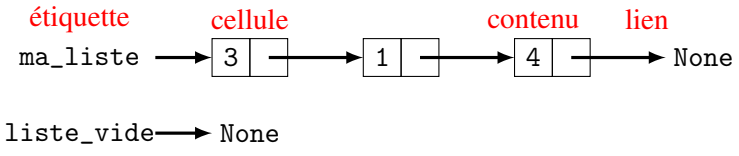
# 3. Structures de données II

Nous avons revu dans le chapitre **1. Structures de données I** les **principaux types de données abstraits (TDA)** que l'on peut retrouver dans le langage Python : tuple, tableau et dictionnaire.
Nous allons étudier ici plusieurs autres structures complexes qui ne sont pas implémentées directement en Python : la liste [1], la pile et la file. Ensuite nous travaillerons sur d'autres structures comme par exemple le dictionnaire, l'ensemble, ... Pour chacune de ces différentes structures nous allons

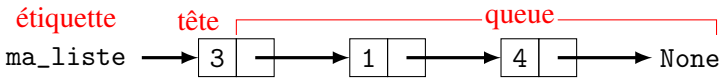
- voir ce qui est attendu (autrement dit, qu'elle est l'**interface** de la structure),
- programmer en python la structure (autrement dit, nous allons proposer une [2] **implémentation**)
- utiliser la structure dans des exercices d'application

## 3.1 Les listes (chaînées)

Une **liste** est une séquence ordonnée d'éléments, comme par exemple la liste d'entiers `ma_liste = [3, 1, 4]`.
La liste est constituée de trois cellules, représentées ici par des rectangles.
Chaque cellule possède une première partie, souvent nommée **car** (*Contents of the Address Register*, soit Contenue de la partie Adresse d'un Registre) , qui contient l'entier correspondant, et d'une deuxième partie, souvent nommée cdr (*Contents of the Decrement Register*, soit Contenue de la partie Décrément d'un Registre) , qui contient un lien vers la cellule suivante. On a donc affaire à une chaîne de cellules, ce qui justifie la dénomination courante de **liste chaînée**. Le champ **cdr** de la dernière cellule renvoie vers une liste vide, conventionnellement représentée ici par `None`.




Une liste peut être implémenté très simplement avec un tuple `(tête, queue)` où `tête` est le premier élément de la liste et `queue` est le reste de la liste qui est lui-même une liste...




1. Il existe bien les objets de type `list` en Python, mais il s'agit plutôt d'un tableau dynamique associé à des méthodes d'accès typiques des listes.
2. En réalité, nous proposerons au moins 2 implémentations différentes.

```
class Cellule :
    def __init__(self, tete, queue):
        self.tete = tete
        self.queue = queue
```

Pour créer une liste, il suffit d'encapsuler les `Cellules` :

```
>>> ma_liste = Cellule(3, Cellule(1, Cellule(4, None)))
```

On peut ensuite avoir accès à tous les éléments de la liste :

```
>>> ma_liste.tete
3
>>> ma_liste.queue.tete
1
>>> ma_liste.queue.queue.tete
4
```

### 3.1.1 Interface

On constate que cette implémentation un peu trop rudimentaire n'est pas vraiment pratique pour la manipulation des listes. Voici un interface un peu plus complet :

| Fonction | Description |
|---|---|
| `Liste` | crée une liste vide |
| `est_vide` | indique si la liste est vide |
| `ajouter_tete` | insère un élément en tête de liste |
| `renvoyer_tete` | renvoie la valeur de l'élément en tête de liste ET le supprime de la liste |

**exercice 3.1** Représenter l'état de la liste `ma_liste` à chaque étape. Le cas échéant donner le résultat obtenu dans la console.

```
>>> ma_liste = Liste()
>>> ma_liste.est_vide()
>>> ma_liste.ajouter_tete(4)
>>> ma_liste.ajouter_tete(1)
>>> ma_liste.ajouter_tete(3)
>>> ma_liste.est_vide()
>>> ma_liste.renvoyer_tete()
>>> ma_liste.renvoyer_tete()
>>> ma_liste.renvoyer_tete()
>>> ma_liste.est_vide()
>>> ma_liste.renvoyer_tete()
```

