LES DICTIONNAIRES

Informatique Tronc Commun

E. CLERMONT

1

RAPPELS SUR LES TYPES EN PYTHON

Les types simples :

- Les **entiers** : int
- Les **flottants** : float
- Les **booléens** : bool
- Les **chaines de caractères** : str

Particularité : chaque caractère de la chaine possède un indice => ce système d'indice permet d'accéder à une partie de la chaine (séquence).

Les types composés ou construits :

- Les **p-uplets** : tuples
- Les **listes** : list
- Les **tableaux**: array

Eléments indexés par une suite d'entiers
=> Séquences

- Les **dictionnaires** : dict

2

TYPES EN PYTHON – MUTABLE/IMMUTABLE

```
1   # Exemples de types immutables
2   # avec des nombres
3   n1 = 5
4   n2 = n1
5   n1 = 6
6   print ( n1, " ",  n2 )          # affiche :
7
8   # avec des tuples
9   tuple1 = (1,2,3,4)
10  tuple2 = tuple1
11  tuple1 = tuple1 + (5,6)
12  print (tuple1, " ", tuple2)     # affiche :
13
14  # Exemples de types mutables
15  l1 = [ 1,2,3,4 ]
16  l2 = l1
17  l1[0] = 5
18  print ( l1, l2 )                # affiche :
19
20  l2 =l1.copy()
21  l1[0] = -1
22  l1.append(7)
23  print ( l1, l2 )                # affiche :
```

3

TYPES EN PYTHON – MUTABLE/IMMUTABLE

En python, les types des variables se divisent en 2 catégories :

- les types **mutables**
- les types **immutables**.

- Un type est **immutable (ou immuable ou persistant)** si la valeur d'une variable de ce type ne peut changer que par l'affectation d'une nouvelle valeur à cette variable.
- Dans le cas contraire, il sera **mutable**.
- **int, float, str, bool, tuple : sont immutables** Les valeurs des variables de ce type ne peuvent être modifiées que par une affectation.
- Les **listes et les dictionnaires sont mutables.**

Les opérations de modification d'une liste le démontrent : ajout d'un nouvel élément (**append**), suppression d'une élément (**del**), … Ces opérations ne sont pas des affectations .

De plus, la valeur de l'élément particulier d'une liste peut être modifiée par l'affectation d'une nouvelle valeur à cet élément. Indirectement, cela modifie également la valeur de la liste. Bien qu'il s'agisse d'une affectation, cela montre également que les listes sont mutables, car le membre gauche de cette affectation n'est pas la liste elle même.

- **Exemple**: Soit la liste **L** a pour valeur [1,9,6,1]. Pour modifier cette valeur en [1,9,6,4], on n'est pas obligé d'écrire **L = [1,9,6,4]** : **L[3]=4** suffit.

4

TYPES EN PYTHON - BILAN

Types	Simple	Construit / composé	Séquence	Mutable	Immutable
int					
float					
bool					
str					
tuple					
list					
array					
dict					

MANIPULATION DE DICTIONNAIRES

- Un **dictionnaire ou tableau associatif** : type Python qui permet de stocker des données.
- Contrairement aux listes ou aux tuples, où les éléments sont indexés par des entiers,

les éléments (ou valeurs) d'un dictionnaire sont indexés par des **clés** qui peuvent être de n'importe quel type (non mutable).

- Par exemple, on peut utiliser des chaînes de caractères pour les clés.

- **Un dictionnaire est alors un ensemble de couples clé : valeur**

- **Exemples de dictionnaires**

```
lesFruits = {"poire": 3, "pomme": 4, "orange": 2}
lesEtudiants = {110:'Marie',111:'Jane','112':'Remi',113:'Rena'}
```


MANIPULATION DE DICTIONNAIRES

Création d'un dictionnaire

- Création d'un dictionnaire vide :

  `dico = { }` ou avec la fonction dict : `dico = dict()`

- Initialisation d'un dictionnaire tout en le créant :

```
annuaire = {"Julie" : "067898734" , "Carla" : "0799910783" , "Leo" : "0669842704" }
```

- Il est alors possible d'ajouter autant d'éléments qu'on le veut, que le dictionnaire soit vide ou non.

