

Algorithmique et structures de données

Cyril Nicaud

`cyril.nicaud@univ-eiffel.fr`

L2 – Université Gustave-Eiffel

Premier semestre 2023

tableaux dynamiques, tables de hachage

I. Représentation ordonnées de données

Requêtes sur une structure de données statique

- ▶ Une **structure de données** est une façon d'organiser la mémoire pour y représenter un ensemble de données
- ▶ Une structure de donnée est **statique** quand elle n'est pas modifiée après son initialisation
- ▶ Une **requête typique** sur ce genre de structure consiste à parcourir ses éléments pour : chercher si x est dedans, trouver le minimum, calculer la moyenne, ...
- ▶ Si on considère que ses éléments sont de plus **ordonnés**, on peut également demander accès au i -ème élément dans l'ordre : c'est le cas pour les tableaux et les listes chaînées en C, les listes en Python, ...

On va se focaliser pour le moment sur les deux requêtes suivantes :

- ▶ `contains(S, x)` : Est-ce que la valeur x est dans S ?
- ▶ `value_at(S, i)` : Quel est la valeur en position i dans S ?

Utilisation d'un tableau en C

On peut utiliser un **tableau** en C avec les *n* données :

```
typedef struct{
    int n;
    int *elements;
} int_array;

int contains(int_array S, int x) {
    for(int i=0; i<S.n; i++)
        if (S.elements[i] == x) return 1;
    return 0;
}

int value_at(int_array S, int i) {
    return S.elements[i];
}
```

Utilisation d'un tableau en C

On peut utiliser un **tableau** en C avec les n données :

```
typedef struct{
    int n;
    int *elements;
} int_array;

int contains(int_array S, int x) {
    for(int i=0; i<S.n; i++)
        if (S.elements[i] == x) return 1;
    return 0;
}

int value_at(int_array S, int i) {
    return S.elements[i];
}
```

Complexités en fonction de $n = |S|$:

- ▷ $\text{contains}(S, x) : \Theta(n)$, dans le pire cas $x \notin S$
- ▷ $\text{value_at}(S, i) : \Theta(1)$

Utilisation d'une liste chaînée en C

De façon similaire avec des listes (simplement) chaînées

```
int contains(list L, int x) {  
    while (L) {  
        if (L->value == x) return 1;  
        L = L->next;  
    }  
    return 0;  
}
```

```
int value_at(list L, int i) {  
    for(int j=0; j<i; j++)  
        L = L-> next;  
    return L->value;  
}
```

Utilisation d'une liste chaînée en C

De façon similaire avec des listes (simplement) chaînées

```
int contains(list L, int x) {  
    while (L) {  
        if (L->value == x) return 1;  
        L = L->next;  
    }  
    return 0;  
}
```

```
int value_at(list L, int i) {  
    for(int j=0; j<i; j++)  
        L = L-> next;  
    return L->value;  
}
```

Complexités en fonction de $n = |S|$:

- ▷ $\text{contains}(S, x) : \Theta(n)$, dans le pire cas $x \notin S$
- ▷ $\text{value_at}(S, i) : \Theta(n)$, on peut dire $\Theta(i)$ si on exprime en fonction de i

Bilan sur les structures statistiques

	contains	value_at
tableaux	$\Theta(n)$	$\Theta(1)$
listes	$\Theta(n)$	$\Theta(n)$

- ▶ Les tableaux sont **théoriquement** plus adaptés dans ce cas
- ▶ En **pratique** également (même si on n'utilise pas `value_at`)

Bilan sur les structures statistiques

	contains	value_at
tableaux	$\Theta(n)$	$\Theta(1)$
listes	$\Theta(n)$	$\Theta(n)$

- ▶ Les tableaux sont **théoriquement** plus adaptés dans ce cas
- ▶ En **pratique** également (même si on n'utilise pas `value_at`)

Et si l'ensemble des données évolue au cours du temps ?

Nouveau cahier des charges

On veut une structure ordonnée S avec les requêtes suivantes :

- ▶ $\text{contains}(S, x)$: Est-ce que la valeur x est dans S ?
- ▶ $\text{value_at}(S, i)$: Quel est la valeur en position i dans S ?
- ▶ $\text{append}(S, x)$: Ajoute x à la fin de S
- ▶ $\text{pop}(S)$: Supprime le dernier élément de S

C'est une structure de donnée **dynamique** : elle évolue au cours du temps avec les ajouts/suppressions

Regardons pour les listes et les tableaux

Utilisation d'une liste chaînée en C

```
list append(list L, int x) {  
    if (!L) return new_one_element(x);  
    list N = L;  
    while (N->next) N = N->next;  
    N->next = new_one_element(x);  
    return L;  
}  
  
list pop(list L) {  
    if (!L->next) {free(L); return NULL;}  
    list resultat = L;  
    while (L->next->next) L = L->next;  
    free(L->next); L->next = NULL;  
    return resultat;  
}
```

Utilisation d'une liste chaînée en C

```
list append(list L, int x) {  
    if (!L) return new_one_element(x);  
    list N = L;  
    while (N->next) N = N->next;  
    N->next = new_one_element(x);  
    return L;  
}  
  
list pop(list L) {  
    if (!L->next) {free(L); return NULL;}  
    list resultat = L;  
    while (L->next->next) L = L->next;  
    free(L->next); L->next = NULL;  
    return resultat;  
}
```

- ▶ Il faut parcourir les listes, donc complexités en $\Theta(n)$
- ▶ Si on avait demandé ajout et suppression au début : $\Theta(1)$

Optimisation des listes chaînées pour nos requêtes

Une première idée :

- ▶ Si on garde un **pointeur sur la fin de liste**, on peut faire `append` en temps $\Theta(1)$
- ▶ Si on utilise une **liste doublement chaînée** (en plus du pointeur sur la fin), on peut faire `pop` en temps $\Theta(1)$

Une seconde idée :

- ▶ On représente les éléments **à l'envers** dans la liste chaînée
- ▶ ajouter/supprimer à la fin devient ajouter/supprimer au début $\rightarrow \Theta(1)$
- ▶ **Attention** les indices sont inversés pour `value_at`

On peut s'en sortir en temps $\Theta(1)$ avec des listes (enrichies)

Utilisation de tableaux

- ▷ `contains(S, x)`: Est-ce que la valeur x est dans S ?
- ▷ `value_at(S, i)`: Quel est la valeur en position i dans S ?
- ▷ `append(S, x)`: Ajoute x à la fin de S
- ▷ `pop(S)`: Supprime le dernier élément de S

Ca doit vous rappeler quelque-chose ...