### 3.1.2 Implémentation

Il existe de multiple façon d'implémenter une liste chaînée. Dans l'implémentation proposée ci-dessous, on utilise deux objets. La `Cellule` que nous avons vue précédemment avec ces deux attributs `tete` et `queue` mais aussi l'objet `Liste` qui, à la création de liste ne contient que `None` et ensuite, est composée de plusieurs `Cellules` qui s'enchaînent les unes aux autres.

```
class Cellule :
    def __init__(self, tete, queue):
        self.tete = tete
        self.queue = queue

class Liste :
    def __init__(self):
        self.cellule = None

    def est_vide(self):
        return self.cellule is None

    def ajouter_tete(self, valeur):
        self.cellule = Cellule(valeur, self.cellule)

    def renvoyer_tete(self):
        assert self.cellule is not None, "Liste Vide !"
        tete = self.cellule.tete
        if self.cellule.queue is not None :
            self.cellule = Cellule(self.cellule.queue.tete, self.cellule.queue.
    queue)
        else :
            self.cellule = None
        return tete
```

**exercice 3.2** Voici un interface étendue d'une liste chaînée. A vous de le programmer (si vous le souhaitez, vous pouvez aussi utilisée les fonctions spéciales, cf chapitre précédent)

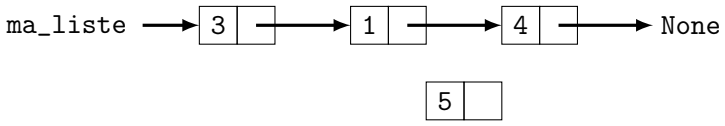
| Fonction | Description |
|---|---|
| `Liste` | crée une liste vide |
| `est_vide` | indique si la liste est vide |
| `ajouter_tete` | insère un élément en tête de liste |
| `renvoyer_tete` | renvoie la valeur de l'élément en tête de liste ET le supprime de la liste |
| `afficher` | affiche dans l'ordre la liste des éléments |
| `longueur` | renvoie la longueur de la liste, autrement dit le nombre d'éléments qui la compose (sans `None`) |
| `nieme_element` | renvoie le $n$-ième élément de la liste |
| `renverser` | inverse l'ordre de la liste |

### 3.1.3 Une structure mutable

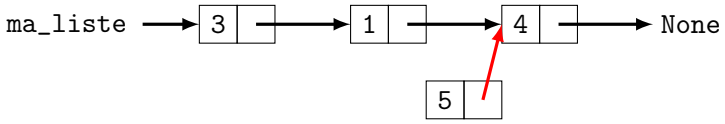
La liste chaînée est un TAD qui est facilement mutable.

#### 3.1.3.1 Insertion d'un élément

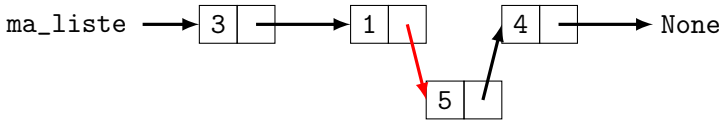
Par exemple, si nous voulons insérer le nombre 5 entre 1 et 4.




Premier étape, on crée un lien entre la cellule contenant 5 et la cellule contenant 4.




Deuxième étape, la cellule contenant 1 ne pointe plus vers la cellule contenant le 4 mais vers la cellule contenant le 5.

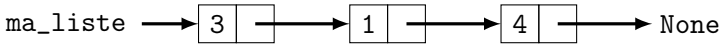



Voici la méthode correspondante :

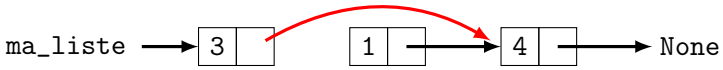
```
67      def inserer(self, valeur, position):
68          """insére dans la file une cellule contenant la valeur à la position
        voulue
69          cette position doit être inférieure à la taille de la liste
70          """
71          assert 0 <= position <= len(self), "erreur d'indexation"
72          if position == 0:
73              self.ajouter_tete(valeur)
74          else :
75              l = self.cellule
76              i = 0
77              # on cherche la cellule précédente
78              while i < position -1 :
79                  l = l.queue
80                  i = i + 1
81              nv_cellule = Cellule(valeur,l.queue)
82              l.queue = nv_cellule
```

#### 3.1.3.2 Suppression d'un élément

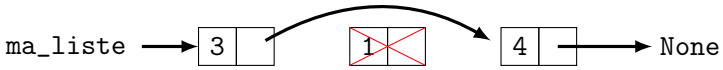
Par exemple, si nous voulons supprimer le 1.




