Cours 24 – Codage – Partie 2
Représentation en mémoire des conteneurs

Dans ce second cours, nous abordons les types « conteneurs », c'est-à-dire les types ou classes : *liste, vecteur (array* de dimension 1*)* et *dictionnaire*, en expliquant l'intérêt en termes de complexité pour des opérations de base : ajout/suppression d'élément, et recherche d'un élément. En pratique, seuls les dictionnaires sont au programme, nous passerons donc rapidement sur les deux autres types.

Source principale du cours : cours de Mickaël Péchaud, https://mpechaud.fr/scripts/donnees/listestableaux.html

A / Le type liste (hors programme)

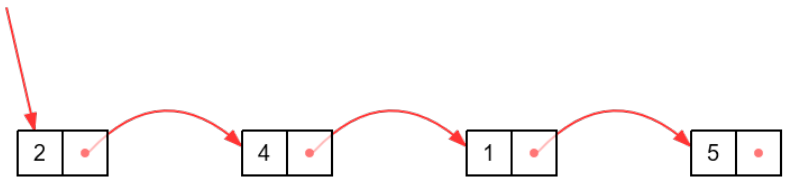
Le type abstrait *liste* n'a pas à être stocké de façon contigüe dans la RAM, ce qui rend cette structure plus « souple » que les tableaux, d'autre part les données contenues n'ont pas nécessairement à être de même type (c'est-à-dire que le découpage entre les données ne se fait pas sur un critère de taille constante, connue *a priori*).

Une liste est implémentée au moyen d'un chaînage entre cellules. Chaque cellule contient une seule donnée, et les cellules n'ont pas à être stockées de manière jointive dans la RAM. Le principe du chaînage entre cellules est expliqué de manière synthétique ci-dessous :

- **Liste simplement chaînée**

  Une cellule est constituée de 2 parties :
  - La donnée elle-même (par exemple, ci-dessous, des données de type *int*, mais les données n'ont pas nécessairement à être toutes de même type).
  - Et un **pointeur** contenant le *type* et l'*adresse* de la 1ère case mémoire de la cellule suivante.

  Exemple :



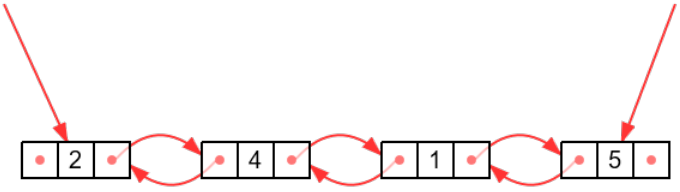
Source : M. Péchaud

  ➔ Une liste est donc une succession de cellules disjointes, chacune pointant vers la suivante, avec :
  - Lorsqu'on appelle la variable *Liste* ainsi définie, le pointeur dirige vers la 1ère cellule
  - Et la dernière cellule ne pointe vers rien (pointeur nul), ce qui indique la fin de liste.

- **Liste doublement chaînée**

  Le principe est exactement le même, mais une cellule est constituée de 3 parties :
  - La donnée
  - Un **pointeur** vers la cellule précédente, et un pointeur vers la cellule suivante.

  Exemple :



Source : M. Péchaud

Inconvénient : la liste doublement chaînée a un poids mémoire légèrement supérieur (deux pointeurs à stocker, au lieu d'un pour la liste simplement chaînée).

Avantage :

		Simplement Chaînée	Doublement Chaînée
Ajout/suppression d'un élément …	… au début	O(1)	O(1)
	… en fin de liste	O(N)	O(1)
Accès à la valeur d'un élément …	… au début	O(1)	O(1)
	… à la fin	O(1)	O(1)
	… au milieu	O(N)	O(N)

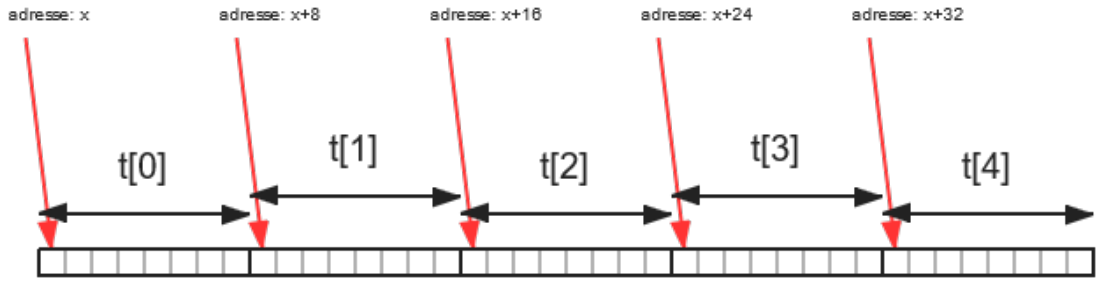
Cet outil est très adapté pour décrire des piles et files.