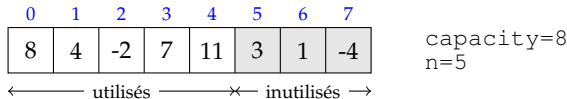
Utilisation de tableaux

- ▷ `contains(S, x)`: Est-ce que la valeur x est dans S ?
- ▷ `value_at(S, i)`: Quel est la valeur en position i dans S ?
- ▷ `append(S, x)`: Ajoute x à la fin de S
- ▷ `pop(S)`: Supprime le dernier élément de S

Ca doit vous rappeler quelque-chose ...

Ce sont les **piles** de l'exercice 1 du **TD 5**!

- ▷ On utilise un tableau trop grand
- ▷ On sait quelle partie est utilisée



- ▷ Les complexités sont en $\Theta(1)$, sauf `contains(S, x)` en $\Theta(n)$

Bilan sur les structures dynamiques de pile

	contains	value_at	append / pop
tableaux*	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
listes	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
listes enrichies	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

- ▶ Les tableaux sont encore **théoriquement** plus adaptés dans ce cas
- ▶ Mais ils ont un **problème** !

Bilan sur les structures dynamiques de pile

	contains	value_at	append / pop
tableaux*	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
listes	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
listes enrichies	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

- ▷ Les tableaux sont encore **théoriquement** plus adaptés dans ce cas
- ▷ Mais ils ont un **problème** !

* *tant qu'on ne dépasse pas la capacité*

*Comment faire si on ne connaît pas le nombre maximum d'éléments ?
allouer un tableau énorme est beaucoup trop couteux en mémoire*

II. Tableaux dynamiques

les listes-tableaux de Python

Augmentation de capacité

- ▶ L'idée est la suivante : si on n'a **plus assez de place**, on **double** la capacité
- ▶ Il faut **ré-allouer** un tableau et recopier les éléments

```
void double_capacity(stack *S) {  
    S->capacity *= 2;  
    int *elements = (int *)malloc(S->capacity * sizeof(int));  
    for(int i=0; i<S->n; i++)  
        elements[i] = S->elements[i];  
    free(S->elements);  
    S->elements = elements;  
}
```

```
void push(stack *S, int x) {  
    if (S->n == S->capacity)  
        double_capacity(S);  
    S->elements[S->n] = x;  
    S->n ++;  
}
```

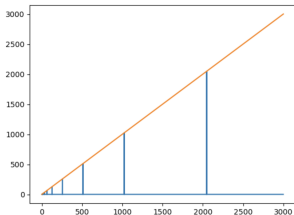
Complexité

```
void double_capacity(stack *S) {  
    S->capacity *= 2;  
    int *elements = (int *)malloc(S->capacity * sizeof(int));  
    for(int i=0; i<S->n; i++)  
        elements[i] = S->elements[i];  
    free(S->elements);  
    S->elements = elements;  
}
```

- ▷ **Problème** : la complexité devient $\Theta(n)$ quand on déclenche l'appel à `double_capacity` !

Si on regarde le nombre d'écritures dans le tableau (ligne la plus effectuée) lors d'une insertion on obtient le graphique ci-contre :

- ▷ en **bleu** le nombre d'insertions
- ▷ en **orange** la droite $x \mapsto x + 1$



Complexité pire cas

Complexité pire cas : insérer dans un tableau dynamique de taille n est en $\Theta(n)$

Mais alors, cette structure est inefficace ?

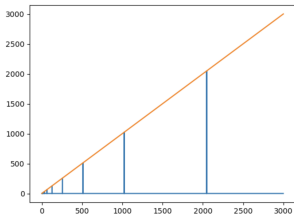
Complexité pire cas

Complexité pire cas : insérer dans un tableau dynamique de taille n est en $\Theta(n)$

Mais alors, cette structure est inefficace ?

Elle est “rarement inefficace”. Mais la notion de **complexité dans le pire cas** ne permet pas de capter cette bonne propriété.

On ne veut pas non plus faire de la complexité en moyenne ici, c’est même difficile à définir (qu’est-ce qui est aléatoire ? on insère juste des valeurs)



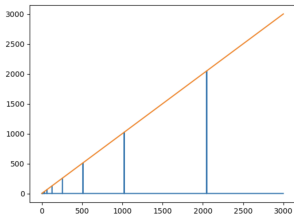
Complexité pire cas

Complexité pire cas : insérer dans un tableau dynamique de taille n est en $\Theta(n)$

Mais alors, cette structure est inefficace ?

Elle est “rarement inefficace”. Mais la notion de **complexité dans le pire cas** ne permet pas de capter cette bonne propriété.

On ne veut pas non plus faire de la complexité en moyenne ici, c’est même difficile à définir (qu’est-ce qui est aléatoire ? on insère juste des valeurs)



On a besoin d’une autre notion de complexité

Complexité amortie

Définition version Θ . Un algorithme est de **complexité amortie** $\Theta(t_n)$ quand n appels à cet algorithme se font en temps total $\Theta(n \times t_n)$

- ▷ On ne regarde pas un seul appel à l'algorithme
- ▷ On calcule la **complexité cumulée** de n appels
- ▷ On divise par n le résultat pour avoir la **complexité amortie**

→ Cela permet de quantifier qu'il ne peut pas y avoir beaucoup de pires cas consécutivement

Définition version \mathcal{O} . Un algorithme est de **complexité amortie** $\mathcal{O}(t_n)$ quand n appels à cet algorithme se font en temps total $\mathcal{O}(n \times t_n)$

Complexité amortie et tableaux dynamiques

Définition version Θ . Un algorithme est de **complexité amortie** $\Theta(t_n)$ quand n appels à cet algorithme se font en temps total $\Theta(n \times t_n)$

Théorème. L'ajout à la fin d'un tableau dynamique, à partir d'un tableau vide, a une complexité amortie en $\Theta(1)$.

	contains	value_at	append
tableaux dynamiques	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$ amortie
listes	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
listes enrichies	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

C'est une structure de données très utilisée :

- ▶ listes/tableaux de Python
- ▶ ArrayList en Java

Complément : preuve du théorème

Théorème. L'ajout à la fin d'un tableau dynamique, à partir d'un tableau vide, a une complexité amortie en $\Theta(1)$.