Exemple :

```
annuaire["Romain" ]= "0765333165"
print(annuaire)
```

Le dictionnaire contient alors :

{'Julie': '067898734', 'Carla': '0799910783', 'Leo': '0669842704', 'Romain': '0765333165'}

MANIPULATION DE DICTIONNAIRES

- On peut stocker n'importe quel type de données dans un dictionnaire.

Exemples :

```
Eleve= { }
Eleve[ 'nom' ] = 'Matthieu'
Eleve[ 'classe' ] = 'MPSI'
Eleve[ 'notes info' ] = [17,12,10]
Eleve[ 'age' ] = 18
print ( Eleve )
#affiche  {'nom': 'Matthieu', 'classe': 'MPSI', 'notes info': [17, 12, 10], 'age': 18}
```

MANIPULATION DE DICTIONNAIRES

- Exemple de valeur de type dictionnaire

```
Eleves= {
    'Matthieu' : { 'Math' : 12, 'Physique':11, 'Informatique':18, 'Anglais' : 14 } ,
    'Ambre':{ 'Math' : 14, 'Physique':16, 'Informatique':20, 'Anglais' : 10 },
    'Chloe':{ 'Math' : 9, 'Physique':12, 'Informatique':11, 'Anglais' : 18 }
}

Eleves['Ambre']['Anglais'] = 15
```


MANIPULATION DE DICTIONNAIRES

Détermination du nombre d'éléments d'un dictionnaire : fonction len(…)

Exemple :

```
annuaire = {'Julie':'067898734','Carla':'0799910783','Leo':'0669842704','Romain':'0765333165'}
lg = len(annuaire)
print(lg)
```

lg vaut 4 : nombre de couples clé/valeur présents dans annuaire.

MANIPULATION DE DICTIONNAIRES

Récupération de valeurs à partir de la clé

- Pour récupérer la valeur associée à une clé,

on utilise les crochets [] comme pour une liste ou un tuple dans lesquels on précise la clé recherchée.

Exemple :

```
annuaire['Julie']          #correspond à '067898734'
```

- Une **erreur « KeyError »** est générée si la clé fournie n'existe pas dans le dictionnaire.

MANIPULATION DE DICTIONNAIRES

Parcours de dictionnaires

Eleves = {'nom': 'Matthieu', 'classe': 'MPSI', 'notes info': [17, 12, 10], 'age': 18}

Affichage des clés	Affichage des valeurs de chaque clé

for cle in Eleve:
  print(cle)
	for cle in Eleve:
	  print(Eleve[cle])

for cle in Eleve.**keys():**
  print(cle)
	for val in Eleve.**values():**
	  print(val)

nom classe notes info age	Matthieu MPSI [17, 12, 10] 18
Affichage des clés et des valeurs for cle, val in Eleve.**items():** print(cle,val)	
nom Matthieu classe MPSI notes info [17, 12, 10] age 18	

EXERCICES APPLICATIFS

Exercice 1

- On considère un dictionnaire dont :
  - les clés sont les noms des élèves
  - les valeurs sont les moyennes générales obtenues.

Ecrire un programme qui :

- **crée** un tel dictionnaire
- puis le **partitionne** en 2 dictionnaires :
  - **etudiants_Admis** dont les clés sont les étudiants admis et les valeurs des clés sont les moyennes obtenues (moyenne supérieure ou égales à 10).
  - **etudiant_NonAdmis** dont les clés sont les étudiants non admis et les valeurs des clés sont les moyennes obtenues (moyenne inférieureou égale à 10).
- Et enfin **affiche** :
  - le nom des étudiants admis et non admis, ainsi que leur note.
  - puis le nom des étudiants admis (sans leur note)

EXERCICES APPLICATIFS

Exercice 2 : Inversion de dictionnaires

1 - Ecrire une fonction **inverseDictionnaire** qui :

- prend en paramètres un dictionnaire,
- retourne un dictionnaire dans lequel les valeurs sont devenues les clés.

