VII

Dictionnaires 1 : intérêt, structure détaillée, et table de hachage

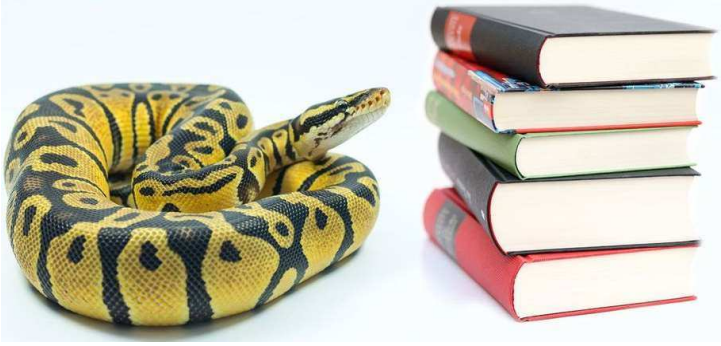


FIGURE VII.1 – Comment est implémenté un dictionnaire en python ?

PLAN DU CHAPITRE

I Pourquoi une nouvelle structure de données ?

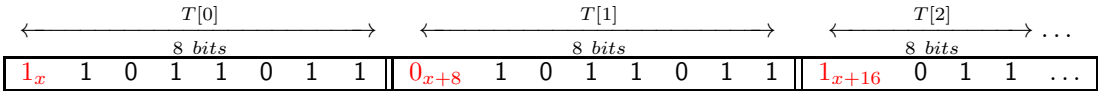
I.1 Préliminaire : complexité des primitives des tableaux, listes (simplement) chainées, et tableaux dynamiques (ou listes en python).

Les ensembles de données en informatique sont généralement stockées dans des structures abstraites, et organisées selon des règles permettant de facilement **rechercher, lire, insérer, ou supprimer** une donnée. La qualité objective d'une telle structure dépend de son aptitude à permettre de réaliser ces opérations **avec une complexité la plus faible possible**.

Nous évoquerons ici les 3 structures de données classiques que sont les tableaux, les listes chaînées, et les tableaux dynamiques. Nous en détaillerons les principes de fonctionnement dans le but de comparer leurs performances dans les opérations de base.

■ Les tableaux

Un tableau est un ensemble de données organisées en mémoire comme une bloc de **cellules contigües**. Si l'on suppose devoir stocker par exemple n entiers naturels chacun codé sur 8 bits ($\in [0, 255]$) dans un tableau nommé T, cette structure de données peut être représentée schématiquement de la façon suivante :



Ainsi, si l'adresse de début du tableau en mémoire est x, alors les intervalles d'adresses de stockage des entiers enregistrés dans ce tableau sont $[x, x+7]$ pour $T[0]$, $[x+8, x+15]$ pour $T[1]$, $[x+16, x+23]$ pour $T[2]$, etc....

Les tableaux sont des structures **non natives en python**, mais disponibles dans le module `numpy`. La déclaration d'un tableau se fait à l'aide de la commande `array` ; par exemple le tableau unidimensionnel des entiers 3,5,8,9 codés par exemple sur 8 bits (après chargement du module `numpy`) s'écrit :

```
>>> import numpy as np
>>>T=np.array([3,5,8,9],dtype=np.int8)
```

Examinons maintenant la complexité des opérations élémentaires pour cette architecture de stockage :

- RECHERCHER : sans équivoque ! ! !..... Cette opération nécessite de balayer la structure de données en comparant l'élément lu avec l'élément cherché ; donc la complexité dans le pire des cas est linéaire :

$$\boxed{C_{tab}(\text{Rechercher}) = \mathcal{O}(n)}$$

- LIRE : l'adresse x du tableau est connue puisque son nom T pointe directement à cette position en mémoire ; ainsi l'accès à l'élément `T[i]` est immédiat (lecture) puisque ce dernier est stocké à l'adresse $x + \texttt{prof} \times i$, avec `prof` la profondeur en bits déclarée pour le stockage des entiers :
  - `dtype=np.int8` pour un stockage sur 8 bits
  - `dtype=np.int16` pour un stockage sur 16 bits
  - `dtype=np.int32` pour un stockage sur 32 bits
  - `dtype=np.int64` pour un stockage sur 64 bits

  La complexité de cette opération est donc en temps constant :

$$\boxed{C_{tab}(\text{lire}) = \mathcal{O}(1)}$$

.

- INSÉRER : les choses se compliquent ici pour les tableaux ; leur taille est en effet déclarée au moment de leur création, donc l'ajout d'une élément à une position donnée va forcément nécessiter la déclaration d'un nouveau tableau de dimension augmentée, puis la recopie élément par élément dans ce nouveau bloc mémoire de l'ensemble des n données du tableau originel.

  La complexité de cette opération est donc en temps linéaire :

  $$\boxed{C_{tab}(\text{insérer}) = \mathcal{O}(n)}$$

- SUPPRIMER : pour supprimer un élément d'un tableau, il faudra procéder, comme pour l'insertion, à la modification de la taille du tableau en procédant à une recopie de tous les éléments conservés en les repositionnant à la bonne adresse.

