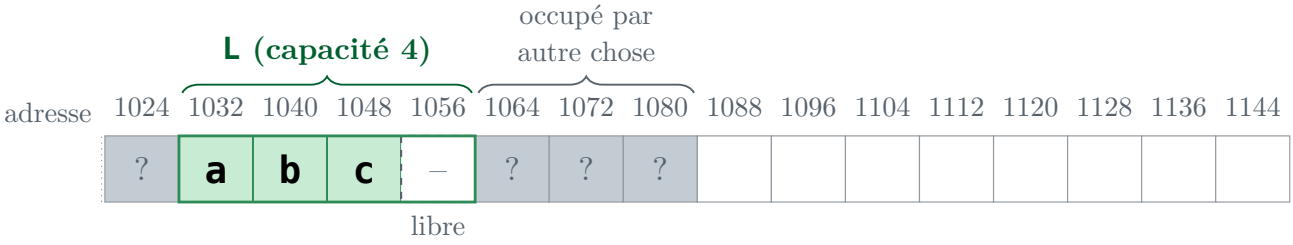
- pour un tableau, l'accès à un élément est en $O(1)$, mais l'ajout en fin est impossible,
- pour une liste chaînée, on peut ajouter des éléments, mais l'accès à un élément est en $O(n)$.

Pourtant, dans une « liste » Python, on peut accéder à un élément en $O(1)$ et ajouter en fin en $O(1)$. Comment est-ce possible ?

Les « listes » Python sont en fait des **tableaux dynamiques**.

Un **tableau dynamique** est un tableau dont on ne remplit que le début : il retient sa **taille** n (nombre de cases utilisées) et sa **capacité** $c \geq n$ (nombre de cases réservées).

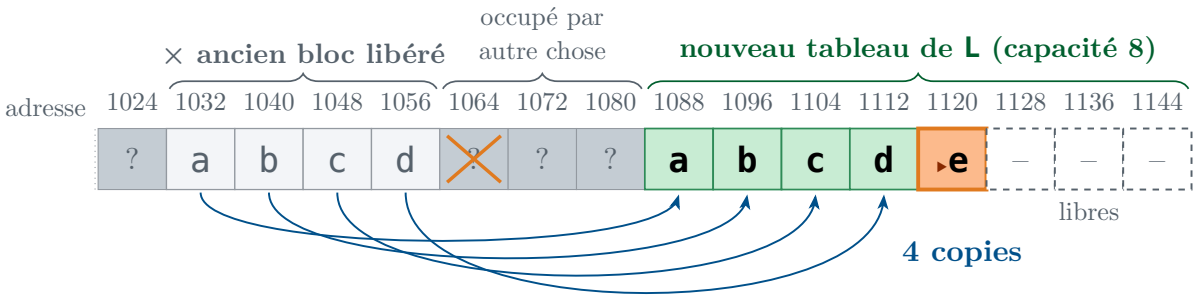
- Tant que $n < c$, ajouter en fin revient à écrire dans la case n, qui est réservée : coût $O(1)$.



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

Capacité non atteinte : l'ajout se fait dans une case réservée.

- Quand le tableau est plein ($n = c$), la case suivante est peut-être occupée par autre chose : il faut **réallouer**, c'est-à-dire réserver ailleurs un tableau plus grand, y recopier les n éléments ($O(n)$), puis écrire le nouvel élément.



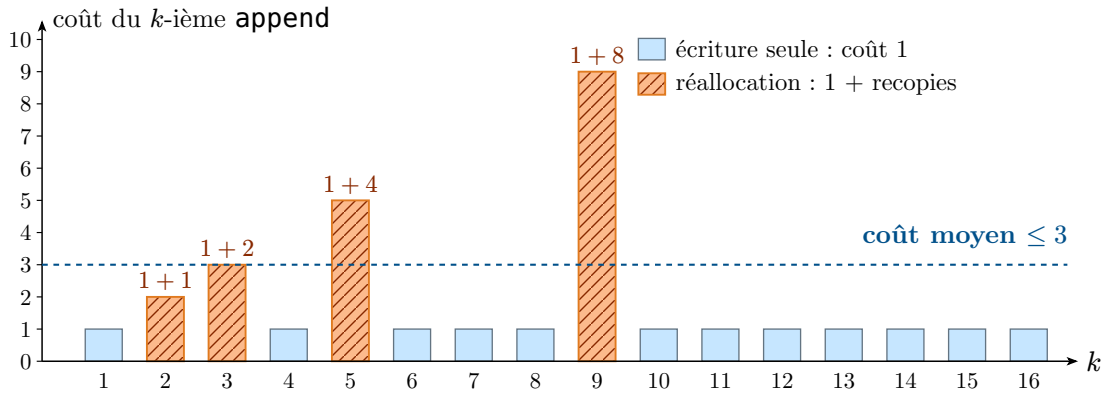
1. `L.append(d)` : on écrit dans la case réservée : $O(1)$
2. `L.append(e)` : la case suivante (1064) est occupée × : impossible de grandir sur place
3. nouveau bloc de capacité 8 : 4 copies, puis ▸ **écriture de `e`** ; l'ancien bloc est libéré

Le choix de la nouvelle capacité a un coût en **temps** (lorsqu'il faut recopier le tableau) et en **mémoire** (les cases réservées mais inutilisées). L'analyse de ces coûts dépend des capacités successives choisies.

Notons $c_0 < c_1 < c_2 < \dots$ les capacités successives. Partant d'un tableau vide, N ajouts coûtent N écritures, plus les recopies des réallocations, qui ont lieu quand la taille atteint $c_0, c_1, \dots$. En notant $C(N)$ la complexité de l'ajout de N éléments, on a

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

Question. Calculer $C(N)$ dans les deux cas suivants :
1. on agrandit le tableau de manière arithmétique : on rajoute r cases à chaque réallocation ($c_j = rj$),
2. on agrandit le tableau de manière géométrique : on multiplie la capacité par $q > 1$ à chaque réallocation ($c_{j+1} \geq qc_j$).



Doublement : coût du k-ième ajout (1 écriture, plus $k-1$ recopies si le tableau était plein).

Coût amorti. Une opération est de coût amorti $O(1)$ si toute suite de N opérations (depuis la structure vide) coûte _____ au total, même si certaines opérations isolées coûtent plus cher.

On a vu qu'avec une croissance géométrique, l'ajout en fin est en _____ amorti. Le choix de q est un compromis : un grand q fait moins de recopies, mais gaspille plus de mémoire. Si la capacité est exactement multipliée par q, jusqu'à $(q-1)n$ cases environ sont réservées et inutilisées.

2.4.1 Ce que fait Python

Une `list` Python est un tableau dynamique… de **références** : chaque case désigne un objet rangé ailleurs en mémoire, en contenant concrètement son **adresse**. Les adresses ont toutes la même taille (8 octets), ce qui permet de mélanger des éléments de types différents.

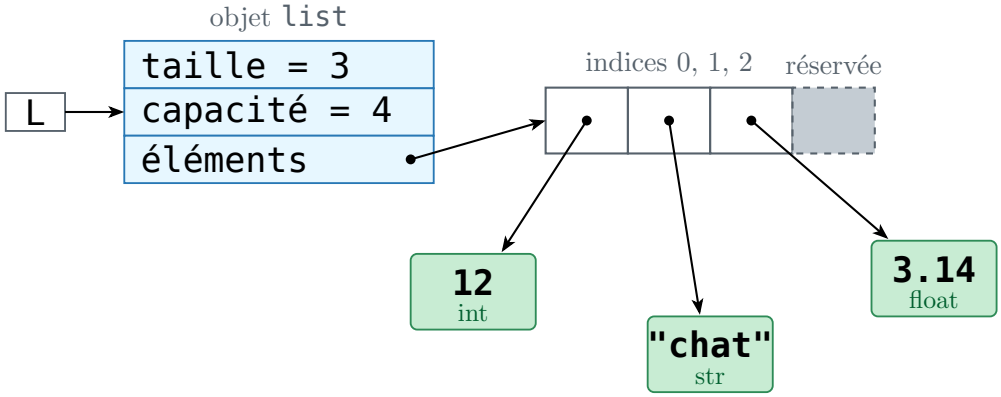
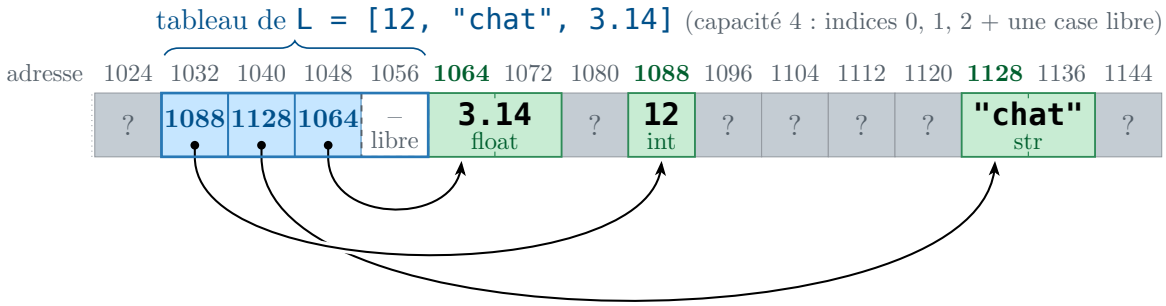