Exemple d'exécution :

```
d1={ 'MPSI' : 43, 'PCSI': 42, 'MP':41, 'PC':32 }
print ( inverseDictionnaire( d1 ) ) # affiche {43: 'MPSI', 42: 'PCSI', 41: 'MP', 32: 'PC'}
```

2- Le dictionnaire résultant a-t-il la même longueur que le dictionnaire initial?

Justifier la réponse.

MANIPULATION DE DICTIONNAIRES

Présence d'une clé dans un dictionnaire

- Pour vérifier **si une clé** (pas une valeur) **existe** dans un dictionnaire,

on peut utiliser le test d'appartenance avec l'instruction **in** qui renvoie un booléen.

Exemple :

```
print('age' in Eleve)      #affiche True
print('tel' in Eleve)      #affiche False
```


EXERCICES APPLICATIFS

- **Exercice 3 : Nombre d'occurrences des lettres dans un mot**

Ecrire une fonction **determine_nb_occurences** qui :

- prend en paramètres un mot
- et qui retourne un dictionnaire qui comporte le nombre d'occurrences de chaque lettre présente dans le mot.

Exemple d'exécution :

```
print ( determine_nb_occurences("bonjour") )
# affiche {'b': 1, 'o': 2, 'n': 1, 'j': 1, 'u': 1, 'r': 1}
```

MANIPULATION DE DICTIONNAIRES

Suppression d'un couple clé,valeur

- Pour supprimer un élément d'un dictionnaire, on utilise la fonction **del.**

  del dico**[**cle**]**

- Exemple :

```
Eleve = {'nom': 'Matthieu', 'classe': 'MPSI', 'notes info': [17, 12, 10], 'age': 18}
del Eleve['classe']
print (Eleve)
# affiche {'nom': 'Matthieu', 'notes info': [17, 12, 10], 'age': 18}
```


MANIPULATION DE DICTIONNAIRES

Copie de dictionnaires

- Les dictionnaires sont des objets mutables. Donc si l'on souhaite copier un dictionnaire, on ne peut pas utiliser l'opérateur d'affectation =.

Exemple :

```
Eleve = {'nom': 'Matthieu', 'classe': 'MPSI', 'age': 18}
Eleve1 = Eleve
Eleve['age'] = 20
print (Eleve, Eleve1)
# affiche   {'nom': 'Matthieu', 'classe': 'MPSI', 'age': 20}
#           {'nom': 'Matthieu', 'classe': 'MPSI', 'age': 20}
```

- On peut utiliser la fonction ***copy*** **:**

```
d = {.....}
d2 = d.copy()
```

- Exemple :

```
Eleve = {'nom': 'Matthieu', 'classe': 'MPSI', 'age': 18}
Eleve2 = Eleve.copy()
Eleve['age'] = 19
print (Eleve, Eleve2)
# affiche   {'nom': 'Matthieu', 'classe': 'MPSI',  'age': 19}
#           {'nom': 'Matthieu', 'classe': 'MPSI',  'age': 18}
```


MANIPULATION DE DICTIONNAIRES

Copie de dictionnaires

- **Attention si certaines valeurs sont des objets mutables.**

Exemple :

```
Eleve = {'nom': 'Matthieu', 'classe': 'MPSI', 'notes info': [17, 12, 10], 'age': 18}
Eleve2 = Eleve.copy()
Eleve['age'] = 19
Eleve['notes info'][2] = 12
print (Eleve, Eleve2)
# affiche   {'nom': 'Matthieu', 'classe': 'MPSI', 'notes info': [17, 12, 12], 'age': 19}
#           {'nom': 'Matthieu', 'classe': 'MPSI', 'notes info': [17, 12, 12], 'age': 18}
```