  La complexité de cette opération est donc en temps linéaire :

  $$\boxed{C_{tab}(\text{supprimer}) = \mathcal{O}(n)}$$

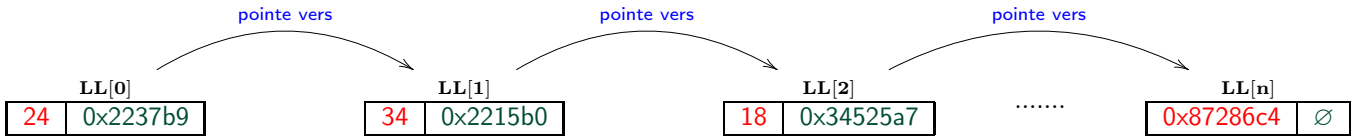
Remarque I-1: TYPE DES DONNÉES D'UN TABLEAU `numpy`

Les tableaux `numpy` ne peuvent contenir que des données de même type.

■ Les listes chaînées (ou *Linked Lists*)

Une liste dite "chaînée" est une structure de données constituée d'une séquence dont chaque élément, appelé aussi *noeud* ou *maillon*, comporte :
- une **valeur** (l'information stockée).
- un **pointeur** contenant l'**adresse mémoire** de l'élément suivant de la liste.

NB : le dernier maillon de la liste chainée (appelé *queue*) **possède un pointeur vide** signalant la fin de la liste.



Reprenons ici l'étude de la complexité des opérations élémentaires pour une liste chaînée `LL` :

- RECHERCHER : dans la pratique, l'adresse du premier élément de la liste chaînée est intégrée dans son nom de variable ; ainsi l'accès à l'élément `LL[0]` est immédiat ; en revanche, la recherche d'un élément ultérieur nécessitera la consultation de l'intégralité des adresses en mémoire des éléments situés avant celui-ci ; un balayage est donc nécessaire jusqu'à trouver l'élément recherché.

  La complexité de cette opération est donc en temps linéaire :

  $$\boxed{C_{LL}(\text{rechercher}) = \mathcal{O}(n)}$$

- LIRE : cette opération intervient ici encore à la suite d'une recherche, donc seule, elle est réalisée avec une complexité temporelle en temps constant :

  $$\boxed{C_{LL}(\text{lire}) = \mathcal{O}(1)}$$

- INSÉRER : Cette fois, la complexité va dépendre de la position de la donnée à insérer :
  - Si celle-ci doit être insérée en début de structure, l'opération se fait en temps constant puisque l'adresse est connue par le nom de la variable :

$$\boxed{C_{LL}(\text{insérer en début}) = \mathcal{O}(1)}$$

  - Si en revanche celle-ci doit être insérer en fin de liste, alors l'adresse mémoire de la cellule concernée ne peut être connue qu'en balayant l'ensemble de la liste chainée. L'opération se fait en donc en temps linéaire :

$$\boxed{C_{LL}(\text{insérer en fin}) = \mathcal{O}(n)}$$

- SUPPRIMER :
  pour les mêmes raisons que l'opération d'insertion, on obtient :

$$\boxed{C_{LL}(\text{supprimer en début}) = \mathcal{O}(1)}$$

$$\boxed{C_{LL}(\text{supprimer en fin}) = \mathcal{O}(n)}$$

IMPLÉMENTATION EN PYTHON :
Les listes chaînées ne sont pas natives en python, on procède donc à une déclaration de classe pour les implémenter. On commence par implémenter les noeuds en créant la classe `Node`, puis la liste à proprement parler avec la classe `listechainee`, contenant par exemple deux primitives `listeaffiche` et `listeaffichen` permettant respectivement l'affichage de la totalité de la liste, et l'affichage du $n-1^{\text{ième}}$ élément de la liste :

Listing VII.1 – Implémentation des listes chaînées

```
1  class Node:
2    def __init__(liste, dataval=None):
3      liste.dataval = dataval
4      liste.valsuivant = None
5
6  class listechainee:
7      def __init__(self):
8          self.teteval = None
9      def listeaffiche(self): # pour afficher la totalité de la liste
10         printval = self.teteval
11         listeaffiche=[]
12         while printval is not None:
13             listeaffiche.append(printval.dataval)
14             printval = printval.valsuivant
15         return listeaffiche
16     def listeaffichen(self,n): # pour afficher le n-1 ième élément de la liste
17         compt=0
18         printval = self.teteval
19         while compt!=n:
20             printval = printval.valsuivant
21             compt+=1
22         return printval.dataval
23     def listedernier(self): #pour afficher le dernier élément de la liste
24         printval = self.teteval
25         valprec=printval # on enregistre la tête, utile si elle est seule
26         printval=printval.valsuivant
27         while printval is not None:
28             valprec=printval
29             printval = printval.valsuivant
30         return valprec.dataval
```

On donne ci-dessous le principe d'instanciation d'une nouvelle liste chaînée, ainsi que quelques exemples d'utilisation des primitives implémentées :

```
>>> LL = listechainee() # Déclare la liste chaînée
>>> LL.teteval = Node("alpha") # Affecte la première valeur
>>> val1 = Node("beta") # puis la seconde
>>> val2 = Node("gamma") # et la troisième

# affectation des pointeurs pour le chainage #
>>> LL.teteval.valsuivant = val1 # le premier élément (tëte de liste) pointe vers le second
>>> val1.valsuivant = val2 # le second élément pointe vers le troisième (et dernier)

# Recherche et lecture #
>>> LL.listeaffiche() # Affiche la totalité de la liste chaînée
alpha
beta
gamma
>>> LL.listeaffichen(1) # Affiche le second élément de la liste chaînée
beta
>>> LL.listedernier()
gamma
```