tableau de références, **contigu** ; objets **n'importe où** en mémoire

Vue dans la mémoire, une référence n'est qu'un entier, l'adresse de la case où commence l'objet :



une case du tableau contient l'**adresse** d'un objet

Quand une liste de taille n est pleine, Python choisit la nouvelle capacité $(n+1) + \lfloor (n+1)/8 \rfloor + 6$, arrondie au multiple de 4 inférieur. Les capacités successives sont

$$0, 4, 8, 16, 24, 32, 40, 52, 64, 76, 92, \ldots$$

soit une croissance géométrique de raison $q \approx 9/8 = 1{,}125$: `append` est en $O(1)$ amorti (avec $C(N) \leq 10N$ environ d'après la question précédente), et, asymptotiquement, moins de 12,5 % de la mémoire est gaspillée.

2.5 Tableau récapitulatif

Pour les listes Python, avec L et M de longueurs n et m :

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

Attention. Le test `x in L` sur une **liste** parcourt toute la liste : il est en _____, et non en $O(1)$. Glissé dans une boucle, il rend facilement un programme quadratique. Par précaution, mieux vaut ne jamais l'utiliser (il est d'ailleurs souvent interdit par les énoncés) et écrire la boucle explicitement. À l'inverse, le test `k in d` sur un **dictionnaire** est en $O(1)$ en moyenne.

> `L = L + [x]` donne le même contenu que `L.append(x)`, mais en $O(n)$ (une nouvelle liste est créée, `L` y est recopiée) au lieu de $O(1)$ amorti.

Et pour comparer les structures (taille n, pour la concaténation, tailles n_1 et n_2) :

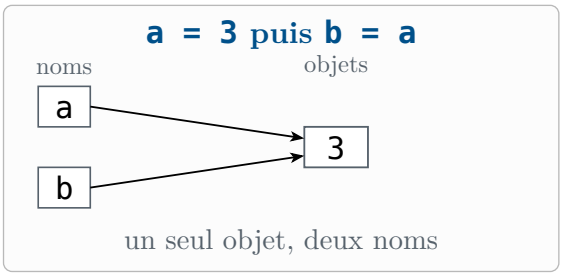
	accès au i-ème	ajout en fin	suppr. en fin	concaténation	recherche
tableau	$O(1)$	impossible*	impossible*	impossible*	$O(n)$
liste chaînée	$O(i)$	$O(n)$	$O(n)$	$O(n_1)$	$O(n)$
tableau dynamique	$O(1)$	$O(1)$ amorti	$O(1)$ amorti	$O(n_1+n_2)$	$O(n)$
▸ **liste Python aux concours**	_____ `L[i]`	_____ `L.append(x)`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

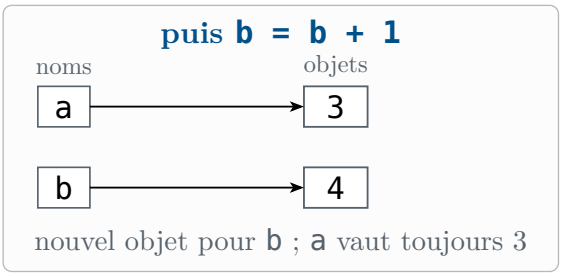
(*) il faut créer un nouveau tableau : $O(n)$. Aux concours, `append` (et souvent `pop()`) sont annoncés en $O(1)$, sans préciser « amorti ».

2.6 Rappel : références et copies

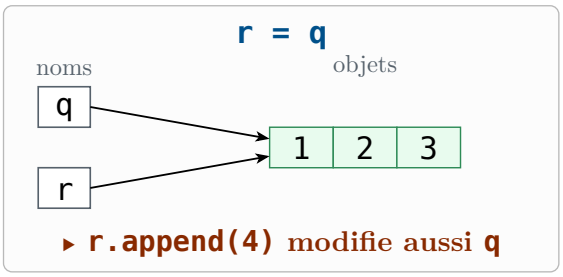
En Python, une variable est une « **étiquette** » posée sur un objet : elle contient une **référence** vers cet objet (concrètement, son adresse). L'affectation `b = a` ne copie jamais l'objet : elle copie la référence, et pose une seconde étiquette sur _____.

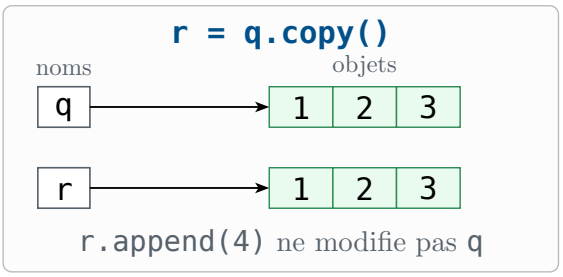
Pour un objet **immuable** (entier, flottant, chaîne, tuple de valeurs immuables), ce partage est sans conséquence : on ne peut pas modifier l'objet, et `b = b + 1` crée un **nouvel** objet sur lequel l'étiquette `b` est déplacée, `a` n'est pas affectée.



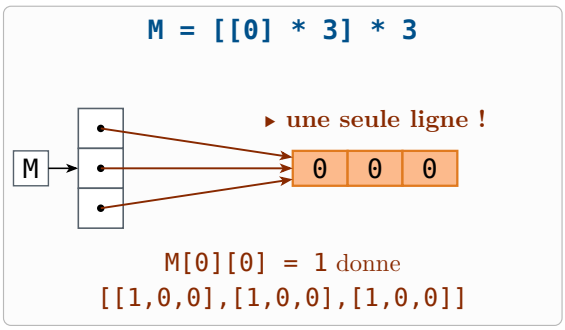


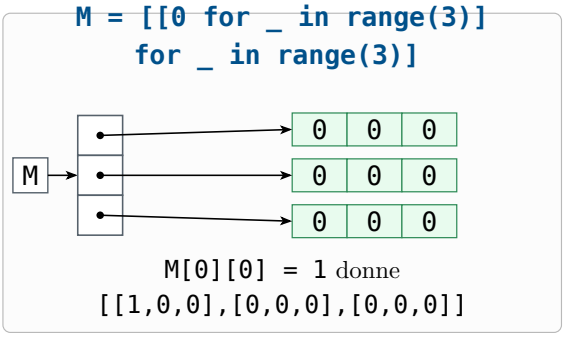
Une liste, elle, est **mutable**. Une variable `q` qui « contient » une liste contient en fait une référence vers l'objet liste, c'est-à-dire vers l'en-tête du tableau dynamique (taille, capacité et adresse du tableau de références). L'affectation `r = q` ne copie rien : `q` et `r` désignent le même tableau, et toute modification faite par `r` (`r.append(4)`, `r[0] = 5`…) se voit par `q`.





Le piège le plus classique concerne les listes de listes : `[x] * n` fabrique une liste de n cases qui contiennent toutes **la même référence**, celle de `x`.





Il en va de même pour `[[]] * m`, qui ne crée pas m listes vides mais m fois la même :

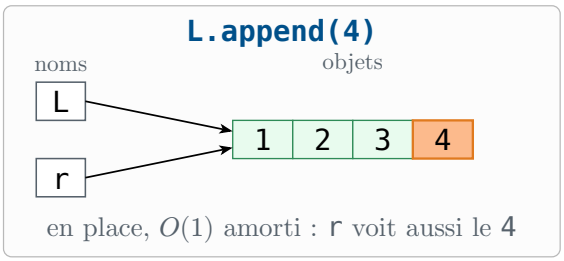
```
T = [[]] * 3
T[0].append(5)
T                    # [[5], [5], [5]]
```

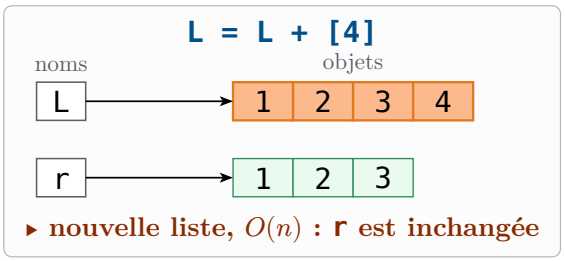
Attention. Pour construire une liste de listes (ou dès qu'il y a un doute), utiliser **toujours** une compréhension : `[[0 for _ in range(m)] for _ in range(n)]`, `[[] for _ in range(m)]`.