Preuve. On compte le nombre d'écritures dans un tableau, qui correspond bien aux lignes les plus effectuées. On a deux cas selon la taille t du tableau :

- ▷ si $t = 2^k$ est une puissance de 2, alors redimensionner le tableau et recopier les valeurs coûte t écritures, plus 1 pour la nouvelle valeur
- ▷ sinon, cela coûte juste 1 écriture

Si on compte le nombre E_n d'écritures pour n insertions, on a donc

$$E_n = n + \sum_{k=0}^m 2^k, \text{ où } m = \lfloor \log_2(n) \rfloor$$

Comme $\sum_{k=0}^m 2^k = 2^{m+1} - 1 \leq 2 \times 2^m \leq 2n$, on a $E_n \leq 3n$

On a aussi $E_n \geq n$, et donc la complexité de n insertions est en $\Theta(n)$.

On a bien montré que la complexité amortie de l'insertion est en $\Theta(1)$

III. Ensembles et fonctions

Sets & Maps

Cahier des charges des ensembles

On change le cahier des charges pour représenter des **ensembles** :

- ▷ `init()` : initialise un ensemble vide
- ▷ `add(S, x)` : ajoute x à l'ensemble S s'il n'y est pas déjà (pas de doublon dans un ensemble)
- ▷ `contains(S, x)` : teste si x est dans S ($x \in S$)
- ▷ `remove(S, x)` : enlève x de l'ensemble S

La structure n'est pas ordonnée : pas de i -ème élément

Cahier des charges des ensembles

On change le cahier des charges pour représenter des **ensembles** :

- ▷ `init()` : initialise un ensemble vide
- ▷ `add(S, x)` : ajoute x à l'ensemble S s'il n'y est pas déjà (pas de doublon dans un ensemble)
- ▷ `contains(S, x)` : teste si x est dans S ($x \in S$)
- ▷ `remove(S, x)` : enlève x de l'ensemble S

La structure n'est pas ordonnée : pas de i -ème élément

On peut réutiliser les **tableaux** ou les **listes** :

- ▷ `init()` est en $\Theta(1)$
- ▷ `add`, `contains` et `remove` sont en $\Theta(n)$: il faut vérifier si $x \in S$ avant de l'ajouter, pour savoir où supprimer, etc.

La complexité amortie n'aide pas, il faut vraiment parcourir la structure pour les opérations hors `init`

Mots dans un roman avec un tableau dynamique

```
S = []  
with open('la_pestes.txt', 'r') as fichier:  
    texte = fichier.read().strip()  
    mots = texte.split()  
    for mot in mots:  
        if mot not in S:  
            S.append(mot)
```

Le temps mis sur un roman de 524 kilo-octets est environ 2 secondes

Fonctions mathématiques (maps)

Comprendre *fonction mathématique*, et pas fonction d'un programme :

Une **fonction** (**map**) f associe à des éléments d'un ensemble de départ K un élément dans l'ensemble d'arrivée V

Exemples.

$$\triangleright f_1('ab') = 7, f_1('abb') = 17 \text{ et } f_1('bb') = 12$$

$$\triangleright f_2(4) = 7.3, f_2(8) = 12.2 \text{ et } f_2(-2) = 1.2$$

Définition (informatique). On appelle **clés** (**keys**) les éléments de K et **valeurs** (**values**) les éléments de V

Exemples.

\triangleright $'ab'$ et $'bb'$ sont des **clés** de f_1

\triangleright 8 et -2 sont des **clés** de f_2

\triangleright pour f_1 , 17 est la **valeur** associée à la **clé** $'abb'$

Cahier des charges pour les maps

*On utilise le terme anglais **map** pour ne pas confondre avec les **fonctions** d'un programme*

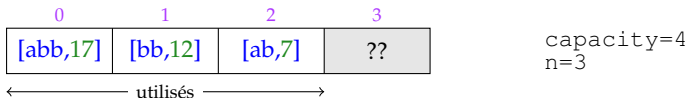
Le cahier des charges pour représenter des **maps** :

- ▷ `init()` : initialise une map sans clé
- ▷ `put(M, k, v)` : attribue la valeur `v` à la clé `k` dans `M`. S'il y avait déjà une valeur attribuée à la clé `k`, elle est changée
- ▷ `get(M, k)` : renvoie la valeur associée à la clé `k` dans `M`
- ▷ `contains(M, k)` : teste si la clé `k` est dans `M`
- ▷ `remove(S, k)` : enlève la clé `k` (et la valeur associée) de `M`

*La structure n'est pas ordonnée : pas de *i*-ème élément*

Utilisation d'un tableau dynamique

On peut stocker les paires [clé, valeur] dans un tableau dynamique : la fonction $f_1('ab') = 7$, $f_1('abb') = 17$ et $f_1('bb') = 12$ est représentée par



Au passage : on n'a pas mis les paires clé/valeurs dans le même ordre

Utilisons cette structure et implantons le **cahier des charges** des **maps**

Avec des listes Python

- ▷ `init()` : initialise une map sans clé
- ▷ `put(M, k, v)` : attribue la valeur `v` à la clé `k` dans `M`
- ▷ `get(M, k)` : renvoie la valeur associée à la clé `k` dans `M`
- ▷ `remove(S, k)` : enlève la clé `k` (et la valeur associée) de `M`

```
def init():  
    return []
```

```
def put(M, k, v):  
    for i in range(len(M)):  
        if M[i][0] == k:  
            M[i][1] = v  
            return  
    M.append([k, v])
```

```
def get(M, k):  
    for i in range(len(M)):  
        if M[i][0] == k:  
            return M[i][1]  
    return None
```

```
def remove(M, k):  
    for i in range(len(M)):  
        if M[i][0] == k:  
            M.pop(i)  
            return
```

Avec des listes Python

- ▷ `init()` : initialise une map sans clé
- ▷ `put(M, k, v)` : attribue la valeur `v` à la clé `k` dans `M`
- ▷ `get(M, k)` : renvoie la valeur associée à la clé `k` dans `M`
- ▷ `remove(S, k)` : enlève la clé `k` (et la valeur associée) de `M`

```
def init():  
    return []
```

```
def put(M, k, v):  
    for i in range(len(M)):  
        if M[i][0] == k:  
            M[i][1] = v  
            return  
    M.append([k, v])
```

```
def get(M, k):  
    for i in range(len(M)):  
        if M[i][0] == k:  
            return M[i][1]  
    return None
```

```
def remove(M, k):  
    for i in range(len(M)):  
        if M[i][0] == k:  
            M.pop(i)  
            return
```

Complexité : à part `init`, elles sont en $\Theta(n)$, où n est le nombre de clés

Application : nombre d'occurrences des mots

On peut s'en servir pour compter le nombre de fois qu'apparaît chaque mot dans un texte :

```
M = init()
with open('la_pestre.txt', 'r') as fichier:
    texte = fichier.read().strip()
    mots = texte.split()
    for mot in mots:
        if get(M, mot) is None:
            put(M, mot, 1)
        else:
            put(M, mot, get(M, mot, 1) + 1)
```

Remarque : si on prend le `else`, on parcourt **3 fois** le tableau pour chaque mot ! (on pourrait ne le faire qu'une fois mais sans utiliser nos fonctions du cahier des charges)

Pour le roman *La Peste*, cela prend près de **32** secondes !