Premier étape, la cellule contenant 3 ne pointe plus vers la cellule 1 mais vers la cellule contenant 4.




Deuxième étape, on supprime le lien entre la cellule contenant 1 et la cellule contenant le 4. Comme la cellule contenant le 1 n'est plus accessible, le **Garbage Collector** [3] de Python se charge de la supprimer de la mémoire.




Voici la méthode correspondante :

```
84 def supprimer(self, position):
85         """supprime dans la file une cellule à la position voulue
86         cette position doit être inférieure à la taille de la liste
87         """
88         assert 0 <= position < len(self), "erreur d'indexation"
89         if position == 0:
90             self.renvoyer_tete()
91         else :
92             l = self.cellule
93             i = 0
94             # on cherche la cellule précédente
95             while i < position -1 :
96                 l = l.queue
97                 i = i + 1
98             # on garde en mémoire la cellule à supprimer
99             a_supprim = l.queue
100            # création du nouveau lien
101            l.queue = l.queue.queue
102            # déconnexion de la cellule à supprimer
103            a_supprim.queue = None
```

### 3.1.4 Avantages et inconvénients de la liste chaînée

Si on compare la liste chaînée à la structure liste implémentée en Python (tableau dynamique), nous pouvons dire :

- accéder à n'importe quel élément d'un tableau dynamique se fait en temps constant $O(1)$, par contre, pour une liste chaînée, il est nécessaire de partir de la tête pour arriver à l'élément recherché, autrement dit dans le pire cas, la complexité est en $O(n)$ ;
- pour insérer ou supprimer un élément d'un tableau dynamique il est nécessaire de décaler tous les éléments qui suivent d'une case, la complexité, dans le pire cas est en $O(n)$, par contre pour une liste chaînée, supprimer ou rajouter un élément à n'importe quel endroit se fait en temps constant $O(1)$.

Voici d'autres comparaisons :

| Opération | Tableau dynamique | Liste chaînée |
|---|---|---|
| Accès à un élément | $O(1)$ | $O(n)$ |
| Supprimer un élément de la liste et renvoyer la liste | $O(n)$ | $O(1)$ |
| Rajouter un élément de la liste et renvoyer la liste | $O(n)$ | $O(1)$ |
| Compter le nombre d'éléments d'une liste | $O(1)$ | $O(n)$ |

3. Le Garbage Collector ou ramasse miette ou gestionnaire automatique de mémoire n'est pas présent dans tous les langages. Dans certain, comme le **C** par exemple, il ne faudra surtout pas oublier de supprimer cette cellule sous peine de saturer rapidement la mémoire vive de l'ordinateur !

**exercice 3.3** On propose les deux implémentations suivantes d'une liste :

Implémentation avec des tuples :

```
def creer_liste():
    return ()

def inserer(liste, element):
    return (element, liste)

def afficher(liste):
    reponse = []
    while liste != ():
        tete, liste = liste
        reponse.append(tete)
    return reponse
```

Implémentation avec des tableaux dynamiques :

```
def creer_liste():
    return []

def inserer(liste, element):
    liste.append(element)
    return liste

def afficher(liste):
    return liste
```

On exécute les commandes suivantes dans la console :

```
>>> l1 = creer_liste()
>>> l1 = inserer(l1, 'Heddy')
>>> l2 = inserer(l1, 'John')
>>> l1 = inserer(l1, 'Ada')
>>> print("l1 =", afficher(l1))
>>> print("l2 =", afficher(l2))
```

1. Quelle constatez-vous ?
2. Comment expliquer ces différences ?

*Exercice inspiré par le manuel « Numérique et Sciences Informatiques - Terminale » Hachette Education* ■