Question. 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)
```

Avec un second nom `r = L` posé sur la liste au départ, la différence entre les deux premières lignes se voit :



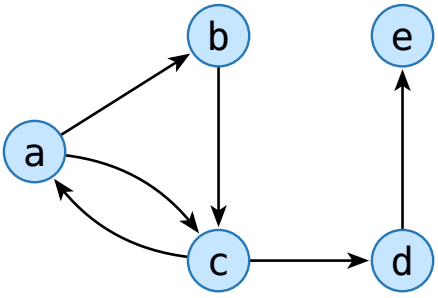


- `M.copy()` et `M[:]` sont des copies **superficielles** : pour une liste de listes, les lignes sont partagées. Pour copier un tableau 2D : `[ligne.copy() for ligne in M]` (ou `copy.deepcopy(M)`).
- Les `tuple` et les `str` sont **immuables** : on ne peut pas les modifier, seulement en construire de nouveaux. Le partage est alors sans danger, à condition qu'un tuple ne contienne pas lui-même d'objet mutable : avec `a = ([1, 2],)` et `b = a`, `b[0].append(3)` modifie aussi la liste contenue dans `a`.

3 Dictionnaires et hachage

3.1 Le problème

Pour représenter un graphe orienté dont les sommets sont désignés par des noms, il est naturel d'associer à chaque sommet la liste de ses voisins :



`voisins`

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

```
voisins = {"a": ["b", "c"], "b": ["c"], "c": ["a", "d"], "d": ["e"], "e": []}
voisins["c"]          # ['a', 'd']
"f" in voisins        # False
voisins["f"] = []     # ajout d'un sommet
```

Si les sommets sont des entiers numérotés de 0 à $n-1$, on peut utiliser un tableau de listes, mais ce n'est pas le cas en général.

De même, si l'on souhaite compter le nombre d'objets de chaque type (par exemple le nombre d'animaux de chaque type dans un zoo), la structure de liste n'est pas adaptée.

Définition. Un **dictionnaire** est une structure qui stocke des couples (clé, valeur), chaque clé apparaissant _____. Les opérations sur un dictionnaire sont :

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

Mathématiquement, si $\mathcal{C}$ est l'ensemble des clés possibles et $\mathcal{V}$ celui des valeurs, un dictionnaire est une application définie sur une partie **finie** $\mathcal{D} \subset \mathcal{C}$ (les clés présentes), que les opérations font évoluer :

$$\begin{aligned} d : \mathcal{D} &\to \mathcal{V} \\ k &\mapsto d(k) \end{aligned}$$

Pour le graphe, avec $S = \{a, b, c, d, e\}$ et A l'ensemble des arcs :

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

pour le décompte des animaux, avec Z l'ensemble des animaux du zoo et T l'ensemble des types :

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

Comment implémenter un dictionnaire sachant que l'ensemble $\mathcal{C}$ des clés possibles n'est pas un sous-ensemble de $\mathbb{N}$ et peut être très grand, et que l'ensemble $\mathcal{D}$ des clés présentes est inconnu (notamment son cardinal) et peut être modifié ?

3.1.1 Une implémentation naïve

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

- `rechercher_naif(D, cle)`, qui prend en argument un dictionnaire `D` et une clé possible `cle`, et renvoie la valeur associée à `cle` si la clé est présente, `None` sinon,
- `ajouter_naif(D, cle, valeur)`, qui prend en argument un dictionnaire `D`, une clé `cle` et une valeur `valeur`, et modifie `D` en place (sans rien renvoyer) : si `cle` est présente, la valeur associée 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` ?

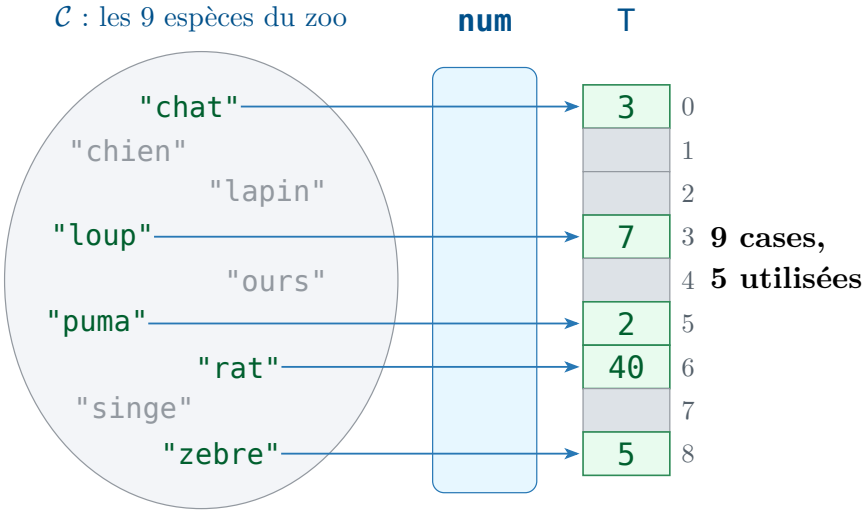