Utilisation de dictionnaires

Les **dictionnaires** vus en AP1 remplissent le cahier des charges des **maps**, on peut compter les mots avec :

```
M = {}  
with open('la_pestes.txt', 'r') as fichier:  
    texte = fichier.read().strip()  
    mots = texte.split()  
    for mot in mots:  
        if mot not in M: M[mot] = 1  
        else: M[mot] += 1
```

Au lieu de 32 secondes, cela prend maintenant 0.02 secondes !

*Les **dictionary** **Python** utilisent une structure de données plus efficace*

Complément : les ensembles en Python

Les ensembles existent en Python, ce sont des `set`

```
S = set()
S.add("L2")
S.add("algo")
S.add("algo")
print("S :", S)
print(" 'toto' est dans S :", 'toto' in S)
print(" 'algo' est dans S :", 'algo' in S)
S.remove("L2")
print("S :", S)
```

```
$ python exemple_sets.py
S : {'L2', 'algo'}
'toto' est dans S : False
'algo' est dans S : True
S : {'algo'}
```

Retour sur l'ensemble des mots

On avait créé l'ensemble des mots avec une liste Python, en vérifiant qu'un mot n'était pas déjà présent avant de l'ajouter.

À la place, on peut utiliser un `set` de Python

```
S = set()
with open('la_pestes.txt', 'r') as fichier:
    texte = fichier.read().strip()
    mots = texte.split()
    for mot in mots:
        S.add(mot)
        # if mot not in S:
        #     S.append(mot)
```

On passe de 2 secondes à 0.02 secondes !

Les `set` de Python utilisent une structure de données plus efficace

IV. Tables de hachage

structure pour les set et dictionary

Principe pour les ensembles

L'idée est d'utiliser

- ▷ un **tableau trop grand** T , de longueur m , appelé la **table**
- ▷ une fonction h qui associe à une **clé** k un entier positif $h(k)$, appelée la **fonction de hachage**

On souhaite implanter le cahier des charges de la façon suivante :

- ▷ pour **ajouter** la clé x , on la met dans la case $T[h(x)]$
- ▷ pour **chercher** si x est dans T , on regarde dans la case $T[h(x)]$
- ▷ pour **supprimer** la clé x , on l'enlève de la case $T[h(x)]$

Remarque : $h(x)$ peut être trop grand, donc on utilise $h(x) \bmod m$
(en Python : `h(x) % m`)

0	1	2	3	4	5	6	7

Principe pour les ensembles

L'idée est d'utiliser

- ▷ un **tableau trop grand** T , de longueur m , appelé la **table**
- ▷ une fonction h qui associe à une **clé** k un entier positif $h(k)$, appelée la **fonction de hachage**

On souhaite implanter le cahier des charges de la façon suivante :

- ▷ pour **ajouter** la clé x , on la met dans la case $T[h(x)]$
- ▷ pour **chercher** si x est dans T , on regarde dans la case $T[h(x)]$
- ▷ pour **supprimer** la clé x , on l'enlève de la case $T[h(x)]$

Remarque : $h(x)$ peut être trop grand, donc on utilise $h(x) \bmod m$
(en **Python** : `h(x) % m`)

				AC			
0	1	2	3	4	5	6	7

Éléments ajoutés : "AC" (4),

Principe pour les ensembles

L'idée est d'utiliser

- ▷ un **tableau trop grand** T , de longueur m , appelé la **table**
- ▷ une fonction h qui associe à une **clé** k un entier positif $h(k)$, appelée la **fonction de hachage**

On souhaite implanter le cahier des charges de la façon suivante :

- ▷ pour **ajouter** la clé x , on la met dans la case $T[h(x)]$
- ▷ pour **chercher** si x est dans T , on regarde dans la case $T[h(x)]$
- ▷ pour **supprimer** la clé x , on l'enlève de la case $T[h(x)]$

Remarque : $h(x)$ peut être trop grand, donc on utilise $h(x) \bmod m$
(en Python : `h(x) % m`)

	PR			AC			
0	1	2	3	4	5	6	7

Éléments ajoutés : "AC" (4), "PR" (1),

Principe pour les ensembles

L'idée est d'utiliser

- ▷ un **tableau trop grand** T , de longueur m , appelé la **table**
- ▷ une fonction h qui associe à une **clé** k un entier positif $h(k)$, appelée la **fonction de hachage**

On souhaite implanter le cahier des charges de la façon suivante :

- ▷ pour **ajouter** la clé x , on la met dans la case $T[h(x)]$
- ▷ pour **chercher** si x est dans T , on regarde dans la case $T[h(x)]$
- ▷ pour **supprimer** la clé x , on l'enlève de la case $T[h(x)]$

Remarque : $h(x)$ peut être trop grand, donc on utilise $h(x) \bmod m$
(en **Python** : `h(x) % m`)

	PR	VB		AC			
0	1	2	3	4	5	6	7

Éléments ajoutés : "AC" (4), "PR" (1), "VB" (10),

Principe pour les ensembles

L'idée est d'utiliser

- ▷ un **tableau trop grand** T , de longueur m , appelé la **table**
- ▷ une fonction h qui associe à une **clé** k un entier positif $h(k)$, appelée la **fonction de hachage**

On souhaite implanter le cahier des charges de la façon suivante :

- ▷ pour **ajouter** la clé x , on la met dans la case $T[h(x)]$
- ▷ pour **chercher** si x est dans T , on regarde dans la case $T[h(x)]$
- ▷ pour **supprimer** la clé x , on l'enlève de la case $T[h(x)]$

Remarque : $h(x)$ peut être trop grand, donc on utilise $h(x) \bmod m$
(en `Python` : `h(x) % m`)

	PR	VB		AC			MvdB
0	1	2	3	4	5	6	7

Éléments ajoutés : "AC" (4), "PR" (1), "VB" (10), "MvdB" (15),

Principe pour les ensembles

L'idée est d'utiliser

- ▷ un **tableau trop grand** T , de longueur m , appelé la **table**
- ▷ une fonction h qui associe à une **clé** k un entier positif $h(k)$, appelée la **fonction de hachage**