- Dans ce cas, il est recommandé d'utiliser la copie en profondeur avec *deepcopy* (module copy)

Exemple :

```
import copy
Eleve3 = copy.deepcopy(Eleve)
Eleve['age'] = 19
Eleve['notes info'][2] = 20
Eleve['notes info'].append(19)
print (Eleve, Eleve3)
# affiche   {'nom': 'Matthieu', 'classe': 'MPSI', 'notes info': [17, 12, 20, 19 ], 'age': 19 }
#           {'nom': 'Matthieu', 'classe': 'MPSI', 'notes info': [17, 12, 12], 'age': 18}
```

EXERCICES APPLICATIFS

Exercice 4 : Chiffres d'affaires de commerciaux

On dispose d'un dictionnaire associant à des noms de commerciaux d'une entreprise le montant des ventes qu'ils ont réalisées.

- Exemple :

  ventes={"Jeff Bezos":10, "Bill Gates Jobs":13, "Mark Zuckerberg":8, "Elon Musk":15}

- Écrire la fonction *chiffre_Affaires* qui prend en entrée ce dictionnaire et renvoie le chiffres d'affaires, ie le montant total de ventes.
- Écrire la fonction *meilleur_Commercial* qui prend en entrée un dictionnaire et renvoie le nom du vendeur ayant réalisé le montant de ventes le plus élevé, ainsi que le montant. Si plusieurs vendeurs sont ex-aequo sur ce critère, la fonction devra retourner l'ensemble des commerciaux.
- Écrire la fonction *supprime_PireCommercial* qui prend en entrée un dictionnaire et supprime le (ou les pires) des commerciaux (ie celui quia vendu le moins) si plusieurs vendeurs sont ex-aequo.

EXERCICES APPLICATIFS

Exercice 5 : nombre de points d'un mot au Scrabble

Vous disposez du dictionnaire suivant :

```
dico_scrabble={"A":1,"B":3,"C":3,"D":2,"E":1,"F":4,"G":2,"H":4,"I":1,"J":8,"K":10,"L":1,"M
":2,"N":1,"O":1,"P":3,"Q":8,"R":1,"S":1,"T":1,"U":1,"V":4,"W":10,"X":10,"Y":10,"Z":10}
```

Créer une fonction **score_scrabble** qui :

- prend en paramètres un mot et un dictionnaire
- et retourne le nombre de points correspondant au mot.

Exemple d'exécution:

```
print ( score_scrabble("bonjour", dico_scrabble) ) # affiche 16
```


COMPARAISON LISTES ET DICTIONNAIRES

- Contrairement à ce que l'on pourrait croire, l'accès aux éléments d'un dictionnaire est très efficace (grâce à l'utilisation d'une fonction de « hachage » qui associe un entier à chaque élément du dictionnaire).
- Points à retenir (en simplifiant, notamment sur la fonction de hachage)

Soit n la taille de la liste ou du dictionnaire

- Une **liste est intéressante lorsque l'ordre des éléments est important**, ou que **l'indexation** par des entiers est importante.
  A l'opposé, un dictionnaire permet d'avoir un ensemble de clés totalement quelconque,
- **Le test d'appartenance** (in) est en $O(1)$ pour un dictionnaire, alors qu'il peut être en $O(n)$ pour une liste.
- Les temps d'accès à un élément donné est en $O(1)$ pour les listes et les dictionnaires(un peu plus rapide pour les listes),
- le temps de suppression d'un élément est en $O(1)$ pour un dictionnaire et en $O(n)$ pour une liste (sauf si c'est le dernier élément qu'on retire avec pop())