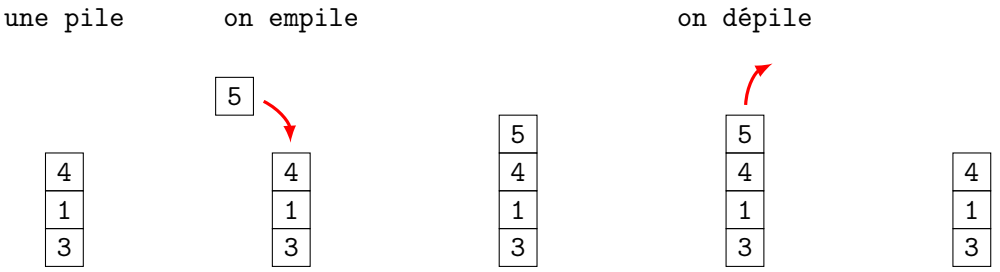
## 3.2 Les piles

Une **pile** est une liste qui se gère comme un empilement d'éléments. Penser à une pile d'assiettes :

- si on rajoute une assiette, elle se situe en haut de la pile,
- si on veut enlever une assiette de la pile, ce sera uniquement celle du haut de la pile.

On dit que l'on applique la méthode **LIFO** (Last In First Out), autrement dit le dernier arrivé sera le premier à partir.

On trouve beaucoup d'applications à l'utilisation des piles. Par exemple, lorsque l'on corrige une faute de frappe sur son ordinateur, on utilise très souvent l'association de touches `Ctrl+z` pour revenir en arrière sur toutes nos actions passées. La liste de nos actions a donc été enregistré dans une pile et `Ctrl+z` nous permet de dépiler ces actions. Lorsque l'on navigue sur le web, notre navigateur enregistre dans une liste tous les sites visités et lorsque l'on clique sur le bouton `Retour en arrière`, on dépile les pages visitées.
Dans le prochain chapitre, on verra également une application des piles dans la programmation récursive.
Les piles sont aussi très utilisées en architecture CPU, dans les calculatrices, ...



On peut s'exercer à la logique LIFO en jouant au jeu « Octaves Flush » d'après une idée de Donald Knuth, programmé en JavaScript par Alain Busser professeur de Mathématiques et NSI à La Réunion :
https://alainbusser.frama.io/NSI-IREMI-974/stacksortable.html

### 3.2.1 Interface

Voici l'interface minimal d'une pile :

| Fonction | Description |
|---|---|
| `Pile` | crée une pile vide |
| `est_vide` | indique si la pile est vide |
| `empiler` | insère un élément en haut de la pile |
| `depiler` | renvoie la valeur de l'élément en haut de la pile ET le supprime de la pile |

**exercice 3.4** Dans quel état se trouve la pile après chacune de ces opérations :

```
>>> ma_pile = Pile()
>>> ma_pile.empiler(1)
>>> ma_pile.empiler(2)
>>> ma_pile.depiler()
>>> ma_pile.empiler(3)
>>> ma_pile.empiler(4)
>>> ma_pile.empiler(5)
>>> ma_pile.depiler()
>>> ma_pile.depiler()
```

### 3.2.2 Implémentation

Plusieurs implémentations sont possibles. La plus simple est d'utiliser le tableau dynamique de Python (le type `List`) avec uniquement les méthodes `append()` et `pop()`.

```
class Pile :
    def __init__(self):
        self.data = []

    def est_vide(self):
        return self.data == []

    def empiler(self, valeur):
        self.data.append(valeur)

    def depiler(self):
        assert self.data != [], "Pile Vide !"
        return self.data.pop()
```

**exercice 3.5** Voici un interface étendue d'une pile. A vous de le programmer (si vous le souhaitez, vous pouvez aussi utilisée les fonctions spéciales, cf chapitre précédent)

| Fonction | Description |
|---|---|
| `Pile` | crée une pile vide |
| `est_vide` | indique si la pile est vide |
| `empiler` | insère un élément en haut de la pile |
| `depiler` | renvoie la valeur de l'élément en haut de la pile ET le supprime de la pile |
| `sommet` | renvoie la valeur qui est sommet de la pile |
| `afficher` | affiche dans sous la forme d'une pile les éléments (à la verticale) |
| `longueur` | renvoie la longueur de la pile |