En *Python* : ce type d'implémentation est utilisé pour les deque (pas pour les type *list* !)

B / Le type vecteur (*array 1D*)

Le type vecteur (utilisé, entre autres, par *Numpy*) doit être stocké de façon contigüe dans la RAM, et les données contenues doivent nécessairement être de même type (c'est donc un type plus contraint que le type *liste*).

Exemple : Prenons l'exemple d'un tableau de *N* entiers codés sur 8 bits



Source : M. Péchaud

Avantage (par rapport à la liste vue en **A**) : le temps d'accès à une valeur est O(1).

En effet, pour accéder à l'élément `t[i]`, il faut lire à l'adresse x + 8*i (si on les données sont en 8 bits).

Inconvénient : pour pouvoir allouer précisément l'espace mémoire dans une zone de « RAM vierge » suffisante, il faut déclarer la taille du vecteur qui est fixée. Par exemple, pour un vecteur de 100 données de 8 bits, il faut 800 bits (ou 100 octets) d'espace contigü.
Il est donc difficile d'ajouter un élément ! Pour ajouter un élément, il faut trouver un nouvel espace contigu vierge, faire une copie terme à terme de l'ancien tableau, et libérer l'espace mémoire utilisé par l'ancien tableau.

Bilan :

		Liste doublement Chaînée	Vecteur (*array* 1D)
Ajout/suppression d'un élément (début, fin ou milieu)		O(1)	O(N)
Accès à la valeur d'un élément …	… à une extrémité	O(1)	O(1)
	… au milieu	O(N)	O(1)

C / Remarque sur le type *list* en *Python* (hors programme)

En *Python*, le choix a été fait d'utiliser la structure de **tableau dynamique** pour décrire les objets de type *List*.

Explication simplifiée de l'implémentation :

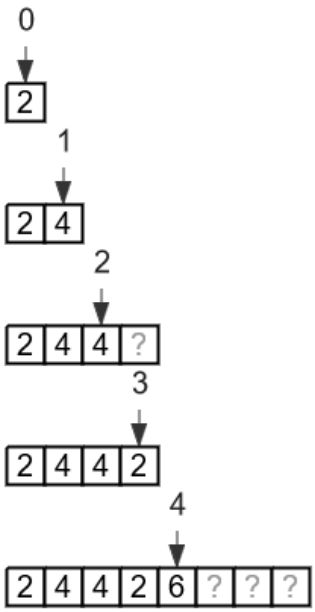
- On se fixe une capacité initiale c ($c \in \mathbb{N}^*$) et une valeur réelle $a > 1$ (en *Python*, par défaut $a = 1.125$)
- On initialise un vecteur (tableau 1D) de c élément, sur un espace contigü en mémoire, et la variable pointe vers l'adresse de la 1ère valeur.

- Lorsqu'on veut ajouter un nouvel élément (à droite !)
  - Si l'élément « tient » dans la capacité maximale du tableau dynamique, on l'ajoute à la suite des éléments déjà contenus (c'est la méthode `.append`), et le **taux de remplissage** augmente. Coût O(1).

    Remarque : pour insérer à gauche ou au milieu (méthode `.insert`), il faut faire une copie comme avec un vecteur normal, coût O(*N*).

  - Si l'élément dépasse la capacité du tableau, on déclare un nouveau tableau de capacité $c' = c \times a$. On fait alors une copie du tableau initial, coût O(*N*), et on continue avec un taux de remplissage plus faible (on se laisse de la « marge »).

Exemple : Illustration d'insertions successives avec une capacité initiale $c = 1$ et un coefficient $a = 2$:



Source : M. Péchaud

Intérêt : en moyenne, la réallocation d'espace mémoire se fait de manière relativement peu fréquente, et on peut montrer que l'ajout d'un élément à droite se fait en temps $O(1)$ (bien sûr, ce coût est en *O*(*N*) dans le pire des cas).