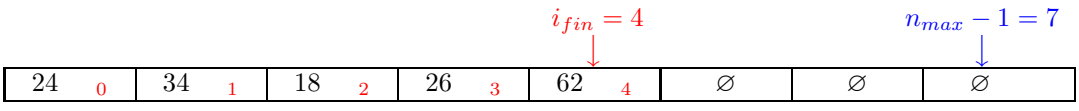
■ Les tableaux dynamiques

Les tableaux présentent l'avantage d'une organisation mémoire simple (les données sont affectées sur des adresses vers lesquelles pointent le nom de variable) ; cela assure notamment une complexité en temps constant $\mathcal{O}(1)$ pour la lecture d'un élément du tableau. Les listes chaînées de leur côté permettent l'ajout ou la suppression d'un élément **en début de liste** en temps constant $\mathcal{O}(1)$, là où les tableaux se contentent d'une complexité linéaire $\mathcal{O}(n)$; elles nécessitent cependant l'emploi des pointeurs "successifs" dont l'usage est un peu laborieux (l'implémentation et l'utilisation de la classe `listechainee` ne sont pas vraiment une sinécure !).

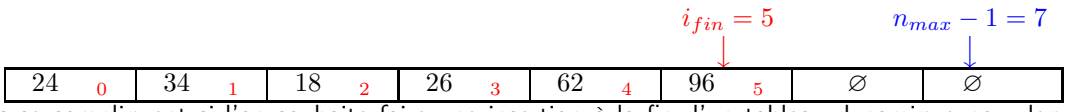
Le tableau dynamique est une structure de données qui permet :

- d'éviter la gestion lourde des pointeurs des listes chaînées
- permettre, comme pour les tableaux, une lecture en temps constant $\mathcal{O}(1)$
- permettre l'ajout et la suppression d'un élément **mais** en temps constant en fin de liste (sous réserve) $\mathcal{O}(1)$ et en $\mathcal{O}(n)$ en début de liste

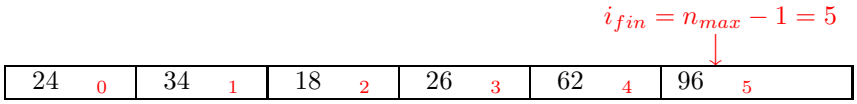
Un tableau dynamique est matérialisé par un tableau classique de taille n_{max} associé à un indice i_{fin} de valeur comprise entre -1 et $n_{max}-1$. Une telle structure est qualifiée de tableau «**dynamique**» en raison de sa capacité à faire évoluer la valeur i_{fin}.
On peut donc représenter schématiquement un tableau dynamique ainsi :



Compte tenu de cette architecture, l'opération de lecture s'effectue en temps constant $\mathcal{O}(1)$; par ailleurs, si $i_{fin} < n_{max} - 1$, alors l'insertion d'un élément en fin de tableau dynamique nécessite simplement l'incrémentation du compteur $i_{fin} \rightarrow i_{fin} + 1$ et l'affectation de la donnée supplémentaire, par exemple ici pour l'entier 96 :



Les choses se compliquent si l'on souhaite faire une insertion à la fin d'un tableau dynamique pour lequel $i_{fin} = n_{max} - 1$:



Il faudra alors rallonger la taille du tableau pour étendre la capacité du tableau dynamique ; ces opérations augmentent considérablement la complexité de l'insertion. Ainsi, lors de la déclaration d'un tableau dynamique, il est arbitrairement choisi une taille de tableau plus importante afin d'éviter de réallocation mémoire.

En pratique, les langages allouent toujours une taille en mémoire augmentée par rapport à la déclaration de l'utilisateur ; ainsi, si les données dans le tableau dynamique occupent une taille $N = i_{fin} + 1$, le langage réserve en mémoire une taille $a \times N$. Le paramètre a dépend de la spécification du langage. A la suite d'ajouts d'éléments, si l'on atteint la taille maximale allouée en mémoire, le langage réalloue une taille $a \times N$ avec la nouvelle valeur de N. On montrera en exercice ci-dessous que cette opération est sensiblement de complexité linéaire $\mathcal{O}(n)$.

On peut résumer ainsi le comportement du langage pour l'ajout d'un élément el dans un tableau dynamiqueTD de paramètre a :

- **Si $N < \lceil a \times N \rceil$ alors faire :**
  - $i_{max} \longleftarrow i_{max} + 1$
  - $TD[i_{max}] \longleftarrow el$
- **sinon faire :**
  - étendre la taille du tableau en mémoire à $a \times (i_{max} + 1)$
  - $i_{max} \longleftarrow i_{max} + 1$
  - $TD[i_{max}] \longleftarrow el$

Exercice de cours: **(I.1) - n° 1.** *On suppose un tableau dynamique contenant un seul élément* 24 *et de paramètre de taille* $a = 2$

1. *Donner la suite des opérations pour ajouter successivement* $34, 18, 26$.