FONCTIONNEMENT DES DICTIONNAIRES

- **Aparté sur les tableaux et listes chainées**

- **Principe d'un tableau (numpy.array)**

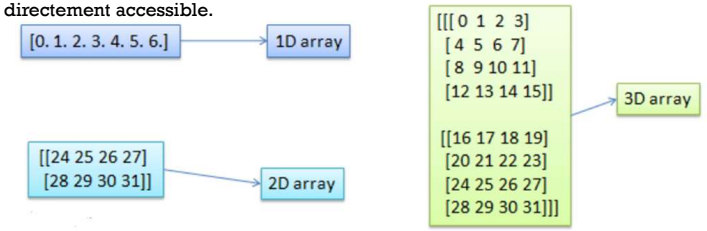
Un tableau est un conteneur possédant **un nombre fixe d'éléments d'un même type**. Le **nombre et le type des éléments** sont définis à la création et ne sont **pas modifiables**.

Le tableau est immuable (contrairement à la liste).

Chaque élément du tableau est directement accessible en un temps 0(1):

les données sont stockées dans des espaces contigus en mémoire.

L'élément tab[i] est directement accessible.



FONCTIONNEMENT DES DICTIONNAIRES

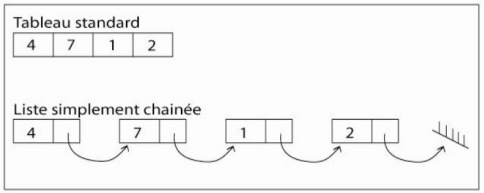
- **Tableaux et listes chainées**

Principe de liste chainée

- Une liste chainée est un conteneur éventuellement vide donc chaque élément contient une donnée et une adresse mémoire de la cellule suivante.

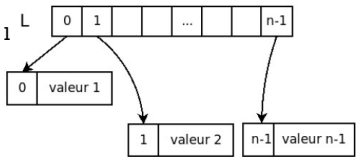
Il est possible de :

- créer un liste vide,
- ajouter un élément dans la liste,
- supprimer un élément,
- insérer un élément à une position donnée
- accéder à la position de tête
- accéder au successeur d'une cellule.

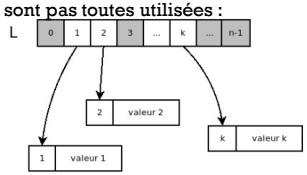


ADRESSAGE DIRECT

Un moyen simple de gérer un ensemble de valeurs est **la liste dynamique** Python.
Gérées par **adressage direct**.
L'univers des clés est un ensemble d'entiers de 0 à $n-1$
et on a directement accès à la case k,
puis à sa donnée correspondante :



On pourrait mettre en place une structure analogue pour les dictionnaires mais « avec des trous », lorsqu'on a un ensemble fini de clés possibles mais qu'elles ne sont pas toutes utilisées :
Avantage : grande souplesse
Inconvénient : si l'ensemble des clés possibles est très grand,
on aura réservé un tableau d'adresses très grand
(donc beaucoup de place mémoire) pour peu de cases
réellement utilisées.



Exemple :pour des clés qui sont des chaînes de 5 caractères, on aurait 26^5 (presque 12 millions) cases à réserver pour n'en utiliser que quelques centaines/milliers.

FONCTIONNEMENT DES DICTIONNAIRES

Hachage

- Le hachage est un **mécanisme** permettant de **transformer la clé en un nombre unique** permettant l'accès à la donnée, un peu à la manière d'un indice dans un tableau.
- La notion de hachage est omniprésente en informatique et est **centrale dans le fonctionnement des dictionnaires**.

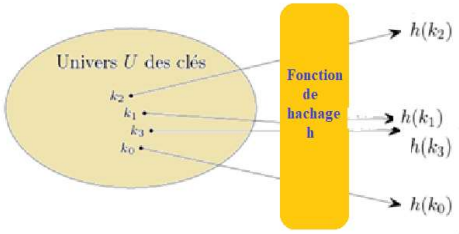
- **Exemple d'utilisations du hachage**
- Stockage des mots de passe dans un système informatique un peu sécurisé : le mot de passe ne **doit pas être stocké en clair**. Une empreinte est générée avec une fonction de hachage, afin que si le service est piraté et que les comptes sont dérobés, il ne soit pas possible de reconstituer le mot de passe à partir de l'empreinte.