On veut faire mieux : recherche, ajout et suppression en $O(1)$!

3.2 Adressage direct

Si les clés sont des entiers de $[\![0, m-1]\!]$, ou si l'on dispose d'une **numérotation** des clés, c'est-à-dire d'une fonction injective

$$\begin{aligned} \text{num} : \mathcal{C} &\to [\![0, m-1]\!] \\ k &\mapsto \text{num}(k) \end{aligned}$$

c'est facile : on range la valeur associée à la clé k dans la case $\text{num}(k)$ d'un tableau de taille m.



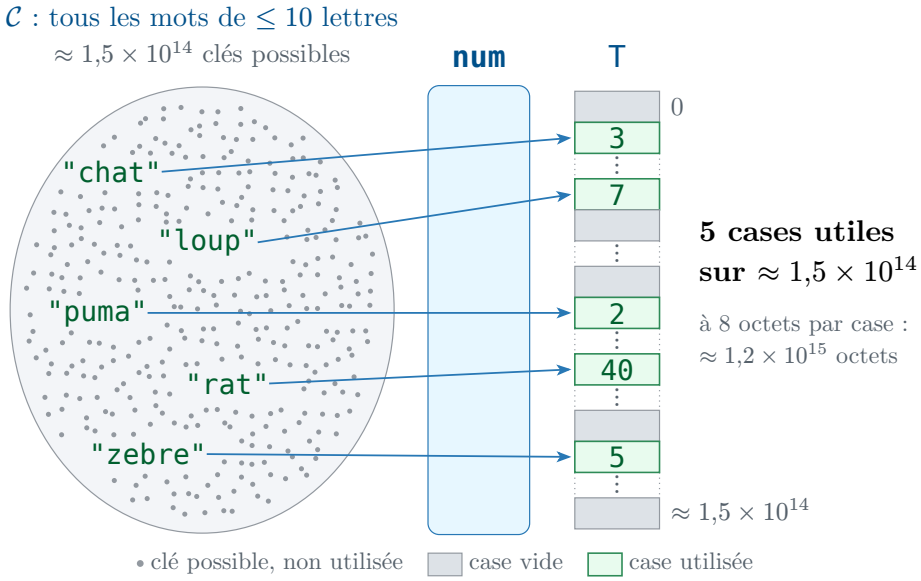
`num("chat") = 0`, `num("chien") = 1`, …, `num("zebre") = 8`

`"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

Toutes les opérations sont en $O(1)$ (si la fonction num est en $O(1)$) : un calcul de numéro, puis un accès au tableau. Mais la mémoire est en $O(m)$, où $m \geq |\mathcal{C}|$ (injectivité de num) est au moins le nombre de clés *possibles*, et non le nombre de clés *présentes* $|\mathcal{D}|$.

Cela est impraticable dès que l'ensemble des clés possibles est grand : pour des clés qui sont des mots d'au plus 10 lettres, il faudrait environ $1{,}5 \times 10^{14}$ cases, alors qu'on en utilisera beaucoup moins.



C'est là que se situe tout l'enjeu de l'implémentation des dictionnaires.

En pratique, si $\mathcal{C}$ est petit et connu, l'implémentation par adressage direct est possible, c'est-à-dire qu'on peut éviter l'usage d'un dictionnaire. C'est par exemple le cas dans les algorithmes suivants :

- le calcul d'occurrences de lettres si on sait par exemple qu'il n'y a que 26 lettres possibles (liste de longueur 26),
- le tri par comptage d'une liste de longueur n si on sait que tous les éléments sont des entiers compris entre 0 et $m-1$ avec $m = O(n)$ (liste de longueur m),
- pour les graphes, c'est l'idée d'une _____ : on utilise un tableau bidimensionnel plutôt qu'un dictionnaire de listes,
- dans le crible d'Ératosthène pour trouver des nombres premiers, on utilise un tableau de booléens plutôt qu'un dictionnaire.

3.3 Fonctions de hachage et collisions

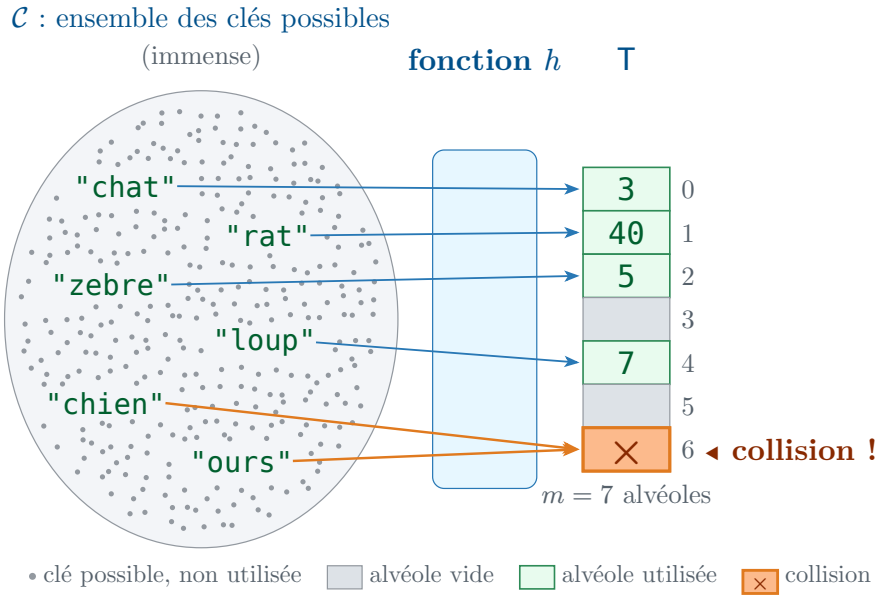
Définition. Soit $\mathcal{C}$ l'ensemble des clés possibles et m un entier. Une **fonction de hachage** est une fonction

$$h : \mathcal{C} \to [\![0, m-1]\!]$$
$$k \mapsto h(k)$$

Le couple (k, v) est rangé dans la case $h(k)$ d'un tableau T de taille m, appelée **alvéole**[1].

C'est le même principe que l'adressage direct, mais avec une fonction h qui n'est plus supposée injective : m peut alors être choisi de l'ordre du nombre de clés **utilisées**.

[1] Le programme ne fixe pas de vocabulaire. On trouve aussi *case* ou *compartiment*, en anglais *bucket* ou *slot*.



Souvent, cette fonction de hachage est la composée de deux fonctions : on calcule d'abord pour chaque clé un grand entier, son **haché**, indépendant de m (il peut ainsi être calculé une seule fois et réutilisé quand m change), que l'on réduit ensuite modulo m : $h(k) = \texttt{hache}(k)\%m$ (pour s'adapter à la taille du tableau). C'est ce que fait Python avec sa fonction `hash` (voir le § 3.8.1).

Dans la suite, on prend comme exemple de clés des mots, et comme fonction `hache` « jouet » la somme des rangs des lettres (a = 0, b = 1, ..., z = 25), avec $m = 7$:

```
def hache(mot):
    s = 0
    for c in mot:
        s = s + ord(c) - ord('a')
    return s

hache("chat") % 7    # (2 + 7 + 0 + 19) % 7 = 28 % 7 = 0 : h("chat") = 0
```

Collision. Deux clés $k \neq k'$ sont en **collision** si _____.

Le choix de m est donc un **compromis** :
- si m est trop petit devant le nombre n de clés présentes, les collisions sont nombreuses,
- s'il est trop grand, on gaspille de la mémoire (dans le pire des cas on retombe sur le défaut de l'adressage direct).

Facteur de charge. On appelle **facteur de charge** le rapport

$$\alpha = \frac{n}{m}$$

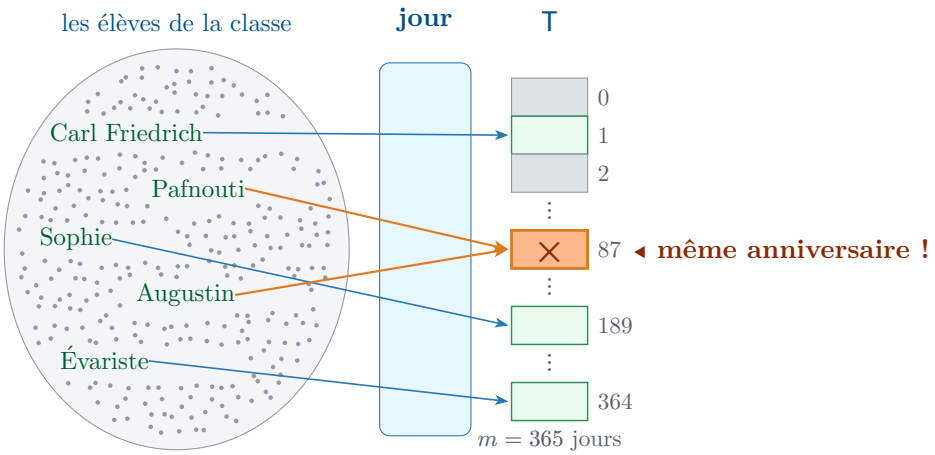
où $n = |\mathcal{D}|$ est le nombre de clés présentes et m le nombre d'alvéoles.

Ce rapport permet de mesurer ce compromis :
- si α est grand, il y a beaucoup de collisions, par exemple, si $\alpha > 1$, par le _____ il y a forcément au moins une collision,
- si α est petit, on gaspille de la mémoire : on a en effet $\frac{n}{\alpha}$ alvéoles pour n clés.

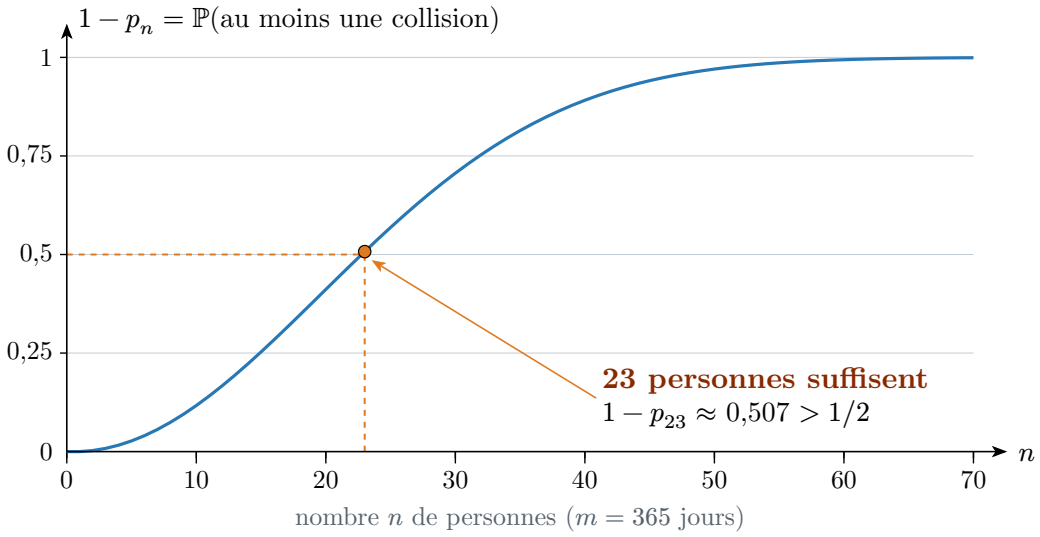
On cherchera dans la suite à le garder borné (et même < 1 en adressage ouvert).

3.3.1 Le paradoxe des anniversaires

En fait, même avec α petit, les collisions arrivent bien plus tôt qu'on ne pourrait naïvement le penser : c'est le fameux paradoxe des anniversaires.



Dès qu'on a 23 élèves, il y a au moins une chance sur deux que deux élèves aient le même anniversaire, pour 42 élèves, la probabilité est d'environ 0,91.



Supposons que les n clés sont envoyées uniformément et indépendamment dans les m alvéoles.

Question. 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 $m \sim \alpha n$.
- 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.

Conclusion : on ne peut pas éviter les collisions, il faut les gérer. Il y a deux grandes stratégies : le **chaînage** et l'**adressage ouvert**.

3.3.2 Coûts moyens

Dans les deux cas, on va calculer le coût d'une **recherche** de la clé k. Plus précisément, si l'ensemble des clés présentes est $\mathcal{D} = \{k_1, ..., k_n\}$ (insérées dans cet ordre) et si $k \in \mathcal{C}$ est une clé potentielle, on s'intéresse au coût :

- d'une recherche **fructueuse**, où $k \in \{k_1, ..., k_n\}$ (choisie uniformément parmi les n clés présentes),
- d'une recherche **infructueuse**, où $k \notin \{k_1, ..., k_n\}$.

L'ajout et la suppression se ramènent à une recherche, suivie d'un travail en $O(1)$:

- **ajouter** (k, v) : on recherche d'abord k. Si la recherche est fructueuse, on remplace la valeur associée à k, si elle est infructueuse, on range le couple (k, v) à une place libre. Ajouter une clé nouvelle coûte donc une recherche infructueuse (il faut s'assurer que la clé est absente), et modifier la valeur d'une clé présente une recherche fructueuse,
- **supprimer** k : on recherche k (recherche fructueuse si k est présente), puis on retire le couple trouvé.

Il suffit donc d'étudier le coût des deux types de recherche.

Ce coût dépend de la fonction de hachage, de la répartition des clés $k_1, ..., k_n$ et de la clé recherchée k.

On modélise ces dépendances en considérant X le coût comme une _____, et on s'intéresse au **coût moyen**, $\mathbb{E}[X]$, et ce en fonction du facteur de charge α.

On notera X_{I} le coût d'une recherche infructueuse et X_{F} celui d'une recherche fructueuse.

On fera dans la suite l'hypothèse suivante, idéalisée (elle n'est pas vérifiée en pratique, voir le § 3.7 pour ce qu'on peut garantir en choisissant bien la fonction de hachage) :

Hachage uniforme. Les alvéoles $h(k_1), ..., h(k_n)$ des clés présentes sont des variables aléatoires _____ et _____ sur $[\![0, m-1]\!]$, pour une clé k absente, $h(k)$ est aussi uniforme et indépendante des précédentes.

3.4 Résolution des collisions par chaînage

3.4.1 Principe

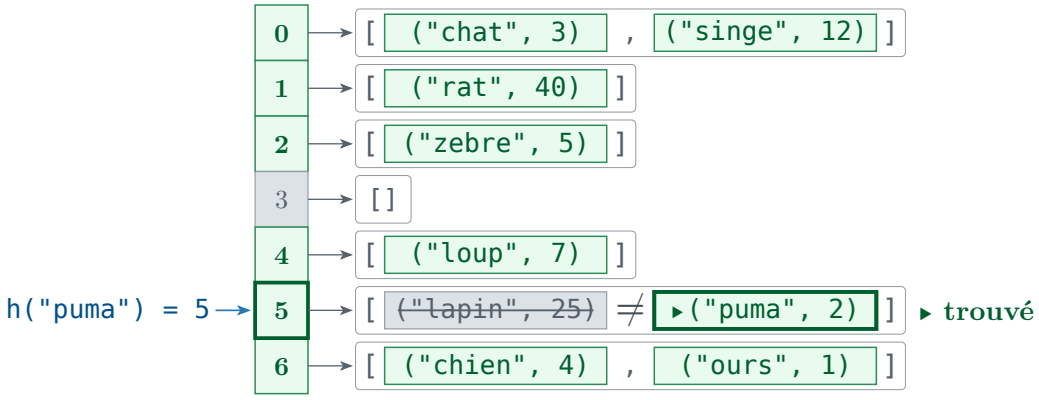
Chaque alvéole $T[i]$ contient la **liste** des couples (k, v) tels que $h(k) = i$: c'est un petit dictionnaire naïf, comme celui du § 3.1.1, mais qui ne contient que les clés « tombées » dans cette alvéole.

- **Rechercher** k : on calcule $i = h(k)$, puis on parcourt la liste $T[i]$ en comparant chaque clé à k.
- **Ajouter** (k, v) : on recherche k dans $T[i]$, si elle y est, on remplace sa valeur, sinon on ajoute le couple à la fin de $T[i]$.
- **Supprimer** k : on recherche k dans $T[i]$ et on retire le couple de cette liste.

3.4.2 Exemple

Avec la fonction jouet et $m = 7$, on insère successivement les couples `("chat", 3)`, `("zebre", 5)`, `("rat", 40)`, `("lapin", 25)`, `("puma", 2)`, `("loup", 7)`, `("singe", 12)`, `("chien", 4)` et `("ours", 1)` (les cinq premiers dans le même ordre que pour l'adressage ouvert, § 3.5.2). Les quatre premiers tombent dans des alvéoles vides, `"puma"`, `"singe"` et `"ours"` entrent en collision (avec `"lapin"`, `"chat"` et `"chien"`) : chacun est ajouté à la fin de la liste de son alvéole.

- Recherche fructueuse de `"puma"` : `hache`(puma) $= 15 + 20 + 12 + 0 = 47$ et $47 \% 7 = 5$, on parcourt l'alvéole 5, on compare `"puma"` à `"lapin"`, puis à `"puma"` : trouvé.
- Recherche infructueuse de `"girafe"` : `hache`(girafe) $= 40$ et $40 \% 7 = 5$, on compare `"girafe"` à `"lapin"` puis à `"puma"`, et la liste est finie : absente.



alvéole = liste de couples (clé, valeur)

3.4.3 En Python

Chaque alvéole étant un dictionnaire naïf, la recherche et l'ajout réutilisent directement les fonctions du § 3.1.1 :