2. *Si l'on suppose que chaque réallocation mémoire se fait en temps proportionnel à la taille du tableau, montrer que la complexité d'un ajout de n éléments dans le cas d'un tableau dynamique est en $\mathcal{O}(n)$ (un peu délicat !).*

IMPLÉMENTATION EN PYTHON : Les listes, très utilisées en Python pour leur souplesse, sont justement des tableaux dynamiques. Le type `list` est natif dans Python et la déclaration d'une liste est élémentaire :

```
>>>import random as rd
>>>L1=[0,1,2,3,4,5,6]
>>>L2=[rd.randint(1,1000) for i in range(10)]
>>>print(L2)
```

$$620, 392, 497, 234, 573, 679, 731, 583, 449, 31$$

```
>>>L=L1+L2
>>>print(L)
```

$$0, 1, 2, 3, 4, 5, 6, 620, 392, 497, 234, 573, 679, 731, 583, 449, 31$$

```
>>>type(L)
list
```

En outre, on peut facilement interroger **la taille mémoire allouée à une liste Python** à l'aide du package `sys` :

```
>>>import sys
>>>L1=[i for i in range(10)]
>>>print(L1)
```

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$$

```
>>>print(sys.getsizeof(L1))
184
>>>L1=L1+[i for i in range(10,20)]
>>>print(L1)
```

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19$$

```
>>>print(sys.getsizeof(L1))
216
```

Remarque I-2: COMPLEXITÉ TEMPORELLE VERSUS COMPLEXITÉ SPATIALE DES LISTES

On constate que Python voit particulièrement large en ce qui conerne le dimensionnement des listes ! C'est donc un gage de bonne complexité temporelle, mais c'est naturellement moins bien pour la complexité spatiale !

Les complexités des opérations élémentaires sont les suivantes :

- RECHERCHER : elle est évidemment identique à celle des tableaux classiques, soit dans le pire des cas :

$$\boxed{C_{tab.\_dyn.}(\text{rechercher}) = \mathcal{O}(n)}$$

.

- LIRE : sans surprise :

$$\boxed{C_{tab.\_dyn.}(\text{lire}) = \mathcal{O}(1)}$$

- INSÉRER : cela dépend de la situation :

$\boxed{C_{tab.\_dyn.}(\text{insérer au début}) = \mathcal{O}(n)}$ et $\boxed{C_{tab.\_dyn.}(\text{insérer en fin}) = \mathcal{O}(1)}$

- SUPPRIMER : cela dépend encore de la situation :

$\boxed{C_{tab.\_dyn.}(\text{insérer au début}) = \mathcal{O}(n)}$ et $\boxed{C_{tab.\_dyn.}(\text{insérer en fin}) = \mathcal{O}(1)}$

I.2 Bilan et recherche d'une structure idéale

Résumons-nous ! Chaque structure de données évoquée ci-dessus présente avantages et inconvénients **du point de vue de la complexité temporelle** des 4 opérations élémentaires *recherche, lecture, insertion, suppression* ; le tableau ci-dessous les compare :

	TABLEAU	LISTE CHAÎNÉE	TABLEAU DYNAMIQUE (LISTES)
Recherche puis lecture au début	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Recherche puis lecture en fin	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
insertion en début	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
insertion en fin	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
suppression en début	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
suppression en fin	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$

La structure des tableaux dynamiques est globalement la meilleure, puisque c'est celle qui permet une complexité en $\mathcal{O}(1)$ pour le maximum d'opérations élémentaires.

CONCLUSION : l'idéal serait de disposer d'un structure de données assurant une complexité en $\mathcal{O}(1)$ pour toutes les opérations évoquées ici, c'est à dire un ensemble de stockage à la fois performant (recherche-lecture) et dynamique (insertion-suppression).

$\Longrightarrow$ Nous allons voir que les dictionnaires répondent parfaitement à cette problématique.

II Les dictionnaires : le meilleur des «trois mondes»

II.1 Une première idée d'implémentation : la table à adressage direct - limitations

Les dictionnaires ont déjà été abordés en cours de première année ; leur principe est d'associer des valeurs à des clés respectives.

Appelons U l'univers des clés avec $U = \{0, 1, \ldots, m-1\}$ où m n'est pas trop grand.

Un première idée d'implémentation d'un dictionnaire serait d'exploiter un simple tableau appelé **table à adressage direct** T dont les indices de position des cases, appelées ici *alvéoles*, sont les clés, et les valeurs stockées à ces indices, les valeurs correspondantes du dictionnaire :

Définition II-1: TABLE À ADRESSAGE DIRECT

Si T est une table à adressage direct alors : $\left[\begin{array}{l} T[k] = None \text{ si la clé } k \text{ n'est associée à aucune valeur} \\ T[k] = v_k \text{ si la clé } k \text{ est associée à la valeur } v_k \end{array}\right.$