On souhaite implanter le cahier des charges de la façon suivante :

- ▷ pour **ajouter** la clé x , on la met dans la case $T[h(x)]$
- ▷ pour **chercher** si x est dans T , on regarde dans la case $T[h(x)]$
- ▷ pour **supprimer** la clé x , on l'enlève de la case $T[h(x)]$

Remarque : $h(x)$ peut être trop grand, donc on utilise $h(x) \bmod m$
(en Python : `h(x) % m`)

	PR	VB		AC			MvdB
0	1	2	3	4	5	6	7

"PR" $\in T$: on calcule $h(PR) = 1$ et on le trouve en $T[1]$

Principe pour les ensembles

L'idée est d'utiliser

- ▷ un **tableau trop grand** T , de longueur m , appelé la **table**
- ▷ une fonction h qui associe à une **clé** k un entier positif $h(k)$, appelée la **fonction de hachage**

On souhaite implanter le cahier des charges de la façon suivante :

- ▷ pour **ajouter** la clé x , on la met dans la case $T[h(x)]$
- ▷ pour **chercher** si x est dans T , on regarde dans la case $T[h(x)]$
- ▷ pour **supprimer** la clé x , on l'enlève de la case $T[h(x)]$

Remarque : $h(x)$ peut être trop grand, donc on utilise $h(x) \bmod m$
(en **Python** : `h(x) % m`)

	PR	VB		AC			MvdB
0	1	2	3	4	5	6	7

"GR" $\notin T$: on calcule $h(GR) = 11$ et il n'est pas en $T[3]$

Trois points à régler

	PR	VB		AC			MvdB
0	1	2	3	4	5	6	7

Trois points à régler

	PR	VB		AC			MvdB
0	1	2	3	4	5	6	7

- ▶ Que faire si on veut ajouter dans une case déjà occupée ? Par exemple la clé "CN" avec $h(CN) = 4$?
- ▶ Comment choisir la fonction de hachage h ?
- ▶ Comment choisir la taille m de la table T ?

Gestion des collisions

Définition. On dit qu'il y a une **collision** quand deux clés sont envoyées sur la même case de la table. Mathématiquement :

$h(x) \equiv h(y) \pmod{m}$ pour deux clés $x \neq y$

Il existe deux façons principales de gérer les collisions

- ▷ **Externe** : chaque case contient une liste des clés (ou un tableau dynamique)
- ▷ **Interne** : si une autre clé est déjà présente, on insère ailleurs dans la table (un peu plus loin)

Python utilise du hachage interne, Java du hachage externe

Table de hachage externe

On met un **tableau dynamique** dans chaque case (ou une liste)

```
def init(m):  
    return [[] for _ in range(m)]  
  
def contains(T, x):  
    h = hash(x) % len(T)  
    return x in T[h]  
  
def add(T, x):  
    h = hash(x) % len(T)  
    if x not in T[h]:  
        T[h].append(x)  
  
def remove(T, x):  
    h = hash(x) % len(T)  
    if x in T[h]:  
        T[h].remove(x)
```

- ▶ On calcule $h := h(x) \bmod m$ et on délègue les opérations à $T[h]$
- ▶ Les complexités (sauf `init`) sont en $\Theta(\ell)$, où ℓ est la taille du tableau $T[h]$

*Si on l'applique au roman, on crée l'ensemble des mots en 0.047 secondes !
(pour $m=1000$)*

Table de hachage interne : add

- ▶ Si on veut tout stocker dans la table, il faut utiliser une autre case que $h(x) \bmod m$ pour stocker x en cas de collision
- ▶ L'idée est de le mettre dans la case d'après (ou celle d'encore après, etc)

	PR	VB		AC			MvdB
0	1	2	3	4	5	6	7

Table de hachage interne : add

- ▶ Si on veut tout stocker dans la table, il faut utiliser une autre case que $h(x) \bmod m$ pour stocker x en cas de **collision**
- ▶ L'idée est de le mettre dans la case d'après (ou celle d'encore après, etc)

	PR	VB		AC			MvdB
0	1	2	3	4	5	6	7


↑
CN

Ajout de "CN" avec $h(\text{CN}) = 4$

Table de hachage interne : add

- ▶ Si on veut tout stocker dans la table, il faut utiliser une autre case que $h(x) \bmod m$ pour stocker x en cas de collision
- ▶ L'idée est de le mettre dans la case d'après (ou celle d'encore après, etc)

	PR	VB		AC			MvdB
0	1	2	3	4	5	6	7




Ajout de "CN" avec $h(CN) = 4$

Table de hachage interne : add

- ▶ Si on veut tout stocker dans la table, il faut utiliser une autre case que $h(x) \bmod m$ pour stocker x en cas de collision
- ▶ L'idée est de le mettre dans la case d'après (ou celle d'encore après, etc)

	PR	VB		AC	CN		MvdB
0	1	2	3	4	5	6	7



Ajout de "CN" avec $h(CN) = 4$

Table de hachage interne : add

- ▶ Si on veut tout stocker dans la table, il faut utiliser une autre case que $h(x) \bmod m$ pour stocker x en cas de collision
- ▶ L'idée est de le mettre dans la case d'après (ou celle d'encore après, etc)

	PR	VB		AC	CN		MvdB
0	1	2	3	4	5	6	7

Ajout de "NF" avec $h(NF) = 4$

Table de hachage interne : add

- ▶ Si on veut tout stocker dans la table, il faut utiliser une autre case que $h(x) \bmod m$ pour stocker x en cas de **collision**
- ▶ L'idée est de le mettre dans la case d'après (ou celle d'encore après, etc)

	PR	VB		AC	CN		MvdB
0	1	2	3	4	5	6	7


↑
NF

Ajout de "NF" avec $h(NF) = 4$

Table de hachage interne : add

- ▶ Si on veut tout stocker dans la table, il faut utiliser une autre case que $h(x) \bmod m$ pour stocker x en cas de **collision**
- ▶ L'idée est de le mettre dans la case d'après (ou celle d'encore après, etc)

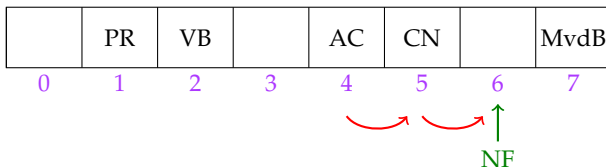
	PR	VB		AC	CN		MvdB
0	1	2	3	4	5	6	7



Ajout de "NF" avec $h(NF) = 4$

Table de hachage interne : add

- ▶ Si on veut tout stocker dans la table, il faut utiliser une autre case que $h(x) \bmod m$ pour stocker x en cas de **collision**
- ▶ L'idée est de le mettre dans la case d'après (ou celle d'encore après, etc)