Bilan :

		Liste doublement Chaînée	Vecteur (*array 1D*)	Tableau dynamique
Ajout/suppression d'un élément …	… au début	O(1)	O(N)	O(N)
	… en fin de liste	O(1)	O(N)	O(1)*
Accès à la valeur d'un élément …	… à l'extrémité (début ou fin)	O(1)	O(1)	O(1)
	… au milieu	O(N)	O(1)	O(1)

* en moyenne

D / Le type dictionnaire

Un dictionnaire permet de représenter une **application surjective** entre deux ensembles finis :

- **Application :** chaque élément de l'ensemble de départ, c'est-à-dire chaque clé, n'a qu'une seule image
- **Surjective :** chaque élément de l'ensemble d'arrivée a au moins un antécédent (clé)

  → il en résulte que, pour un dictionnaire `Dict` : == ssi application bijective (valeurs uniques)

```
len(set(Dict.keys)) == len(Dict.keys()) <= len(set(Dict.items()))
```

Remarque : en anglais, *set* signifie ensemble, le type ensemble `set` est hors programme CPGE, mais notez simplement ici qu'on se sert de `set(Liste)` (qui fonctionne en fait pour tout itérable) qui renvoie la liste triée des éléments singuliers, éliminant ainsi tous les doublons.

Exemple :
```
print(set([1,3,5,4,1,5,1,3,1]))     # affiche {1, 3, 4, 5}
```

Il est également important de rappeler qu'un dictionnaire n'est pas ordonné

Exemple :
```
dico = {2:"a", "cle":10.0, -4.17:5}   # les clés (comme les valeurs) n'ont
                                      # pas forcément à être toutes de même type
dico.keys()[0]  # Imaginons qu'on veuille récupérer la 1ère clé : on a alors l'erreur
```
```
TypeError: 'dict_keys' object is not subscriptable
```

Soit le code suivant :

```
import random as rd
import time

def fonct(N):
    L = []  # liste vide
    D = {}  # dictionnaire vide
    for i in range(N) :  # on va générer un dico ayant autant de clés que la liste
        Valeur = rd.randint(0,N)  # on pourra tirer plusieurs fois la même valeur
        L.append(Valeur)   # il peut y avoir plusieurs fois la même valeur dans une liste
        cle_aleatoire = 1e6*rd.random()  # nombre aléatoire entre 0 et 1 million (on est
sûr
                    # de ne pas tirer 2x le même, il doit y avoir unicité de chaque clé)
        D[cle_aleatoire] = Valeur   # 2 clés ≠ peuvent être associées à la même valeur
    return L, D


def comparaison(N):   # on veut comparer le temps de recherche dans une liste VS un dico
    Ltest, Dtest = fonct(N)   # on génère un dico et une liste aléatoires de longueur N

    tic = time.perf_counter()
    "x" in Ltest   # cherche la valeur "x" dans Ltest (balaie forcément tout sans trouver)
    tac = time.perf_counter()
    print("Recherche liste :", tac - tic)   # Δt écoulé pour cette recherche

    tic = time.perf_counter()
    "x" in Dtest  # on cherche la valeur "x" dans les clés de Dtest (même chose)
    tac = time.perf_counter()
    print("Recherche dico :", tac-tic)    # même chose
```

```
>>> comparaison(1000)
Recherche liste : 1.61e-5
Recherche dico : 5.00e-7
```

```
>>> comparaison(10000)
Recherche liste : 1.55e-4
Recherche dico : 5.00e-7
```

```
>>> comparaison(100000)
Recherche liste : 1.66e-3
Recherche dico : 7.00e-7
```

→ On constate que le temps de recherche est O(*N*) dans une liste (cohérent) et O(1) pour un dictionnaire

Remarque : si la liste était triée, on aurait pu abaisser le coût de calcul par une recherche dichotomique en $O(\log_2(N))$
Pour rappel, le coût pour trier la liste est en O(N log(N)) > N

→ Il est donc (évidemment) plus coûteux de trier la liste puis rechercher un élément, que de chercher directement l'élément dans la liste non triée. Cela peut être faux si on a un grand nombre de fois des recherches à faire dans la même liste.

Intérêt :
Pour certains algorithmes que nous avons vu ensemble, nous voulons tenir à jour un ensemble des cas déjà rencontrés. **Exemple** : ensemble des sommets déjà visités pour les algorithmes de parcours de **graphe**, ensemble des sous-problèmes déjà traités (pour mémoïser la solution en cas de chevauchement) pour un problème traité en **programmation dynamique** (cf. cours de PT). Dans ces cas, l'utilisation d'un dictionnaire plutôt qu'une liste donne des temps d'exécutions nettement plus courts.