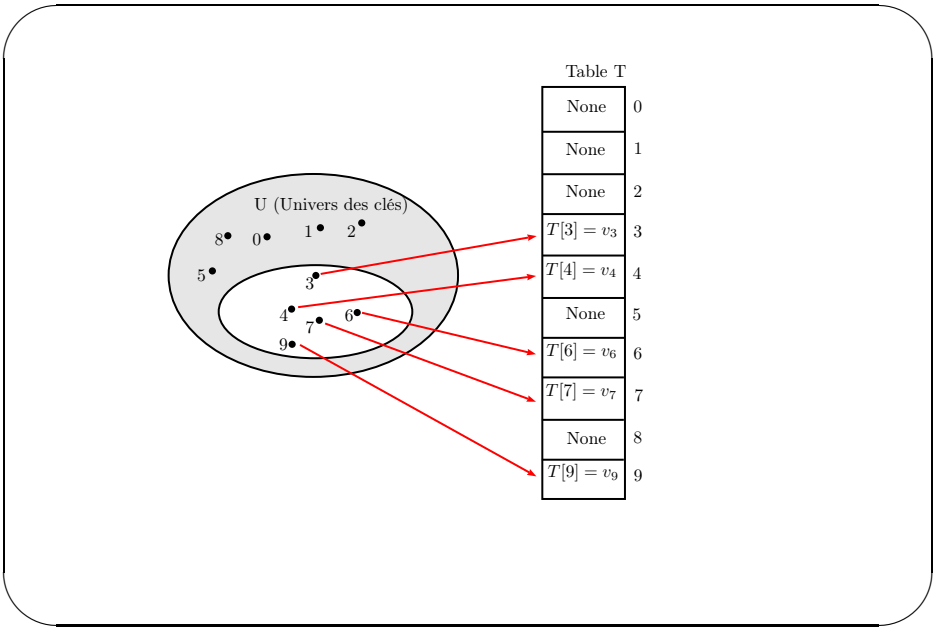


FIGURE VII.2 – Représentation schématique d'une `table à adressage direct` d'implémentation naïve ($m = 10$ ici)

Exercice de cours: **(II.1) - n° 2.** *Proposer l'implémentation en Python des primitives* `recherche-lecture`, `insertion`, `suppression` *pour la table à adressage direct.*

Ici l'Univers des clés se limite à un sous ensemble $\{0 \dots m-1\}$ des entiers, alors cette implémentation n'est ni plus ni moins qu'une structure analogue à une liste de taille m. Ce n'est évidemment pas l'idée du dictionnaire.

Par ailleurs, plusieurs limitations apparaissent :

- **Les clés se limitent à des entiers** ce qui n'est pas très pratique, et en tout cas peu explicite pour pointer vers une valeur. **Une chaine de caractère serait par exemple plus judicieuse**.
- Si l'univers des clés $\mathbb{U}$ est très grand, alors la gestion de la table qui doit être **de la même taille que U peut devenir délicate** (grands nombres, espace mémoire alloué à la table très grande...).
- Le nombre de clés réellement exploitées peut être très faible par rapport à la taille de U. **Une très grande partie de l'espace mémoire alloué à T est donc gaspillé.**

II.2 Table et fonction de hachage - collisions

Un première idée pour pouvoir exploiter des clés de type *chaîne de caractères* est de disposer d'une fonction injective f qui transforme chaque clé chaîne de caractère `ch` en un entier $f(ch) \in [0, m-1]$:

Cette solution naïve est en fait inadaptée car le cardinal de U peut être très élevé, obligeant alors à réserver, comme pour l'adressage direct, un tableau de même taille (donc immense !). Par exemple si l'on retient que les clés sont des mots d'au plus 5 lettres minuscules, alors le cardinal de U est :

$$card(U) = 26^5 + 26^4 + 26^3 + 26^2 + 26 = 12356630$$

CONCLUSION : le problème de la taille de la table d'adressage perdure donc !

On vient de voir qu'en choisissant des chaines de caractères en guise de clés, le cardinal de l'univers des clés est un nombre gigantesque ; **on doit donc renoncer à définir une fonction d'adressage injective** de $\mathbb{U}$ dans $\{0, 1, 2, \dots, card(\mathbb{U}) - 1\}\}$

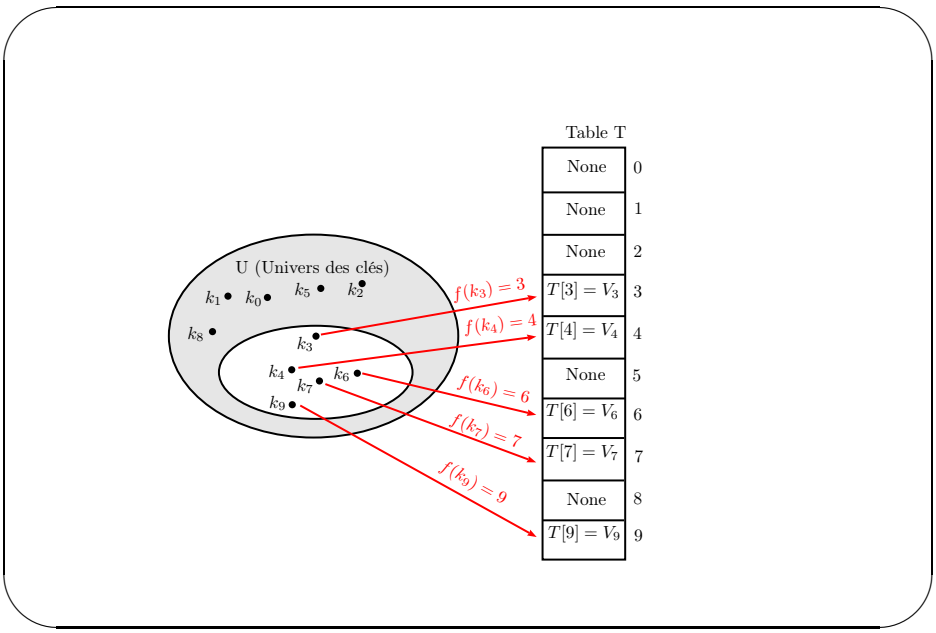


FIGURE VII.3 – Fonction injective de transformation des clés (toujours pour $m = 10$)

La solution est d'introduire une fonction f, non injective donc, qui conduira à une tableau associatif appelé **table de hachage** du dictionnaire. C'est donc une application de $\mathbb{U}$ dans $\{0 \dots m\}$.