Ajout de "NF" avec $h(NF) = 4$

Table de hachage interne : add

- ▶ Si on veut tout stocker dans la table, il faut utiliser une autre case que $h(x) \bmod m$ pour stocker x en cas de collision
- ▶ L'idée est de le mettre dans la case d'après (ou celle d'encore après, etc)

	PR	VB		AC	CN	NF	MvdB
0	1	2	3	4	5	6	7



Ajout de "NF" avec $h(NF) = 4$

Table de hachage interne : add

- ▶ Si on veut tout stocker dans la table, il faut utiliser une autre case que $h(x) \bmod m$ pour stocker x en cas de collision
- ▶ L'idée est de le mettre dans la case d'après (ou celle d'encore après, etc)

	PR	VB		AC	CN	NF	MvdB
0	1	2	3	4	5	6	7

Ajout de "CD" avec $h(CD) = 6$

Table de hachage interne : add

- ▶ Si on veut tout stocker dans la table, il faut utiliser une autre case que $h(x) \bmod m$ pour stocker x en cas de collision
- ▶ L'idée est de le mettre dans la case d'après (ou celle d'encore après, etc)

	PR	VB		AC	CN	NF	MvdB
0	1	2	3	4	5	6	7


↑
CD

Ajout de "CD" avec $h(CD) = 6$

Table de hachage interne : add

- ▶ Si on veut tout stocker dans la table, il faut utiliser une autre case que $h(x) \bmod m$ pour stocker x en cas de collision
- ▶ L'idée est de le mettre dans la case d'après (ou celle d'encore après, etc)

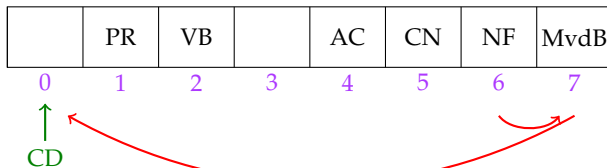
	PR	VB		AC	CN	NF	MvdB
0	1	2	3	4	5	6	7



Ajout de "CD" avec $h(CD) = 6$

Table de hachage interne : add

- ▷ Si on veut tout stocker dans la table, il faut utiliser une autre case que $h(x) \bmod m$ pour stocker x en cas de **collision**
- ▷ L'idée est de le mettre dans la case d'après (ou celle d'encore après, etc)



Ajout de "CD" avec $h(CD) = 6$

Table de hachage interne : add

- ▶ Si on veut tout stocker dans la table, il faut utiliser une autre case que $h(x) \bmod m$ pour stocker x en cas de **collision**
- ▶ L'idée est de le mettre dans la case d'après (ou celle d'encore après, etc)

CD	PR	VB		AC	CN	NF	MvdB
0	1	2	3	4	5	6	7



Ajout de "CD" avec $h(CD) = 6$

Table de hachage interne : **contains**

- ▷ Pour chercher x il faut continuer de case en case jusqu'à tomber sur x ou sur une case vide

	PR	VB		AC	CN	NF	MvdB
0	1	2	3	4	5	6	7

Recherche de "NF" avec $h(NF) = 4$

Table de hachage interne : **contains**

- Pour chercher x il faut continuer de case en case jusqu'à tomber sur x ou sur une case vide

	PR	VB		AC	CN	NF	MvdB
0	1	2	3	4	5	6	7


↑
NF

Recherche de "NF" avec $h(NF) = 4$

Table de hachage interne : **contains**

- Pour chercher x il faut continuer de case en case jusqu'à tomber sur x ou sur une case vide

	PR	VB		AC	CN	NF	MvdB
0	1	2	3	4	5	6	7




Recherche de "NF" avec $h(NF) = 4$

Table de hachage interne : contains

- Pour chercher x il faut continuer de case en case jusqu'à tomber sur x ou sur une case vide

	PR	VB		AC	CN	NF	MvdB
0	1	2	3	4	5	6	7



Recherche de "NF" avec $h(NF) = 4$ ✓

Table de hachage interne : **contains**

- Pour chercher x il faut continuer de case en case jusqu'à tomber sur x ou sur une case vide

	PR	VB		AC	CN	NF	MvdB
0	1	2	3	4	5	6	7

Recherche de "CD" avec $h(CD) = 6$

Table de hachage interne : **contains**

- Pour chercher x il faut continuer de case en case jusqu'à tomber sur x ou sur une case vide

	PR	VB		AC	CN	NF	MvdB
0	1	2	3	4	5	6	7


↑
CD

Recherche de "CD" avec $h(CD) = 6$

Table de hachage interne : **contains**

- Pour chercher x il faut continuer de case en case jusqu'à tomber sur x ou sur une case vide

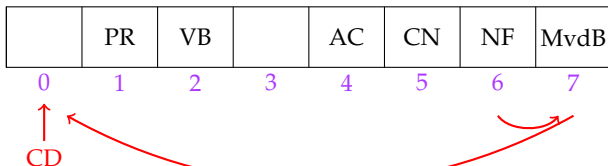
	PR	VB		AC	CN	NF	MvdB
0	1	2	3	4	5	6	7



Recherche de "CD" avec $h(CD) = 6$

Table de hachage interne : **contains**

- Pour chercher x il faut continuer de case en case jusqu'à tomber sur x ou sur une case vide



Recherche de "CD" avec $h(CD) = 6$ X

Table de hachage interne : **remove**

- ▷ On ne peut pas juste faire une recherche et supprimer la clé

	PR	VB		AC	CN	NF	MvdB
0	1	2	3	4	5	6	7

Avec $f(AC) = f(CN) = f(NF) = 4$

Table de hachage interne : **remove**

- ▷ On ne peut pas juste faire une recherche et supprimer la clé

	PR	VB			CN	NF	MvdB
0	1	2	3	4	5	6	7

Avec $f(AC) = f(CN) = f(NF) = 4$

On supprime AC → on ne peut plus retrouver CN et NF

Table de hachage interne : **remove**

- ▷ On ne peut pas juste faire une recherche et supprimer la clé

	PR	VB		X	CN	NF	MvdB
0	1	2	3	4	5	6	7

Avec $f(AC) = f(CN) = f(NF) = 4$

Une solution : **marquer** la case pour signaler qu'il y a eu une suppression auparavant