■

**exercice 3.6** Rajouter les fonctions suivantes à l'interface de la pile.

| Fonction | Description |
|---|---|
| `echanger` | permute les deux éléments au sommet de la pile sauf si la pile à moins de deux éléments |
| `fond` | supprime et renvoie l'élément situé au bas de la pile |
| `renverser` | inverse l'ordre de la pile (sans utiliser la méthode `reverse()` des tableaux dynamiques Python (type `List`)) |

■

**exercice 3.7 — Palindrome.** Un palindrome est un mot qui peut se lire de la même façon de gauche à droite comme de droite à gauche comme par exemple **ICI**, **KAYAK**, **LAVAL** ou **ANNA**.
Utiliser la structure `Pile` pour vérifier si un mot proposer par l'utilisateur est un palindrome.
On pourra utiliser le fichier `TAD.py` contenant une implémentation de tous les types abstraits de données étudiés à l'aide de l'instruction :

```
from TAD import Pile
```

■

**exercice 3.8 — Parenthèses.** Une chaîne de caractères `chaine` est dite bien parenthésée lorsque l'une de ces conditions est vérifiée :

- `chaine` est vide ;
- `chaine` est de la forme `(U)`, `{U}` ou `[U]` où `U` est une chaîne de caractères bien parenthésée ;
- `chaine` est de la forme `UV` où `U` et `V` sont des chaînes de caractères bien parenthésées.

Par exemple, `'{ [ ( ) ( ) ] }'` est une expression bien parenthésée tandis que `'( [ ) ( ) ]'`ne l'est pas.
Écrire une fonction `parentheses` qui prend pour argument une chaîne de caractères et renvoie `True` lorsque celle-ci est bien parenthésée, et `False` sinon.
On pourra d'abord commencer par un seul type de parenthèse () puis on pourra généraliser la démarche.
On pourra utiliser le fichier `TAD.py` contenant une implémentation de tous les types abstraits de données étudiés.
■

Rmq En s'inspirant du dernier exercice on pourrait vérifier si un fichier écrit en HTML est convenablement balisé.

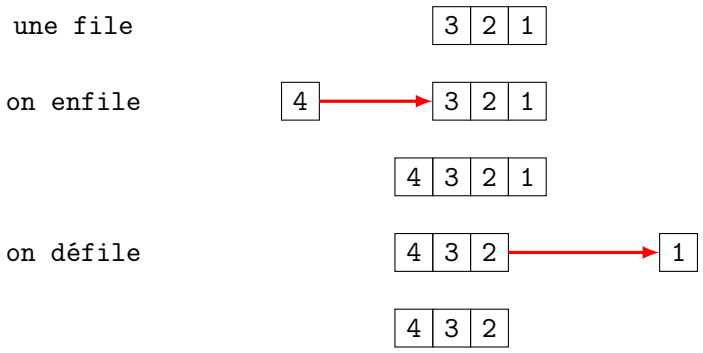
**exercice 3.9 — **.** Implémenter la structure `Pile` à l'aide de la structure `Cellule` utilisée dans la première partie de ce cours.
On pourra utiliser le fichier `TAD.py` contenant une implémentation de tous les types abstraits de données étudiés.
■

## 3.3 Les files

Une **file** est une liste qui se gère comme une file d'attente :
- si on rajoute une nouvelle personne, elle se situe à la fin de file d'attente,
- si on enlève une personne, ce sera uniquement la première personne de la file.

On dit que l'on applique la méthode **FIFO** (First In First Out), autrement dit le premier arrivé sera le premier parti.