```
def creer_table(m):
    return [[] for _ in range(m)]      # et surtout pas [[]] * 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
```

3.4.4 Analyse

L'efficacité de cette méthode repose sur l'idée suivante : il y a forcément des collisions, mais la taille de chaque alvéole (c'est-à-dire le nombre d'éléments qui entrent en collision dans la même alvéole) est en moyenne faible si le facteur de charge est petit.

Dans l'exemple du paradoxe des anniversaires, avec 42 élèves, on a vu qu'il y a une probabilité de 0,91 qu'il y ait au moins une collision, mais les 42 élèves occupent en moyenne environ 39,7 jours distincts, et le nombre moyen d'élèves par jour d'anniversaire comptant au moins un élève n'est que d'environ 1,06.

Le coût d'une recherche est le nombre de clés de l'alvéole **examinées** (comparées à k), le calcul de h étant supposé en $O(1)$. On suppose de plus qu'aucune clé n'a été supprimée.

Dans le pire des cas (une alvéole contient toutes les clés), le coût est $O(n)$, mais on va calculer le coût moyen sous l'hypothèse de **hachage uniforme** (ici on peut théoriquement avoir $\alpha > 1$).

Question. (Recherche infructueuse.) Soit $k \notin \{k_1, ..., k_n\}$.

Calculer $\mathbb{E}[X_I]$, où X_I est le nombre de clés examinées lors de sa recherche.

Question. (Recherche fructueuse.) Soit $k \in \{k_1, ..., k_n\}$.

Calculer $\mathbb{E}[X_F]$, où X_F est le nombre de clés examinées.

Bilan. Sous l'hypothèse de hachage uniforme, une recherche (fructueuse ou non), un ajout ou une suppression coûtent en moyenne _____. Si l'on maintient α borné, toutes les opérations sont en $\boldsymbol{O(1)}$ **en moyenne**.

3.5 Résolution des collisions par adressage ouvert

3.5.1 Principe

Autre stratégie, sans listes : chaque case du tableau contient **au plus un** couple. Chaque clé k a une **suite de sondage** $s(k,0), s(k,1), ..., s(k,m-1)$, permutation de $[\![0, m-1]\!]$: ce sont les cases que l'on essaie, dans l'ordre. L'exemple le plus simple est le **sondage linéaire** : $s(k,i) = (h(k)+i)\%m$, c'est-à-dire qu'on essaie la case $h(k)$, puis les suivantes.

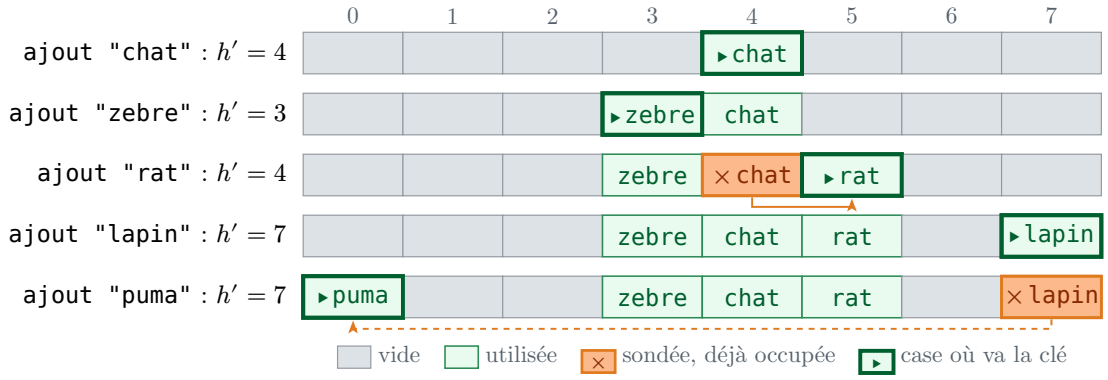
- **Rechercher** k : on sonde $s(k,0), s(k,1), ...$ jusqu'à trouver la clé, ou une case vide (la clé est alors absente).
- **Ajouter** (k,v) : on recherche d'abord k. Si elle est présente, on remplace sa valeur, sinon, on range le couple dans la première case _____ rencontrée pendant la recherche (vide, ou marquée « supprimé », voir ci-dessous).
- **Supprimer** k : on ne peut pas simplement vider la case, sinon on couperait les suites de sondage des clés rangées après (une recherche s'arrêterait trop tôt). On y place un marqueur « supprimé », que la recherche saute et qu'un ajout peut réutiliser.

Il faut donc $\alpha = n/m < 1$, et même (nombre de clés + nombre de marqueurs) $< m$: une recherche infructueuse doit rencontrer une case vide pour s'arrêter.

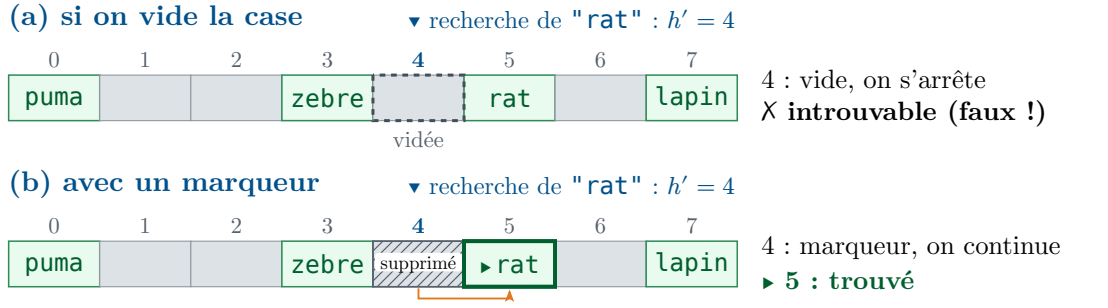
3.5.2 Exemple

Avec $m = 8$ et le sondage linéaire, où l'on note $h'(k) =$ `hache`$(k)\%8$, on insère successivement les couples `("chat", 3)`, `("zebre", 5)`, `("rat", 40)`, `("lapin", 25)` et `("puma", 2)`. Les deux premiers tombent dans des cases vides, `"rat"` entre en collision avec `"chat"`, `"lapin"` tombe dans une case vide, puis `"puma"` entre en collision avec `"lapin"` : un couple en collision est rangé dans la première case libre qui suit (circulairement).

- Recherche fructueuse de `"puma"` : $h'(\text{puma}) = 47\%8 = 7$, la case 7 contient `"lapin"`, on essaie la case $(7+1)\%8 = 0$: trouvé.
- Recherche infructueuse de `"loup"` : $h'(\text{loup}) = 60\%8 = 4$, les cases 4 (`"chat"`) et 5 (`"rat"`) ne contiennent pas `"loup"`, la case 6 est vide : absente.



Suppression de `"chat"` dans cet exemple : si l'on vidait la case 4, la recherche de `"rat"` (rangé en 5) s'arrêterait à tort sur cette case vide.



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

3.5.3 En Python

Sans suppression (donc sans marqueur), avec le sondage linéaire et `None` pour une case vide :