D.1 / La table de hachage

Focalisons-nous d'abord sur la manière dont est stocké un ensemble (type *set*), ou la liste des clés d'un dictionnaire (sans encore parler des valeurs associées). Pour rappel, quand on fait une recherche dans un dictionnaire, c'est bien sur l'ensemble des clés que se fait la recherche, pas sur les valeurs associées. Par exemple, le test :

```
X in dico      # renvoie True ssi la variable X est une des clés du dictionnaire
```

Notations : soit U (appelé univers) l'ensemble fini des valeurs pouvant être prises, et un sous-ensemble de U, fini de valeurs, nommé E, que l'on souhaite décrire. On veut ici optimiser la recherche, c'est-à-dire déterminer si un élément

de U appartient ou non à E.

Idée générale : nous allons utiliser une structure de **tableau dynamique** (voir **C**), mais plutôt que d'insérer un élément à droite, nous allons l'insérer à un index précis.

soit $h:\begin{cases} U \to \mathbb{N} \\ e \to h(e) \end{cases}$ appelée fonction de hachage

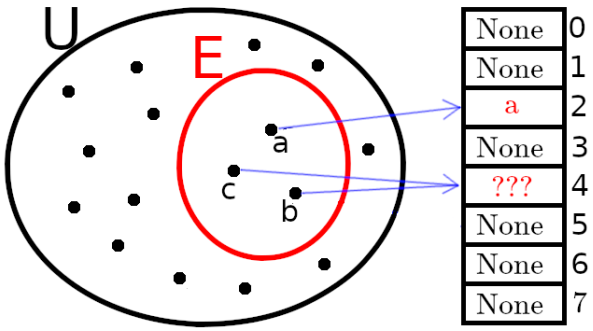
L'idée naïve est d'insérer l'élément e à l'index $h(e)$ dans le tableau dynamique. Ceci facilitera alors grandement la recherche, puisque pour déterminer si une valeur e de U appartient à E, il suffira de vérifier si la « case » (on parle rigoureusement d'**alvéole** pour une table de hachage) à l'index $h(e)$ est vide ou non.

Remarque 1 : on ne peut pas définir h sur des objets mutables (comme des listes, vecteurs, dictionnaires, …). Les clés d'un dico ne peuvent donc pas être mutables.

```
>>> dic = { 1:[2,5], 3:[5,6,7]}     # pas de souci, les valeurs peuvent être des listes
>>> dic = { [2,5]:1, [5,6,7]:3}    # Error: unhashable type: 'list'
```

Remarque 2 : le cardinal $\mathrm{Card}(U)$, càd le nombre de valeurs pouvant être prises par une variable (qui peut être de type *int, float, str…*) est gigantesque ! Pour que le codage de $h(e)$ ne soit pas trop long, on doit renoncer au fait que h soit bijective , en revanche on s'assure qu'elle soit surjective

Remarque 3 : un tableau dynamique a une taille finie *C* (sa capacité, que l'on peut modifier selon le remplissage voir **C**). Or $\mathrm{Card}(h(U))$ peut être très grand, donc il faudrait une capacité *C* gigantesque. On stocke donc plutôt la donnée e à l'index $h(e)\%C$. Notons que, h étant surjective, $h\%C$ l'est également.



e =	"a"	"b"	"c"	…
h(e)	9	4	11	…
h(e)%7	2	**4**	**4**	…

collision

Source : M. Péchaud

Conséquence : deux valeurs $(e_1, e_2) \in U^2$ peuvent avoir la même image $h(e_i)\%C$, et on devrait donc les écrire sur la même case mémoire du tableau dynamique (l'une écraserait donc l'autre). On dit alors qu'il y a collision.

➔

➔ Pour minimiser le risque de collision, on choisit une fonction h homogène càd telle que les valeurs (entières) des images sont toutes équiprobables.

Exemple : une fonction de hachage `hash()` est déjà définie nativement sur *Python*.

```
>>> hash(2)
2

>>> hash("a")  # type str
262945134970056

>>> hash(7.114034274631611420858 2)    # type float
262945134970056

>>> hash(262945134970056)    # type int
262945134970056
```

Gestion des collisions : quand on fait une insertion d'une valeur e, on vérifie si l'**alvéole** $h(e)\%C$ du tableau dynamique est vide. Si c'est le cas, on fait l'insertion normalement. Si elle est pleine, on peut par exemple :

- Débuter une liste chaînée des différentes valeurs stockées à partir de cette case (puisque les listes chaînées n'ont pas à être stockées de façon contigüe dans la RAM)
- Ou bien débuter un **sondage**, c'est-à-dire rechercher **à partir de l'alvéole** $h(e)\%C$ la prochaine alvéole vide à partir de cette alvéole. C'est l'option qui a été choisie en *Python*.