FONCTIONNEMENT DES DICTIONNAIRES

Hachage

- **Définition d'une fonction de hachage**

Une fonction de hachage est une fonction qui **calculer une empreinte unique** à partir de la donnée fournie en entrée.



La fonction de hachage doit respecter les **règles suivantes** :
- La longueur de l'empreinte (valeur retournée par la fonction de hachage) doit être toujours la même, quelle que soit la donnée fournie en entrée.
- Connaissant l'empreinte, il ne doit pas être possible de reconstituer la donnée d'origine
- Des données d'entrée différentes doivent donner **dans la mesure du possible** des empreintes différentes.
- Des données d'entrée identiques doivent donner des empreintes identiques.

FONCTIONNEMENT DES DICTIONNAIRES

Il existe une fonction hash en Python.

Exemple d'utilisation:

```
#test fonction hash
unentier = 1
unechaine = 'dictionnaire'
unflottant = 4.55
# Afficher les valeurs de hachage
print("La valeur de hachage de l'objet integer est:"+ str(hash(unentier)))
print("La valeur de hachage de l'objet string est:"+ str(hash(unechaine)))
print("La valeur de hachage de l'objet float est:"+ str(hash(unflottant)))
```

```
La valeur de hachage de l'objet integer est:1
La valeur de hachage de l'objet string est:-3411529041035809772
La valeur de hachage de l'objet float est:1268213655067531268
```

FONCTIONNEMENT DES DICTIONNAIRES

Table de hachage

L'idée est de **réduire l'ensemble des clés grâce à une fonction de hachage**, c'est-à-dire une fonction h qui part du domaine des clés et renvoie un entier entre 0 et m − 1 de sorte que pour les clés utilisées, les valeurs de h soient différentes.

- Soit U l'ensemble des clés envisageables.
- L'idée fondamentale de la table de hachage est de se ramener au cas de l'adressage direct,

  càd à des clés qui correspondant à des indices dans un tableau.

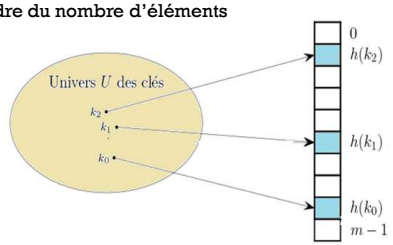
Soit m, entier pas trop grand (idéalement un entier de l'ordre du nombre d'éléments d'informations que l'on compte gérer)

On se donne une fonction de hachage h :

h : U → {0, …, m−1}

- L'idée est de ranger l'élément de clé k non pas dans une case de tableau t[k], comme dans l'adressage direct (cela n'a d'ailleurs aucun sens si k n'est pas un entier) ,

  mais dans **t[h(k)]**.



FONCTIONNEMENT DES DICTIONNAIRES

Table de hachage pour les dictionnaires

A la création d'un dictionnaire,

- **une fonction de hachage h est choisie**
- et **un tableau de taille m est créé**.

Chaque élément de ce tableau sera nommé **alvéole**.

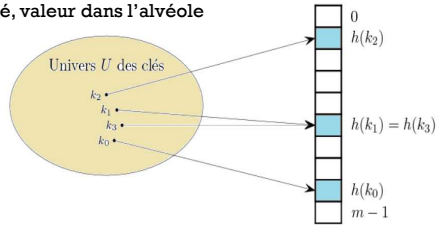
- A chaque insertion d'un couple clé-valeur, **h(k) est calculée**.

La valeur obtenue est comprise **entre 0 et m-1**.

- Une 1ère approche serait de placer le couple-clé, valeur dans l'alvéole

mais le tableau est de taille limitée.