- ▷ On peut **ajouter** une clé dans une **case marquée**
- ▷ Si on tombe sur une **case marquée** lors d'une **recherche**, on doit continuer à avancer

Récapitulatif sur le hachage avec gestion interne

- ▶ On gère les **collisions** en insérant (ou recherchant) plus loin que la position initiale (il existe d'autres stratégies que d'avancer de 1 en 1 pour limiter les tailles clusters)
- ▶ Il faut faire attention à ce qu'il y ait suffisamment de cases vides pour ne pas pénaliser les performances
- ▶ S'il y a des **suppressions**, il faut faire attention, on peut par exemple **marquer** les cases où il y a eu des suppressions
- ▶ Il y a des avantages à utiliser des portions contiguës de mémoire
- ▶ C'est la solution choisie en **Python** pour les **set** et **dictionary** (avec des variantes)

Fonction de hachage : tests expérimentaux

On considère 3 fonctions de hachages différentes pour les chaînes :

▷ `hash`, fournie par `Python`

▷ `ord_0`, l'encodage sous forme d'un entier du premier caractère :

```
ord_0(x) = ord(x[0])
```

▷ `ord_sum`, la somme des encodages entiers des caractères :

```
ord_sum(x) = sum(ord(c) for c in x)
```

On utilise notre code `Python` pour le hachage externe, sur les mots du roman **“La Peste”**, avec $m = 10\,000$.

hachage	temps (en secondes)	plus longue chaîne
<code>hash</code>	0.051	8
<code>ord_0</code>	0.207	1388
<code>ord_sum</code>	0.095	38

Fonction de hachage

Ce qu'on veut pour la fonction de hachage :

- ▷ Qu'elle **répartisse bien** les clés : deux clés différentes, même très similaires doivent avoir des valeurs de hachage très différentes
- ▷ Qu'elle se **calcule rapidement** : on va avoir besoin de calculer la valeur de hachage souvent

⇒ Idéalement, on voudrait que $h(x) \bmod m$ soit un entier au hasard (uniforme) de $\{0, \dots, m-1\}$, mais on ne peut pas utiliser le hasard, car il faut que $h(x)$ soit constant (qu'on puisse retrouver x après)

On veut donc une **fonction de hachage** qui soit **rapide à calculer** et qui se comporte **comme si on avait tiré au sort** chaque valeur, mais déterministe (sans effectuer de tirage au sort pour avoir toujours le même résultat) → **c'est compliqué !**

Fonction de hachage

Ce qu'on veut pour la fonction de hachage :

- ▶ Qu'elle **répartisse bien** les clés : deux clés différentes, même très similaires doivent avoir des valeurs de hachage très différentes
- ▶ Qu'elle se **calcule rapidement** : on va avoir besoin de calculer la valeur de hachage souvent

⇒ Idéalement, on voudrait que $h(x) \bmod m$ soit un entier au hasard (uniforme) de $\{0, \dots, m-1\}$, mais on ne peut pas utiliser le hasard, car il faut que $h(x)$ soit constant (qu'on puisse retrouver x après)

On veut donc une **fonction de hachage** qui soit **rapide à calculer** et qui se comporte **comme si on avait tiré au sort** chaque valeur, mais déterministe (sans effectuer de tirage au sort pour avoir toujours le même résultat) → **c'est compliqué !**

Règle d'or : ne faites pas vous-même vos fonctions de hachage, utilisez celle intégrées dans le langage ou trouvées dans des livres

Exemple “simple” : 64-bit FNV-1a

```
#define FNV_OFFSET 14695981039346656037UL
#define FNV_PRIME 1099511628211UL

// Return 64-bit FNV-1a hash for key (NUL-terminated).
static uint64_t hash_key(char *key)
{
    uint64_t hash = FNV_OFFSET;
    for (char *p = key; *p != 0; p++) {
        hash ^= (uint64_t) (unsigned char) (*p);
        hash *= FNV_PRIME;
    }
    return hash;
}
```

Dans les célèbres, il y a également [SHA-256](#) qui est sûre cryptographiquement (elle est utilisée pour les **bitcoins**)

Complexité des opérations (hachage externe)

On utilise n pour le nombre de clés dans la table de hachage, et m pour la taille de la table.

Pour le hachage externe :

- ▷ Dans le pire cas, toutes les clés ont la même valeur de hachage, les opérations (hors `init`) sont en $\Theta(n)$, on travaille avec une seule liste ← le pire cas n'est pas pertinent ici

Complexité des opérations (hachage externe)

On utilise n pour le nombre de clés dans la table de hachage, et m pour la taille de la table.

Pour le hachage externe :

- ▷ Dans le pire cas, toutes les clés ont la même valeur de hachage, les opérations (hors `init`) sont en $\Theta(n)$, on travaille avec une seule liste ← le pire cas n'est pas pertinent ici
- ▷ Une fonction de hachage idéale est modélisée par des valeurs aléatoires et avec ce modèle on obtient que les opérations sont en $\Theta(\frac{n}{m})$ en moyenne

Théorème. Si $\alpha := \frac{n}{m}$ est borné par une constante, alors les opérations dans une table de hachage externe ont une complexité moyenne $\Theta(1)$

C'est pour ça que c'est très efficace en pratique

Complexité des opérations (hachage interne)

On utilise n pour le nombre de clés dans la table de hachage, et m pour la taille de la table.

Pour le hachage interne :

- ▷ Le pire cas n'est toujours pas pertinent : on va avoir du $\Theta(n)$ si tous ont la même valeur de hachage
- ▷ Si le modèle aléatoire de h est raisonnable, on a également une complexité en $\Theta(\frac{n}{m})$ pour les opérations, sauf `init`
- ▷ Attention : on doit avoir $n \leq m$ sinon il n'y a plus de place !

Complexité des opérations (hachage interne)

On utilise n pour le nombre de clés dans la table de hachage, et m pour la taille de la table.

Pour le hachage interne :

- ▷ Le pire cas n'est toujours pas pertinent : on va avoir du $\Theta(n)$ si tous ont la même valeur de hachage
- ▷ Si le modèle aléatoire de h est raisonnable, on a également une complexité en $\Theta(\frac{n}{m})$ pour les opérations, sauf `init`
- ▷ **Attention** : on doit avoir $n \leq m$ sinon il n'y a plus de place !