  → il est donc primordial d'écrire la **valeur** de e dans l'alvéole (et pas seulement « Vrai »).

Conséquence sur la recherche d'élément : l'algorithme de recherche d'un élément e dans une table de hachage de capacité *C* est

- `index =` $h(e)\%C$
- `tant que Table_hachage[index] != e :`
  - `index += 1`

Si la fonction de hachage est homogène et si la capacité de la table est adaptée, la probabilité de collision est faible et le sondage (bien que probable) se fait sur un petit nombre de décalages, et la recherche a en moyenne un coût O(1).

Bilan : (rappel : pour pouvoir décrire un ensemble par une table de hachage, les éléments doivent tous être uniques)

		Vecteur (*array 1D*)	Tableau dynamique	Table Hachage
Ajout/suppression d'un élément …	… en fin de liste	O(N)	O(1)*	O(N)
	… n'importe où ailleurs	O(N)	O(1)*	O(1)*
Accès à la valeur d'un élément d'index quelconque		O(1)	O(1)	O(1)
Recherche d'un élément dans la « liste » des valeurs		O(N)	O(N)	O(1)*

* : En moyenne

D.2 / Implémentation des dictionnaires en *Python* (hors programme)

Le choix retenu pour gérer les dictionnaires en *Python* est de tenir à jours simultanément deux tables. Pour chaque association $(\text{clé} \to \text{valeur})$, on tient à jour une table de hachage appelée **table des indices**. Mais conjointement, on tient à jours une seconde table, dite **table des entrées** qui au même index contient la valeur associée.

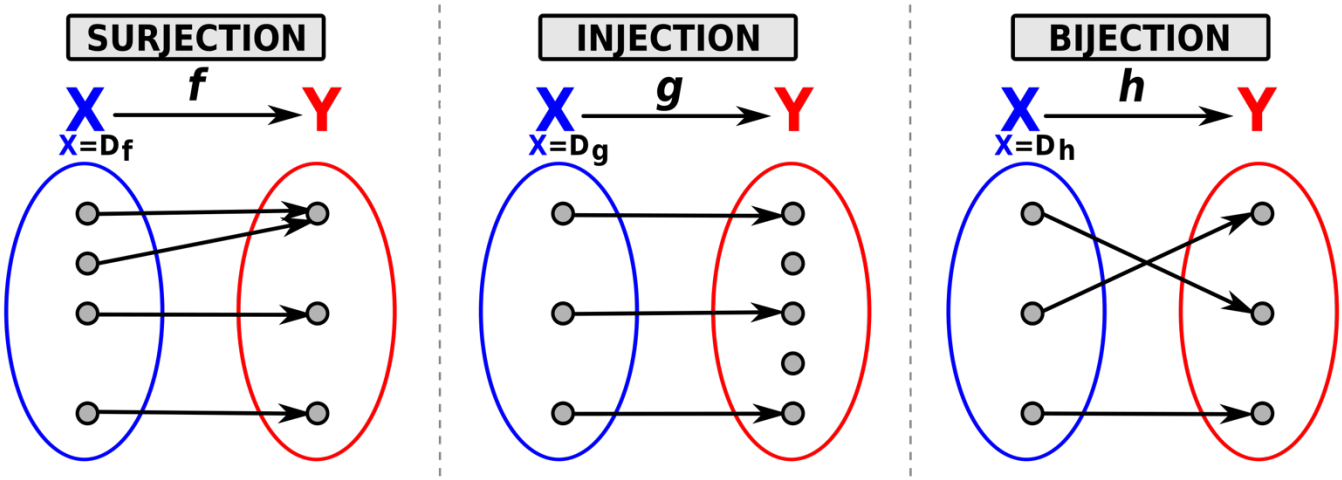
Illustration :
`i = hash(clé)` donne un index.

Après la gestion des collisions potentielles, et donc un potentiel décalage :

`table_des_indices[hash(clé) + decalage]`
contient la `clé` (ce qui permet de détecter les éventuelles collisions). Et au même index :

`table_des_entrees[hash(clé) + decalage]`
contient la `valeur` que l'on souhaite associer à `clé`

Remarque : on voit bien que cette structure est adaptée pour trouver la valeur associée à une clé : $(\text{clé} \to \text{valeur})$ et pas du tout à remonter d'une valeur à la clé correspondante $(\text{valeur} \underset{\times}{\to} \text{clé})$, cohérent avec le caractère non injectif d'un dictionnaire et de la fonction `h()`.


SURJECTION
f
X
Y
X=Df
INJECTION
g
X
Y
X=Dg
BIJECTION
h
X
Y
X=Dh