=> Il y a aura donc forcément des **collisions** : plusieurs clés auront nécessairement les mêmes valeurs de hachage.



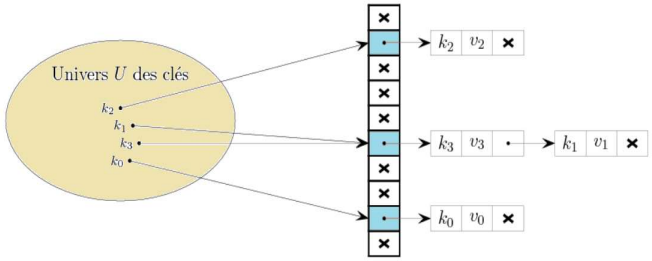
FONCTIONNEMENT DES DICTIONNAIRES

Table de hachage pour les dictionnaires – Gestion des collisions

- Approche choisie pour gérer des collisions inévitables : **Résolution par chainage**

Placer dans les alvéoles des pointeurs (ie des adresses) vers des listes chainées

dans lesquelles les couples-clés, valeurs sont ajoutés ou supprimés au fur et à mesure des actions effectuées sur le dictionnaire



FONCTIONNEMENT DES DICTIONNAIRES

Table de hachage pour les dictionnaires – Gestion des collisions

- Approche choisie pour gérer des collisions inévitables : **Résolution par adressage ouvert**

On cherche à calculer une autre place libre ou la 1ère place libre qui suit.
La position de ces cases est déterminée par une méthode de « sondage ».

Lors d'une recherche, si la case obtenue par hachage direct ne permet pas d'obtenir la bonne clé, une recherche sur les cases obtenues par une méthode de sondage est effectuée jusqu'à trouver la clé, ou non, ce qui indique qu'aucune clé de ce type n'appartient à la table.

Méthodes de sondage courantes :

- **le sondage linéaire** : l'intervalle entre les cases est fixe, souvent 1. on parcourt les alvéoles successivement jusqu'à ce qu'on en trouve une vide. On revient au début de la table si la fin a été atteinte.
- le **sondage quadratique** : l'intervalle entre les cases augmente linéairement (les indices des cases augmentent donc quadratiquement), ce qui peut s'exprimer par la formule :

  $h_i(k) = k + i^2$
- le **double hachage** : l'adresse de la case est donnée par une deuxième fonction de hachage, ou hachage secondaire.

FONCTIONNEMENT DES DICTIONNAIRES

Table de hachage pour les dictionnaires

- Lorsque le nombre de clés augmente fortement,

la table est alors **redimensionnée** avec un nouveau choix de fonction de hachage et de m.

- La **performance d'une table de hachage dépend de la fonction de hachage** choisie.

En général, on cherche à limiter le nombre de collisions tout en optimisant le taux d'occupation des alvéoles.

=> le taux ou le facteur de remplissage :

$$\alpha = n/m$$

où n est le nombre de clés (ou de couples clés-valeurs)

et m la longueur du tableau.

- En Python, les clés peuvent être de type entier, flottant, chaines de caractères (mais surtout pas de types mutables tels que des listes)

Dans la pratique, les clés transmises sont transformées en entier si elles ne sont pas de type entier puis la valeur de hachage est calculée sur cet entier (ce fonctionnement est également appliqué pour la compression et le cryptage). On effectue donc un prétraitement pour obtenir un entier.

FONCTIONNEMENT DES DICTIONNAIRES

Exemples classiques de fonctions de hachage :

- **Division** **h (k) = k%m**

La qualité du hachage dépend fortement de la valeur de m choisie si les clés ont une répartition non aléatoire. En pratique, on choisit des nombres premiers loin de puissances de 2.

- **Multiplication** **h(k) = partieEntiere (m * frac(k * c))**

Dans la méthode de multiplication, on multiplie la clé **k** par un nombre réel constant c dans la plage **0 < c < 1** et on extrait la *partie fractionnaire de* ***k * c*** . Ensuite, on multiplie cette valeur par la taille de la table **m** et prend la partie entière.