On trouve à nouveau un grand nombre d'applications à l'utilisation des files. La gestion de stocks, les mémoires tampons ou la mémorisation de données à traiter séquentiellement.
On pourra voir sur le lien suivant que le choix entre la méthode **LIFO** ou la méthode **FIFO** est une vraie question dans la gestion du stock :
https://www.mecalux.fr/blog/methode-lifo-fifo-peps




### 3.3.1 Interface

Voici l'interface minimal d'une file :

| Fonction | Description |
|---|---|
| `File` | crée une file vide |
| `est_vide` | indique si la file est vide |
| `enfiler` | ajoute un élément en bout de file |
| `defiler` | renvoie la valeur de l'élément en début de la file ET le supprime de la file |

**exercice 3.10** Dans quel état se trouve la file après chacune de ces opérations :

```
>>> ma_file = File()
>>> ma_file.enfiler(1)
>>> ma_file.enfiler(2)
>>> ma_file.defiler()
>>> ma_file.enfiler(3)
>>> ma_file.enfiler(4)
>>> ma_file.enfiler(5)
>>> ma_file.defiler()
>>> ma_file.defiler()
```

■

### 3.3.2 Implémentation

Plusieurs implémentations sont possibles. La plus simple est d'utiliser le tableau dynamique de Python (le type `List`) avec uniquement les méthodes `append()` pour enfiler un élément et `pop(0)` pour supprimer le premier élément (et donc défiler).

```
class File :
    def __init__(self):
        self.data = []

    def est_vide(self):
        return self.data == []

    def enfiler(self, valeur):
        self.data.append(valeur)

    def defiler(self):
        assert self.data != [], "file Vide !"
        return self.data.pop(0))
```

**exercice 3.11** Voici un interface étendue d'une file. A vous de le programmer (si vous le souhaitez, vous pouvez aussi utilisée les fonctions spéciales, cf chapitre précédent)

| Fonction | Description |
|---|---|
| `File` | crée une file vide |
| `est_vide` | indique si la file est vide |
| `enfiler` | ajoute un élément en bout de file |
| `defiler` | renvoie la valeur de l'élément en début de la file ET le supprime de la file |
| `tete` | renvoie la valeur qui est en tête de la file |
| `afficher` | affiche sous la forme d'une file les éléments (à l'horizontale) |
| `longueur` | renvoie la longueur de la file |
| `renverser` | inverse l'ordre de la pile (sans utiliser la méthode `reverse()` des tableaux dynamiques Python (type `List`)) |

■

**exercice 3.12 — Pair ou Impair.** On dispose d'une file contenant des nombres entiers. Écrire une fonction qui renvoie une file contenant les nombres pairs et une file contenant les nombres impairs.
On pourra utiliser le fichier `TAD.py` contenant une implémentation de tous les types abstraits de données étudiés.
■

**exercice 3.13 — Deux piles **.** On peut implémenter la structure `File` à l'aide de deux `Piles`. L'idée est la suivante :

- on crée une pile d'entrée et une pile de sortie
- quand on veut enfiler, on empile sur la pile d'entrée
- quand on veut défiler, on dépile sur la pile de sortie
- si celle-ci est vide, on dépile entièrement la pile d'entrée dans la pile de sortie.

Implémenter ainsi la structure `File`.
On pourra utiliser le fichier `TAD.py` contenant une implémentation de tous les types abstraits de données étudiés.
■

**exercice 3.14 — La suite look'n'say.** La suite **look'n'say** est une suite inventée par le mathématicien anglais John Conway. Elle consiste à « lire ce que l'on voit » dans une série de chiffres.
Par exemple pour la série **1211** :

- je vois un '1', donc j'écris **11**

- je vois ensuite un '2', donc j'écris **12**
- je vois enfin deux '1', donc j'écris **21**

J'obtiens ainsi une nouvelle série **111221** avec laquelle je vais pouvoir reprendre le même processus.
Écrire une fonction `look_n_say` qui prend en entrée une série de chiffres sous la forme d'une file et un entier $n$ et qui renvoie en sortie le résultat attendue sous la forme d'une file également après $n$ itérations du processus look'n'say.
On pourra utiliser le fichier `TAD.py` contenant une implémentation de tous les types abstraits de données étudiés.
■

## 3.4 Les dictionnaires

Un **dictionnaire** associe des clés à des valeurs sans aucune notion d'ordre, il n'y a donc pas de premier ou de dernier élément.
Chaque clé est unique et à chaque clé on doit n'associer qu'une seule valeur. Par contre il est tout à fait possible d'avoir plusieurs fois la même valeur.