```
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):   # table non pleine
    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)
```

3.5.4 Analyse

On suppose dans toute cette analyse qu'**aucune clé n'a été supprimée** : la table ne contient que n clés et des cases vides, sans marqueur. Comme pour le chaînage, ajouter une clé nouvelle coûte alors une

recherche infructueuse, et rechercher une clé présente parcourt exactement les cases sondées lors de son insertion. Le coût d'une recherche est le nombre de cases **sondées**.

Sondage uniforme. La suite de sondage de chaque clé est une permutation de $[\![0, m-1]\!]$ choisie uniformément, indépendamment des autres clés.

Question. (Recherche infructueuse.) Soit $k \notin \{k_1, ..., k_n\}$, avec $n < m$.

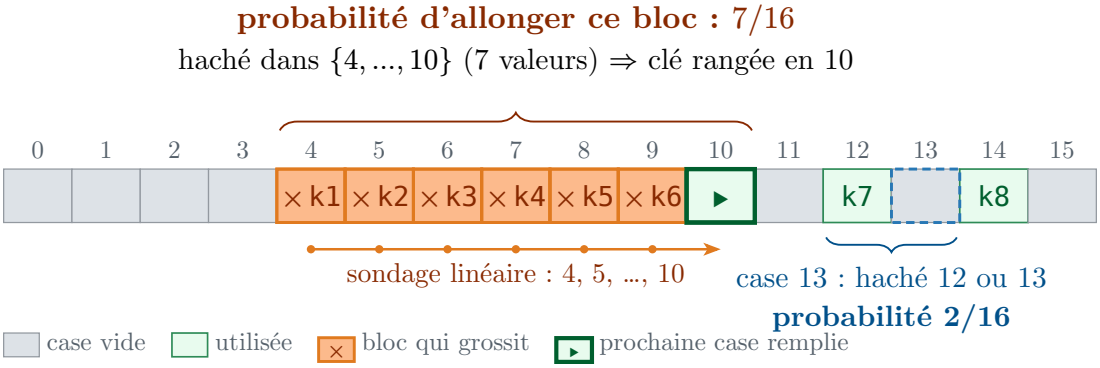
Calculer $\mathbb{E}[X_{\mathrm{I}}]$, où X_{I} est le nombre de cases sondées lors de sa recherche, puis montrer que $\mathbb{E}[X_{\mathrm{I}}] \leq 1/(1-\alpha)$.

Indication : on rappelle que $\mathbb{E}[X_{\mathrm{I}}] = \sum_{i \geq 1} \mathbb{P}(X_{\mathrm{I}} \geq i)$.

Question. (Recherche fructueuse.) Soit $k \in \{k_1, ..., k_n\}$, 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 $\mathbb{E}[X_{\mathrm{F}}] \leq \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$.

L'hypothèse de sondage uniforme est idéalisée. Le sondage linéaire de l'exemple (on essaie la case $h(k)$, puis $h(k)+1$, $h(k)+2$… modulo m) fait nettement moins bien : les cases occupées forment des **blocs** contigus, et une clé qui tombe n'importe où dans un bloc allonge ce bloc. Plus un bloc est long, plus il a de chances de grossir : c'est le **regroupement primaire**.



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

Knuth a montré que, pour le sondage linéaire on a $\mathbb{E}[X_{\mathrm{I}}] = \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$ et $\mathbb{E}[X_{\mathrm{F}}] = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$.

3.5.5 Chaînage ou adressage ouvert ?

Aucune des deux stratégies n'est meilleure que l'autre : si α est maintenu borné (et, en adressage ouvert, loin de 1), les deux donnent des opérations en $O(1)$ en moyenne, et les différences ne portent que sur les constantes, la mémoire et les usages.

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

C'est pourquoi des langages différents ont fait des choix différents :

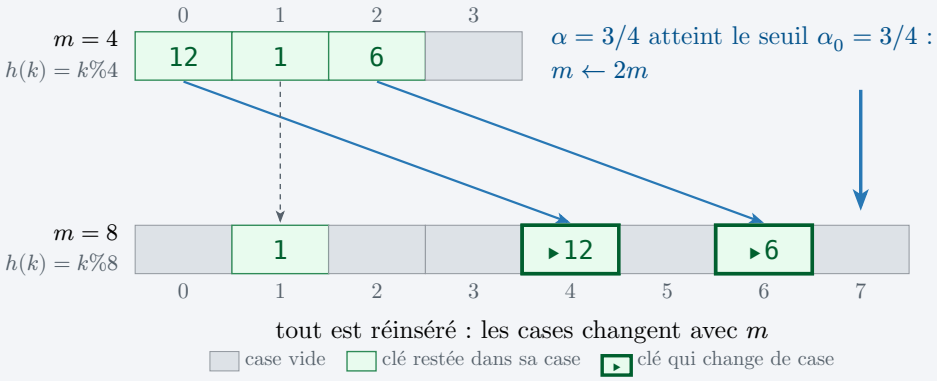
Langage	**Structure**	**Stratégie**
Java	`HashMap`	chaînage
C++	`std::unordered_map`	chaînage
Python	`dict`	adressage ouvert
Rust	`HashMap`	adressage ouvert
Julia	`Dict`	adressage ouvert (sondage linéaire)
Lua	tables	mixte[2]

[2]La partie « dictionnaire » d'une table Lua est un tableau dont chaque case contient un couple et l'indice de la case suivante de sa liste : les collisions sont chaînées comme dans le chaînage, mais les listes sont rangées dans les cases libres du tableau lui-même, comme en adressage ouvert (*chained scatter table*). Une table Lua a en plus une partie « tableau » pour les clés entières $1, 2, \ldots, n$.

3.6 Redimensionnement (HP)

HORS PROGRAMME · redimensionnement

Dans les deux stratégies, le coût moyen ne reste constant que si $\alpha = n/m$ reste borné (et, en adressage ouvert, nettement inférieur à 1). Comme n augmente au fil des ajouts, on **agrandit la table** dès que α atteint un seuil α_0 (par exemple 1 pour un chaînage, 2/3 pour les dictionnaires de Python, 3/4 dans le schéma ci-dessous) : on crée une table de taille $2m$ et on y **réinsère** tous les couples, car les alvéoles $\texttt{hache}(k)\%m$ changent avec m.



Un redimensionnement coûte $O(n)$ en moyenne (on réinsère les n couples). Mais comme la taille **double** à chaque fois, c'est exactement le calcul des tableaux dynamiques avec $q = 2$: N ajouts successifs coûtent $O(N)$ au total, l'ajout reste en $O(1)$ amorti. En adressage ouvert, la reconstruction fait au passage disparaître les marqueurs « supprimé ».

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

HORS PROGRAMME · choix d'une fonction de hachage

Une fonction de hachage ne peut pas être injective, mais on veut qu'elle se comporte « comme si » : que les clés réellement utilisées se répartissent le plus uniformément possible, ce qui justifie l'hypothèse de hachage uniforme. Elle doit être :

1. **déterministe** : au cours d'une même exécution, la même clé a toujours le même haché (sinon on ne retrouve plus rien),
2. **rapide** à calculer (en $O(1)$, ou en temps proportionnel à la taille de la clé),
3. bien **répartie** : les clés réellement utilisées doivent se répartir uniformément dans les alvéoles, même si elles se ressemblent (`"chat1"`, `"chat2"`…).

Les deux fonctions naïves suivantes sont de mauvais exemples :

- le rang de la première lettre : au plus 26 valeurs, quel que soit m, et très mal réparties (beaucoup de mots en « c », « p », « s », peu en « w »),
- la somme des rangs des lettres (notre fonction `hache`) : les valeurs restent petites (au plus $25 \times$ longueur, donc quelques centaines), si bien qu'avec m grand la plupart des alvéoles restent vides, et deux anagrammes (`chien`, `niche`, `chine`) ont toujours le même haché.

3.7.1 Clés entières

- **Méthode de la division** : $h(k) = k\%m$. Il faut éviter que m partage des « régularités » avec les clés : si $m = 2^p$, $h(k)$ ne dépend que des p derniers bits de k, si toutes les clés sont des multiples de 10 et $m = 10$, tout tombe dans l'alvéole 0. On choisit plutôt m premier, loin d'une puissance de 2.
- **Méthode de la multiplication** : $h(k) = \lfloor m\{k\theta\} \rfloor$, où $\{x\}$ désigne la partie fractionnaire et $\theta \in]0,1[$ un irrationnel, par exemple $\theta = \left(\sqrt{5}-1\right)/2$ (Knuth). La valeur de m importe peu. La répartition est liée à l'équirépartition de la suite $(\{k\theta\})_k$.

3.7.2 Clés chaînes de caractères

On voit le mot comme l'écriture d'un entier en base b :

$$\texttt{hache\_poly}(c_0c_1...c_{\ell-1}) = \left(c_0 b^{\ell-1} + c_1 b^{\ell-2} + ... + c_{\ell-1}\right)\%M$$

où c_i est le code de la lettre d'indice i, b une base (31 par exemple) et M un grand entier. Chaque lettre influe sur le résultat en fonction de sa **position** : les anagrammes ne sont plus systématiquement en collision.

On le calcule en $O(\ell)$ opérations arithmétiques (ℓ : longueur du mot) par la méthode de Horner, avec $c_i =$ `ord(c)` :