Définition II-2: TABLE DE HACHAGE D'UN DICTIONNAIRE

La correspondance entre une clé k et la valeur v_k vers laquelle elle pointe doit être assurée par une **fonction non injective** f. La valeur v_k est alors stockée dans l'alvéole $f(k)$, donc :

$$v_k = T[f(k)]$$

NB : on dit qu'un élément de clé k est hachée, et que $f(k)$ est la valeur de hachage de la clé k.

Cette structure corrige en effets les défauts de l'adressage direct :

- la table de hachage ne contient que les éléments effectivement "hachés", **ce qui réduit considérablement le nombre des indices de la table à gérer**.
- L'occupation mémoire est désormais limitée au strict nécessaire.

PROBLÈME : comme $card(\mathbb{U}) > m$, le risque que deux clés k_i et k_j aient la même valeur de hachage $f(k_i) = f(k_j)$ est très grand ; elles pointeront donc vers la même alvéole. On parle alors de **collisions**, situation qu'il va falloir nécessairement gérer.

Pour bâtir la fonction $f(k)$, on exploite une fonction dite "fonction de hachage" :

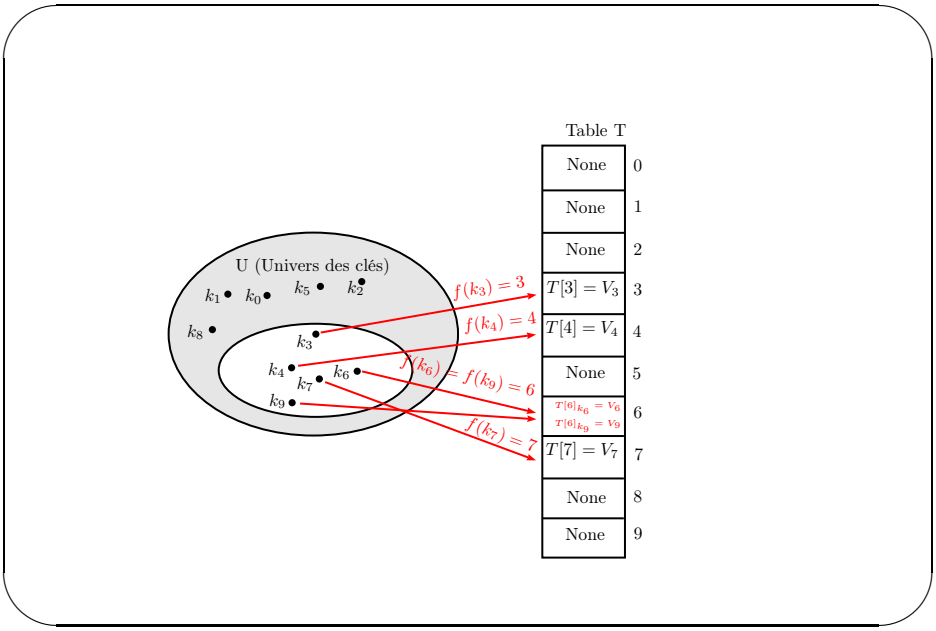


FIGURE VII.4 – Fonction $f(k)$ qui fait correspondre à une clé k une alvéole de la table de hachage. Les clés k_6 et k_9 pointent vers la même alvéole $T[6]$ et **entrent donc en collision**.

Définition II-3: FONCTION DE HACHAGE

La fonction :

$$\begin{array}{rcccl} f & : & \mathbb{U} & \rightarrow & \{0,1,\ldots,m\} \\ & & k & \mapsto & f(k) \end{array}$$

permettant le pointage vers une alvéole $T[f(k)]$ de la table de hachage pour la clé k, est construite à partir d'une fonction h appelé **fonction de hachage** avec :

$$\boxed{f(k) = h(k) \ mod \ m}$$

h doit être judicieusement choisie afin de minimiser le nombre de collisions.

Propriété II-1: UNIFORMITÉ DE LA FONCTION DE HACHAGE

Pour minimiser le nombre de collisions, une fonction de hachage doit être la plus uniforme possible, c'est à dire assurer la répartition la plus uniforme possible des clés dans les alvéoles de la table de hachage, immitant donc un comportement le plus "aléatoire" possible.

Plus formellement : pour $f(k) \in [\![0, m-1]\!]$, il faut que la probabilité que $h(k) \ mod \ m$ soit égal à $f(k)$ soit $\frac{1}{m}$.