Il faut cependant toujours choisir une constante, qui peut faire varier l'efficacité de la compression. Knuth suggère que pour c =$(\sqrt{5}-1)/2$, la fonction marchera bien dans la plupart des cas.

EXERCICES APPLICATIFS

Exercice 6

- Dans cet exercice, les clés sont de type chaînes de caractères. On cherche à transformer ces clés en s'appuyant sur le codage ASCII de chaque caractère.
  Pour obtenir le code ASCII associé à chaque caractère, Python propose la fonction ord.

ord(*c*): renvoie le nombre entier représentant le code Unicode du caractère passé en paramètre.

Exemples : ord('a') renvoie le nombre entier 97

ord('€') (symbole euro) renvoie 8364.

1- Ecrire une **fonction transforme_chaine_entier(chaine)** qui prend en paramètre une chaine de caractères *chaine* et retourne la valeur :

$$\sum_{i=0}^{n-1} ord(chaine[i]) * 256^i$$

Exemples d'exécution :

print ("test devient ", transforme_chaine_entier("test"))

affiche test devient 1952805748

print ("test devient ", transforme_chaine_entier("test du dictionnaire"))

affiche test du dictionnaire devient 664505362109310594164837447122569333616419631717

EXERCICES APPLICATIFS

- Les entiers obtenus sont donc de très grande taille.

⇒ Il va donc falloir les transformer pour les stocker dans la table de hachage.

2- Ecrire la fonction **hachage(nb, m)** telle que :

hachage : nb -> nb%m avec m qui correspond à la longueur de la clé de hachage.

- Appliquées aux 2 valeurs entières précédentes, pour une clé de longueur 256, on obtient respectivement: 101 et 116

3- **Ecrire et tester hachage_mult (nb, m)** qui réalise cette fonction de hachage par multiplication :

Hachage_mult **:** nb , k -> partieEntiere (m * frac(k * c))

Exemple d'exécution :

print ("test devient après hachage multiplication", hachage_mult(nbtest1, longueur_cle , c))

#test devient après hachage multiplication 176

EXERCICES APPLICATIFS

Exercice 7 : Algorithme de compression LZ78

- L'algorithme de compression Lempel-Ziv '78, est l'un des tout premiers algorithmes de compression génériques.
- Il a rapidement été adopté dans de nombreux logiciels commerciaux et libres.
- Il est par exemple à la base du format de compression zip.
- Il s'agit d'un algorithme basé sur un dictionnaire :

les suites de symboles dans la source sont encodées par un dictionnaire dynamique, qui est rempli au fur et à mesure que l'on parcourt le texte.

EXERCICES APPLICATIFS

Exercice : Algorithme de compression LZ78 – Principe de la compression

a|b|r|ac|ad|ab|ra|rab|ar|aba|ran|

Entrée (Clé)	Index (valeur)	Chaine à ajouter
''	0	-
a	1	0,a
b	2	0,b
r	...	...
...	...	...
...	...	...
...	...	...

code : '0,a|0,b|0,r|1,c|1,d|1,b|3,a|7,b|1,r|6,a|7,n|'

EXERCICES APPLICATIFS

Exercice : Algorithme de compression LZ78

1- Compression
Créer la fonction compressionLZ78(texte) qui effectue la compression du texte passé en paramètre et retourne le code, ainsi que le dictionnaire.

2- Décompression

L'algorithme de décompression fonctionne en sens inverse.
À partir de la liste alternée, appelée code, il faut reconstruire le dictionnaire au fur et à mesure.
Créer la fonction decompressionLZ78(code, dico) qui effectue la décompression.

Astuce : Pour faciliter la programmation, on peut inverser le dictionnaire utilisé pour la compression : les valeurs deviennent clés et les clés deviennent valeurs.

LES DICTIONNAIRES