Théorème. Si $\alpha := \frac{n}{m}$ est borné par une constante < 1 , alors les opérations dans une table de hachage interne ont une complexité moyenne $\Theta(1)$

Choix de m

Théorème. Si $\alpha := \frac{n}{m}$ est borné par une constante, alors les opérations dans une table de **hachage externe** ont une complexité moyenne $\Theta(1)$

Théorème. Si $\alpha := \frac{n}{m}$ est borné par une constante < 1 , alors les opérations dans une table de **hachage interne** ont une complexité moyenne $\Theta(1)$

- ▷ C'est presque le même théorème pour les deux solutions
- ▷ En pratique on impose $\alpha \leq \alpha_0$ avec $\alpha_0 \in [0.66, 0.8]$ pour le **hachage interne** et $\alpha_0 \in [0.66, 5]$ pour le **hachage externe**

Question : que faire si à force d'ajouter des clés, on a α qui devient supérieur à α_0 ?

Choix de m

Théorème. Si $\alpha := \frac{n}{m}$ est borné par une constante, alors les opérations dans une table de **hachage externe** ont une complexité moyenne $\Theta(1)$

Théorème. Si $\alpha := \frac{n}{m}$ est borné par une constante < 1 , alors les opérations dans une table de **hachage interne** ont une complexité moyenne $\Theta(1)$

- ▷ C'est presque le même théorème pour les deux solutions
- ▷ En pratique on impose $\alpha \leq \alpha_0$ avec $\alpha_0 \in [0.66, 0.8]$ pour le **hachage interne** et $\alpha_0 \in [0.66, 5]$ pour le **hachage externe**

Question : que faire si à force d'ajouter des clés, on a α qui devient supérieur à α_0 ?

→ On **redimensionne** la table en doublant m ! Et on ré-insère toutes les clés (on est obligé à cause du **mod** m , cf TD)

Bilan complexité et choix de m

- ▶ On **redimensionne la table** si le taux de remplissage $\alpha = \frac{n}{m}$ dépasse un certain seuil α_0 fixé à l'avance, en doublant la taille par exemple
- ▶ Comme la complexité **pire cas** n'est pas pertinente, on modélise la **fonction de hachage** par une **fonction aléatoire**
- ▶ On obtient ainsi des complexités en $\Theta(1)$ **amortie en moyenne** pour les opérations **add**, **remove** et **contains**
- ▶ C'est vrai pour le **hachage externe** et le **hachage interne**

En pratique :

- ▶ Si on suit la **règle d'or** et donc qu'on utilise des bonnes fonctions de hachage, le modèle aléatoire décrit bien le comportement
- ▶ On a des opérations **add**, **remove** et **contains** très efficaces
- ▶ Les **table de hachage** sont des structures de données très efficaces et très utilisées

Et les maps ?

Question : On n'a vu que les **sets**, peut-on utiliser les **tables de hachages** pour les **maps** ?

Et les maps ?

Question : On n'a vu que les **sets**, peut-on utiliser les **tables de hachages** pour les **maps** ?

OUI ! il suffit de

- ▷ Stocker des paires $[k, v]$, où k est la clé et v la valeur
- ▷ Manipuler les fonctions de hachage uniquement sur les clés k
- ▷ ... et ça fonctionne pareil

	PR	VB		AC			MvdB
	17	-8		7			9
0	1	2	3	4	5	6	7

$$\begin{array}{llll} f(AC) = 7 & f(PR) = 17 & f(VB) = -8 & f(MvdB) = 9 \\ h(AC) = 4 & h(PR) = 1 & h(VB) = 9 & h(MvdB) = 15 \end{array}$$

- ▷ Pareil pour le **hachage externe**, on stocke les paires $[k, v]$ dans les listes ou dans les tableaux dynamiques

Important : précision sur la complexité

Théorème. Les opérations `add`, `remove` et `contains` ont une complexité **amortie en moyenne** $\Theta(1)$

Attention : ce théorème considère que les opérations élémentaires sur les clés se font en temps constant $\Theta(1)$.

Important : précision sur la complexité

Théorème. Les opérations `add`, `remove` et `contains` ont une complexité **amortie en moyenne** $\Theta(1)$

Attention : ce théorème considère que les opérations élémentaires sur les clés se font en temps constant $\Theta(1)$.

En réalité il faut/faudrait prendre en compte :

- ▷ Le temps de calcul de la **valeur de hachage**, qui n'est par exemple pas constant pour une chaîne (même s'il est rapide)
- ▷ Le temps de calcul de la **comparaison de deux clés**, ce n'est pas non plus constant pour des chaînes

Pour la **véritable complexité** on peut dire qu'une opération coûte :

- ▷ un appel à la **fonction de hachage**
- ▷ $\Theta(1)$ appels, en amortie et en moyenne, à la **fonction de comparaison des clés**

Que devez-vous savoir ?

- ▷ Ce qu'est un **tableau dynamique**, comment il fonctionne algorithmiquement, notamment le **redimensionnement**
- ▷ Ce qu'est la **complexité amortie**
- ▷ La notion d'**ensemble** (**set**) et de **fonction** (**map**)
- ▷ Comment fonctionnent les **tables de hachage interne et externe**
- ▷ Ce qu'on attend d'une **bonne fonction de hachage**
- ▷ Que les complexités sont en $\Theta(1)$ **amortie en moyenne** si on utilise le redimensionnement pour garder un α en dessous d'un seuil α_0

Complément : mutable, non-mutable

En **Python** (et dans beaucoup de langages, comme **Java**) il y a des structures **modifiables** (**mutable**) et **non-modifiables** (**non-mutable**)

Par exemple :

- ▶ Une **chaîne** `s = "algo"` est **non-modifiable**
- ▶ Une **liste** `s = ['a', 'l', 'g', 'o']` est **modifiable**

Règle : on ne peut utiliser que des **structures non-modifiables** comme **clés** dans une table de hachage

En effet, la **valeur de hachage** étant calculée à partir de ce que contient la structure, elle change si on la modifie → on ne retrouvera pas la donnée dans la table !

Examen

A l'**examen** il y aura :

- ▶ Un **QCM facile** avec points négatifs portant sur les 5 cours
 - ▶ Pour des grandes données, est-ce qu'il vaut mieux utiliser un algorithme en $\Theta(n)$ ou un algorithme en $\Theta(n^2)$?
- ▶ Un **QCM normal** sans point négatif portant sur les 5 cours
 - ▶ Voilà un code `Python` et un test unitaire, quel est la couverture de code ? ☐ 40% ☐ 60% ☐ 80% ☐ 100%
- ▶ Une **preuve de correction** à rédiger
- ▶ Une **preuve de terminaison** à rédiger
- ▶ Une **analyse de complexité pire cas** à rédiger

Aucun document papier et aucun appareil électronique autorisés