### 3.4.1 Interface

Voici l'interface minimal d'un dictionnaire :

| Fonction | Description |
|---|---|
| `Dico` | crée un dictionnaire vide |
| `est_vide` | indique si le dictionnaire est vide |
| `ajouter` | ajoute un couple clé valeur au dictionnaire |
| `supprimer` | supprime un couple clé valeur du dictionnaire en donnant en argument uniquement la clé |
| `valeur` | renvoie la valeur associée à la clé donnée en argument |
| `cles` | renvoie la liste des clés |

### 3.4.2 Implémentation

Le langage Python fournit déjà le type `dict` qui implémente parfaitement un dictionnaire.

**exercice 3.15 — **.** Implémenter le type structuré **dictionnaire** comme une **liste** de **tuple** `(cle, valeur)`.
On pourra utiliser le fichier `TAD.py` contenant une implémentation de tous les types abstraits de données étudiés.
■

### 3.4.3 Table de hachage

Dans l'exercice précédent vous avez remarqué qu'il peut être fastidieux dans un dictionnaire de retrouver la bonne clé. En effet, on ne sait pas où elle est située et donc on doit effectuer une recherche exhaustive de la liste contenant les tuples `(cle, valeur)`. Pourtant le type `dict` de Python retrouve très rapidement la valeur associée à la clé donnée. Python utilise pour cela une **table de hachage**.
Voici le principe. On souhaite transformer la clé en un indice dans un tableau à l'aide d'une **fonction de hachage**. Par exemple, on va choisir un tableau de taille 5 et pour notre fonction de hachage, nous allons considérer que la clé est une chaîne de caractères. Pour trouver l'indice, nous allons faire la somme de tous les codes des caractères de la clé. Nous allons ensuite regarder le reste de la division par 5 (la taille du tableau). Nous avons alors un indice dans notre tableau associé à notre clé.

```
1 HTAILLE = 5 # taille de la table de hachage
2
3 def hachage(cle):
4     code = 0
5     for car in cle :
6         code = code + ord(car)
7     return code % HTAILLE
```

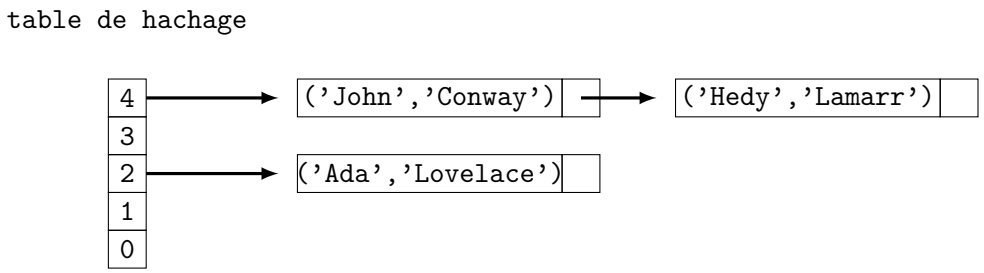
Ainsi, pour notre couple clé/valeur `('Ada','Lovelace')`, nous avons dans la clé `'Ada'` :

- `'A'` qui a pour code 65
- `'d'` qui a pour code 100
- `'a'` qui a pour code 97

Le total vaut $65+100+97=262$ et son reste par la division par 5 sera **2**.
Nous placerons alors notre couple à l'index 2 de notre table de hachage.
Inversement pour retrouver notre couple, la fonction de hachage nous dira d'aller retrouver notre couple à l'index 2 de la table de hachage.

Mais la taille de la table de hachage étant limitée, il y a un risque de **collision**. C'est ce que l'on obtient avec nos deux autres couples. Pour le couple `('John','Conway')`, la clé `'John'` est associée par la fonction de hachage à l'indice 4. De même avec le couple `('Hedy','Lamarr')`, dont la clé `'Hedy'` est associée elle aussi à l'indice 4 ! Comment faire pour les distinguer !

Une solution consiste à stocker chaque couple `(cle, valeur)` ayant le même indice dans une liste qui sera stockée dans la table de hachage.




Dans notre cas, on constate qu'en moyenne le temps de recherche d'un couple connaissant la clé est divisé par 5 (la taille de la table de hachage) pour une taille mémoire à peu près équivalente.

Vous trouverez dans le fichier `6 hachage.py` un exemple d'implémentation d'un dictionnaire simplifié avec table de hachage.