II.3 Gestion des collisions : résolution par chaînage

a - Principe

L'idée, pour résoudre le problème des collisions, est de placer dans une liste chaînée toutes les valeurs hachées vers une même alvéole. L'alvéole ne contient alors pas la liste, mais l'adresse mémoire contenant la tête de la liste chaînée, le chainage permettant l'accès aux valeurs suivantes, si elles existent (pour les valeurs hachées sans collision, la liste ne comporte qu'un seul élément) :

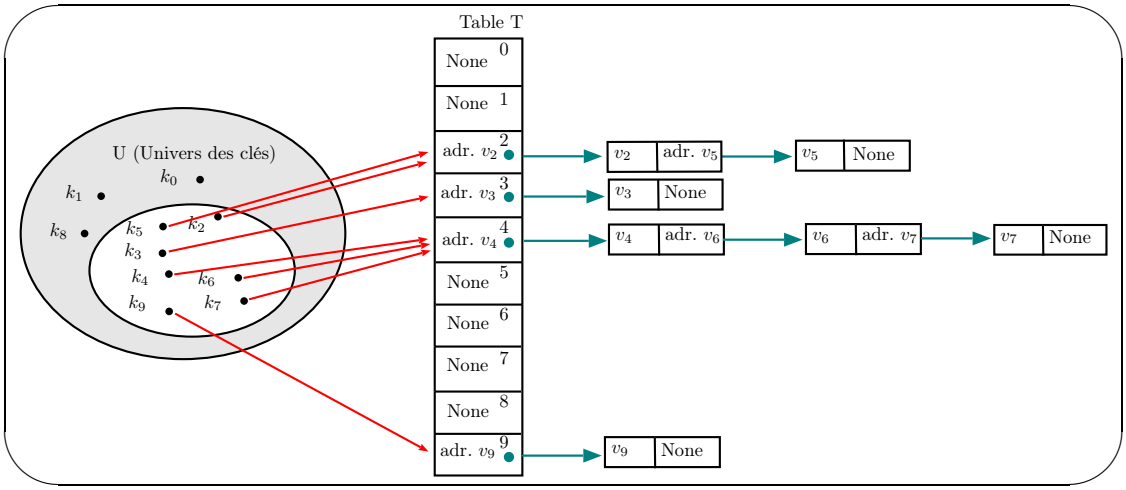


FIGURE VII.5 – Résolution des collisions par listes chaînée : chaque alvéole de la table de hachage $T[f(k)]$ contient l'adresse pointant vers la tête ce la liste chaînée.

b - Analyse de complexité

Le type dictionnaire est natif en Python (type `dict`) et la syntaxe des primitives sera examinée dans le prochain chapitre ; on se contentera donc ici de les indiquer en langage naturel :

INSÉRER_DICT_LISTE_CHAINÉE(T,X) :
insérer x en tête de la liste chainée $T[f(\text{clé}(x))]$

RECHERCHER_DICT_LISTE_CHAINÉE(T,K) :
rechercher un élément de clé k dans la liste chaînée $T[f(k)]$

SUPPRIMER_DICT_LISTE_CHAINÉE(T,X) :
supprimer x de la liste $T[f(cl(x))]$

Les complexités rattachées à ces opérations sont :

- INSÉRER-LIRE : sans surprise, le temps de calcul du hachage se faisant en temps constant $\mathcal{O}(1)$, et l'insertion dans une liste chaînée guère plus, on a :

$$C_{dict}(\text{insérer}) = \mathcal{O}(1)$$

- RECHERCHER-LIRE-SUPPRIMER : ces trois opérations vont nécessiter de repérer dans la liste chaînée la position de l'élément recherché ; l'étude de cette complexité est un peu plus délicate !

La table de hachage T stocke toujours n éléments répartis dans m alvéoles ; le **taux de remplissage** de la table est défini par $\alpha = \frac{n}{m}$ (nombre moyen d'éléments par chaîne) avec donc $\alpha \geq 1$.

◇ **Pire des hachages :** c'est la situation où les n éléments sont hachés vers la même alvéole, la liste chaînée étant alors de longueur n ; la complexité est donc celle obtenue avec la liste chaînée seule (si l'on exclut le temps de hachage), soit :

$$\boxed{C_{dict}(\text{rechercher(Pire)}) = \mathcal{O}(n)}$$

L'intérêt des dictionnaires est alors nul ! Cela justifie encore une fois le soin qu'il faut apporter au choix de la fonction de hachage pour éviter cette situation.

◇ **Meilleur des hachages :** dans ce cas, nous allons supposer un hachage uniforme, c'est à dire que chaque élément possède la même chance d'être haché vers l'une quelconque des alvéoles (hachage uniforme).

Appelons j l'indice repérant une alvéole $j \in [0, 1, 2, \ldots, m-1]$ et n_j la longueur de la liste chaînée stockée dans l'alvéole $T[j]$; il vient donc :

$$n = n_0 + n_1 + \ldots + n_{m-1} \qquad \text{et} \qquad \overline{n_j} = \alpha = \frac{n}{m}$$

On doit alors considérer deux cas de figure :

– **Si la recherche est infructueuse**, i.e. l'élément n'appartient pas au dictionnaire :

en considérant le hachage uniforme, toute clé k possède des chances égales d'être hachée en direction d'une quelconque des m alvéoles. La complexité de recherche dans ce cas correspond donc au temps moyen nécessaire au balayage complet de la liste chainée présente en $T[f(k)]$, or cette liste chaînée possède une longueur moyenne $\overline{n_j} = \alpha$.

En comptant le temps de hachage en $\mathcal{O}(1)$, la complexité moyenne est donc :

$$\boxed{C_{dict_{infr.}}(\text{rechercher-lire}) = \Theta(1+\alpha)}$$

– **Si la recherche est fructueuse** : ce cas de figure est plus délicat à évaluer car chaque liste n'a pas la même probabilité d'être examinée.