```
def hache_poly(mot, b=31, M=2**61 - 1):
    s = 0
    for c in mot:
        s = (s * b + ord(c)) % M
    return s
```

3.7.3 Hachage universel

Quelle que soit la fonction h fixée, un adversaire qui la connaît peut choisir ses clés : si $|\mathcal{C}| \geq nm$, une alvéole reçoit au moins n clés de $\mathcal{C}$ (principe des tiroirs), et il lui suffit d'utiliser celles-là pour que le coût moyen des opérations devienne de l'ordre de n, au lieu de $O(1)$. La parade consiste à **tirer h au hasard** au début de l'exécution, dans une famille bien choisie.

Famille universelle. Une famille $\mathcal{H}$ de fonctions $\mathcal{C} \to [\![0, m-1]\!]$ est **universelle** si pour toutes clés $k \neq k'$, $\mathbb{P}_{h\in\mathcal{H}}(h(k) = h(k')) \leq 1/m$ lorsque h est choisie uniformément dans $\mathcal{H}$.

Dans l'analyse du chaînage, on n'a utilisé que $\mathbb{P}\big(h(k_j) = h(k)\big) \leq 1/m$: avec une famille universelle, le résultat $\mathbb{E}[X_{\mathrm{I}}] \leq \alpha$ vaut donc pour **n'importe quel** ensemble de clés, la probabilité portant sur le choix de h et non plus sur les clés. On admet que si les clés sont des entiers de $[\![0, p-1]\!]$, avec p premier, la famille des

$$h_{a,b} : k \mapsto ((ak + b)\%p)\%m, \quad a \in [\![1, p-1]\!], b \in [\![0, p-1]\!]$$

est universelle (Carter et Wegman, 1979).

3.8 Les dictionnaires de Python

3.8.1 La fonction `hash`

La fonction `hash` de Python joue le rôle de notre fonction `hache` : elle calcule le haché de tout objet hachable, que le dictionnaire réduit ensuite à la taille de sa table.

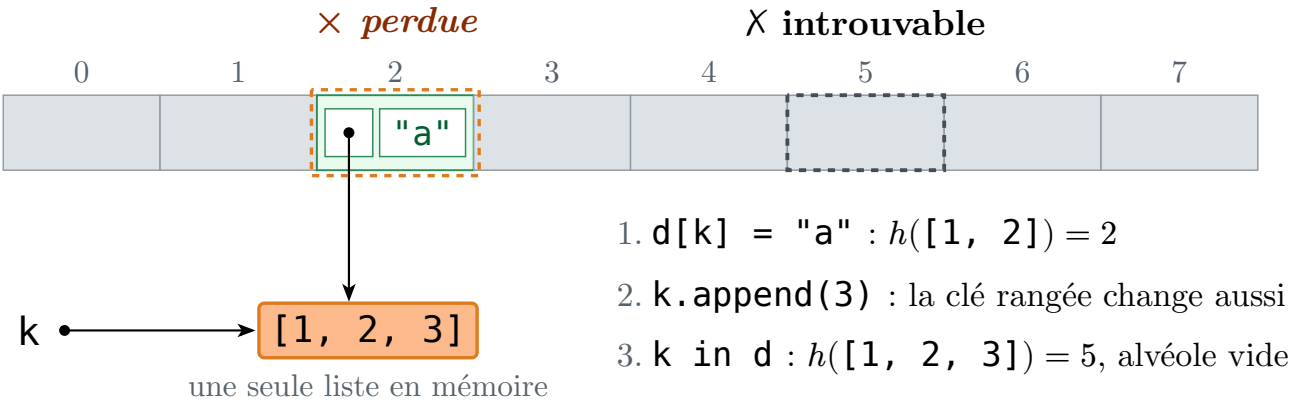
```
>>> hash(12), hash(2**61 + 42), hash(-3)
(12, 43, -3)
>>> hash("chat")        # change à chaque lancement de Python !
3192665300061180367
>>> hash((1, "a"))      # tuple de hachables (varie : contient une chaîne)
6478549121272982084
>>> hash([1, 2])
TypeError: unhashable type: 'list'
```

- Les entiers « raisonnables » sont leur propre haché : sur une machine 64 bits, `hash(n)` vaut $n\%(2^{61}-1)$ pour $n \geq 0$ et $-$`hash`$(-n)$ pour $n < 0$, sauf `hash(-1) == -2` (la valeur -1 est réservée, dans le code C de Python, pour signaler une erreur).
- Pour les chaînes, Python utilise une fonction *SipHash*, paramétrée par une clé secrète tirée au hasard à chaque lancement de l'interpréteur (voir le § 3.7.3) : c'est pourquoi `hash("chat")` change d'une exécution à l'autre.

3.8.2 Quelles clés sont autorisées ?

Règle. Une clé de dictionnaire doit être _____, ce qui en pratique signifie _____ : `int`, `float`, `bool`, `str`, `tuple` (dont les éléments sont eux-mêmes admissibles). Les `list`, `dict` et `set` sont interdits.

La raison est la suivante. Imaginons qu'on puisse utiliser la liste `k = [1, 2]` comme clé : le couple est rangé dans l'alvéole $h([1,2])$. Si l'on exécute ensuite `k.append(3)`, la clé stockée dans la table (qui n'est qu'une **référence** vers la liste `k`) vaut désormais `[1, 2, 3]`. Une recherche de `k` irait regarder l'alvéole $h([1,2,3])$, a priori différente : la clé est « perdue » dans la table, alors qu'elle y est toujours. Et rechercher `[1, 2]` échoue aussi : on regarde la bonne alvéole, mais la clé qui s'y trouve ne vaut plus `[1, 2]`. **Le haché d'une clé ne doit jamais changer.**



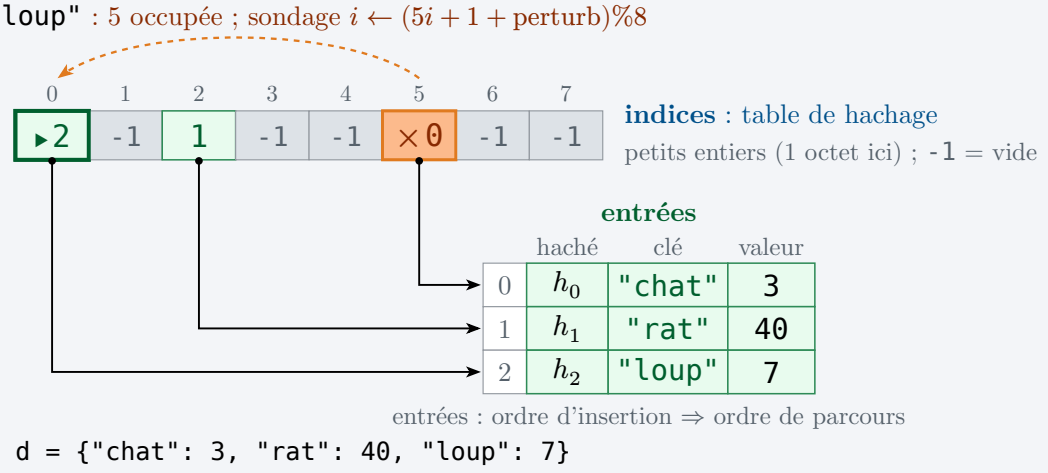
- Le dictionnaire exige aussi : `a == b` ⇒ `hash(a) == hash(b)`. Ainsi `1`, `1.0` et `True` sont égaux et ont le même haché : ce sont **la même clé**. `{1: "a", 1.0: "b", True: "c"}` vaut `{1: "c"}`.
- Les fonctions sont hachables : leur haché dépend de leur **identité** (l'adresse de l'objet fonction : c'est elle qu'on voit quand, erreur classique en TP, on écrit `print(f)` au lieu de `print(f(x))` et qu'on obtient `<function f at 0x7f3a2c1b8e50>`), et non de ce qu'elles calculent. Deux fonctions ne sont la même clé que si elles sont le même objet. On s'en sert rarement comme clés, en revanche, des fonctions peuvent très bien être les **valeurs** d'un dictionnaire, par exemple `{"carre": carre, "cube": cube}`.

3.8.3 L'implémentation de `dict` (HP)

HORS PROGRAMME · implémentation de Python

Python utilise l'**adressage ouvert**, avec deux tableaux :

- une table des **indices**, de taille $m = 2^p$ (au moins 8), qui est la vraie table de hachage : chaque case contient « vide », « supprimé », ou l'indice d'une entrée,
- un tableau des **entrées** (haché, clé, valeur), rangées dans l'**ordre d'insertion** : c'est pourquoi, depuis Python 3.7, le parcours d'un dictionnaire suit l'ordre d'insertion.




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

- **Sondage** : avec $H = \texttt{hash}(k)$, la première case sondée est $H\%2^p$ (les p bits de poids faible du haché, calcul très rapide, pour des clés entières consécutives, aucune collision). Ensuite, avec `perturb` initialisé à H (vu comme un entier positif sur 64 bits), on répète : `perturb` $\leftarrow$ `perturb // 32` (on retire ses 5 derniers chiffres binaires), puis $i \leftarrow (5i + 1 + \texttt{perturb})\%2^p$. Les bits de poids fort du haché interviennent ainsi peu à peu, ce qui compense le fait que m soit une puissance de 2, quand `perturb` est devenu nul, la récurrence $i \leftarrow (5i + 1)\%2^p$ parcourt toutes les cases.
- **Facteur de charge** : au plus 2/3, marqueurs « supprimé » compris (les tables de taille 8, 16, 32… acceptent 5, 10, 21… entrées). Au-delà, la table est **redimensionnée** : reconstruite avec la plus petite taille $2^p \geq 3n$, soit le double de la taille précédente s'il n'y a pas eu de suppression. Dans le modèle du sondage uniforme, une recherche infructueuse coûterait alors au plus $1/(1-2/3) = 3$ sondages en moyenne, une recherche fructueuse au plus $\frac{3}{2}\ln 3 \approx 1{,}65$: c'est un ordre de grandeur, pas une garantie, puisque le sondage perturbé de Python n'est pas un sondage uniforme.
- **Suppression** : la case correspondante de la table des indices reçoit le marqueur « supprimé ». En pratique, cette table contient des entiers : -1 pour « vide », -2 pour « supprimé », un entier ≥ 0 pour l'indice d'une entrée. L'entrée elle-même est vidée (clé et valeur effacées) mais reste en place dans le tableau des entrées, jusqu'au prochain redimensionnement.

3.9 Utiliser les dictionnaires

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

Ces coûts constants sont des coûts **en moyenne** (hypothèse de hachage uniforme) et, pour l'ajout, **amortis** (redimensionnement).

Pour des clés `str` ou `tuple`, il faut ajouter le coût du hachage de la clé, proportionnel à sa taille, et celui de la comparaison de la clé trouvée avec la clé recherchée.

Attention. Accéder à une clé absente, c'est-à-dire écrire `d[k]` où `k` n'est pas une clé présente, lève `KeyError`. Il faut traiter à part le cas d'une clé absente, le code est donc presque toujours de la forme

```
if k in d:
    ...        # k est présente : on peut utiliser d[k]
else:
    ...        # k est absente
```

Le type `set` (ensemble) est un dictionnaire sans valeurs : `s = set()`, `s.add(x)`, `x in s` en $O(1)$ en moyenne. Il est hors programme (et parfois interdit) : en pratique on peut le remplacer par un dictionnaire dont les valeurs sont `True`.

Des exercices d'application sur les listes et les dictionnaires (programmes usuels, à traiter en travail personnel) sont proposés dans un document séparé.