La probabilité qu'une liste soit examinée est évidemment proportionnelle au nombre d'éléments qu'elle contient. Le nombre d'éléments examinés pour la recherche de l'élément el dans une liste chaînée correspond donc à 1 (élément trouvé), plus le nombre d'éléments ajoutés après el et qui sont placés avant dans la liste chaînée.

Ainsi, le nombre moyen d'éléments examinés s'obtient en faisant la moyenne sur les n éléments x de la table **de 1 plus le nombre moyen d'éléments ajoutés à la liste de x après que celui-ci a été ajouté à sa liste.**

Posons x_i le ième élément inséré dans la table avec $i = 1, 2, \ldots, n$, et également $k_i = cle[x_i]$. On appelle $X_{ij} = I(f(k_i) = f(k_j))$ la variable indicatrice de l'événement $f(k_i) = f(k_j)$, soit :

$$X_{ij} = I(f(k_i) = f(k_j)) = \begin{cases} 1 \text{ si } f(k_i) = f(k_j) \\ 0 \text{ sinon} \end{cases}$$

En supposant un hachage uniforme, nous savons que $P(f(k_i) = f(k_j)) = \frac{1}{m}$. On montre facilement ensuite (c.f. remarque ci-dessous) que dans ces conditions $\overline{X_{ij}} = \frac{1}{m}$.

Le nombre moyen d'éléments examinés dans une recherche fructueuse est donc :

$$\begin{aligned}
&\overline{\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)} \\
&\qquad=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\overline{X_{ij}}\right)\\
&\qquad=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)\\
&\qquad=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)=1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)\\
&\qquad=1+\frac{1}{nm}\left(n^2-\frac{n(n-1)}{2}\right)=1+\frac{n-1}{2m}\\
&\qquad=1+\frac{\alpha}{2}-\frac{\alpha}{2n}
\end{aligned}$$

Le temps total de calcul est donc (en incluant le temps requis requis pour la hachage) :

$$\Theta\left(2+\frac{\alpha}{2}-\frac{\alpha}{2n}\right)$$

soit :

$$\boxed{C_{dict_{fruct.}}(\text{rechercher-lire})=\Theta(1+\alpha)}$$

Remarque II-1: DÉMONSTRATION

Le lien entre la variable indicatrice X_{ij} et la probabilité $P(f(k_i)=f(k_j))$ est simple à établir :

$$\begin{aligned}
\overline{X_{ij}}&=\overline{I(f(k_i)=f(k_j))}\\
&=1\times P(I(f(k_i)=f(k_j))+0\times P(I(f(k_i)\neq f(k_j)))\\
&=1\times\frac{1}{m}+0\times\left(1-\frac{1}{m}\right)\\
&=\frac{1}{m}\ CQFD!
\end{aligned}$$

Exercice de cours: **(II.3) - n° 3.** *Insertion de clés (d'après* **Introduction à l'algorithmique** *de Thomas Cormen Ed. Dunod 2nde édition 2001)*
Montrer comment on réalise l'insertion des clés $5, 28, 19, 15, 20, 33, 12, 17, 10$ *dans une table de hachage où les collisions sont résolues par chainage. On suppose que la table contient* 9 *alvéoles et que la fonction d'adressage est* $f(k) = k \bmod 9$.

II.4 Bilan

Les dictionnaires constituent donc une structure de données plutôt idéale puisqu'elle réduit de manière substantielle la complexité temporelle des opérations de base de traitement des données ;

	Tableau	Liste chaînée	Tableau dynamique (listes)	Dictionnaire
Recherche puis lecture au début	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Recherche puis lecture en fin	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$ ou $\Theta(1+\alpha)$
insertion en début	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
insertion en fin	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
suppression en début	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
suppression en fin	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

II.5 Comparaison "de terrain" : dictionnaire VS liste pour la recherche d'un élément

Listing VII.2 – Affectation classique d'une liste d'entiers aléatoires sans répétition

```
import random as rd
import time as t

def liste_aleatoire_list(n,a,b):
  L,n=[],0
  while n < N:
    q=rd.randint(a,b)
  if not(q in L):
    L.append(q)
    n+=1
  return L
for k in range(2,5):
  t0=t.time()
  liste_aleatoire_list(10**k,10**9,10**10)
  t1=t.time()
  print('k = %s -temps
    de calcul:'%(k),t1-t0,"s")
```

```
k = 2 -temps de calcul : 0.0 s
k = 3 -temps de calcul : 0.015605449676513672 s
k = 4 -temps de calcul : 0.7729330062866211 s
```

Listing VII.3 – Affectation d'une liste d'entiers aléatoires sans répétition avec usage d'un dictionnaire.

```
import random as rd
import time as t

def liste_aleatoire_dict(n,a,b):
  L,D,n=[],{},0
  while n < N:
    q=rd.randint(a,b)
    if not(q in D):
      L.append(q)
      D[q]=n
      n+=1
  return L
for k in range(2,5):
  t0=t.time()
  liste_aleatoire_dict(10**k,10**9,10**10)
  t1=t.time()
  print('k = %s -temps
    de calcul:'%(k),t1-t0,"s")
```

```
k = 2 -temps de calcul : 0.0 s
k = 3 -temps de calcul : 0.0 s
k = 4 -temps de calcul : 0.015629291534423828 s
```

Exercice de cours: **(II.5) - n° 4.** *Commenter les résultats obtenus.*