Informatique Tronc Commun

PTSI-PT*

Séquence 3

Table de hachage et dictionnaire

Thème : Dictionnaires, fonctions et tables de hachage

Objectifs

- Dictionnaires, clés et valeurs ;
- Principes du hachage et limitations sur le domaines des clés utilisables ;
- Usage des dictionnaires en programmation Python ;
- Syntaxe pour l'écriture des dictionnaires ;
- Parcours d'un dictionnaire ;

Table des matières

Cours - Dictionnaire et table de hachage

1 Dictionnaires

1.1 Description du type

Dans cette section, nous allons décrire ce qu'est un dictionnaire et notamment les opérations permises par celui-ci.

En 1ère année, nous avons utilisé les dictionnaires en « boîte noire » et comme un simple moyen de stockage de données.

Nous fixons deux ensembles C et V. L'ensemble C sera appelé l'ensemble des « clefs » et V l'ensemble des « valeurs ». Un dictionnaire permet d'associer à un élément de C un élément de V. Un dictionnaire se présente donc comme un ensemble non ordonné de couple (clé,valeur) comme ceci : {(clé1,valeur1),(clé2,valeur2),(clé3,valeur3),...,(clé n,valeur n)}. Un dictionnaire est une fonction (au sens mathématique) de C dans V. Une clef peut ne pas avoir de valeur associée, autrement dit un dictionnaire n'est pas forcément une « application » de C vers V.

On notera f, cette fonction : $f : C \longrightarrow V$

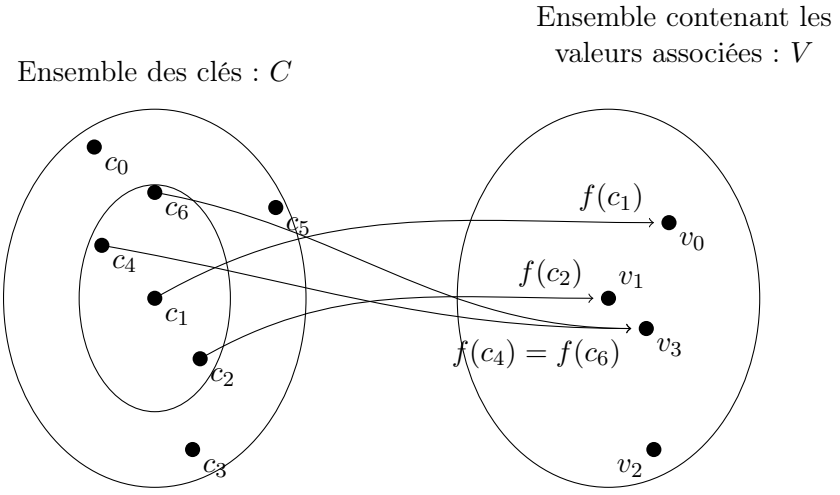


FIGURE 1 – Représentation d'un dictionnaire. Les clés c_1, c_2, c_4 et c_6 sont reliés à des valeurs. Celles-ci peuvent être identiques.

À cette structure sont associées des fonctions primitives (qu'on appellera primitives par la suite), permettant de la manipuler efficacement (complexité la meilleure possible). La liste suivante est non-exhaustive :

- Créer un dictionnaire ;
- Supprimer un élément sur la base de sa clé ;
- Insérer une paire clé-valeur ; c'est-à-dire étant donné $c \in C$ et $v \in V$, décider que v sera associée à c. Si une valeur était déjà associée à c, la nouvelle écrase l'ancienne ;
- Lire une valeur sur la base de sa clé.

Ces opérations sont réalisées de manière particulièrement efficace grâce au principe des tables de hachage (voir section 4).

1.2 Utilisation en Python

Les dictionnaires de Python sont **modifiables**. Cependant, attention, les **clefs doivent être des objets immutables et définis de manière unique**. L'ensemble des clefs C peut contenir des éléments de types différents, ce qui est important est que les clefs soient immutables et unique.

- Créer un dictionnaire vide : `d={}` ou `d=dict()` ;
- Associer v à la clé c : `d[c]=v` ou `d=dict([(c,v)])` ou `d.update({c:v})`. Le mot « Associer » signifie Modifier ou Insérer.
- Renvoyer la valeur associée à la clé c : `d[c]`. Déclenche une erreur si la clé c n'est pas présente dans `d`.
- Supprimer l'entrée correspondant à c dans `d` : `del d[c]`. On peut préférer utiliser `d.pop(c)` qui en plus renvoie l'élément supprimé. Déclenche une erreur si la clé c n'est pas présente dans `d`.
- La méthode `keys` renvoie l'ensemble des clés de `d`.
- Test de présence d'une clé c dans `d` : `c in d`. Il vaut mieux éviter `c in d.keys()` pour tester si la clé c est dans le dictionnaire `d`. En effet, de cette façon on crée la liste des clés du dictionnaire et on la parcourt pour tester si l'élément est présent. La complexité de cette approche est $\mathcal{O}(n)$ avec n le nombre de clés.

1.3 Un petit exemple pour utiliser un dictionnaire

À la suite d'une élection, les bulletins sont dépouillés. On récupère un tableau contenant tous les noms inscrits sur les bulletins trouvés dans l'urne. Par exemple :
`urne=["Maurice","Robert","Blanche","Ahmed","Maurice","Ahmed","Blanche","Ahmed",...]`

Pour déterminer le vainqueur de l'élection, en un seul parcours de l'urne, l'utilisation d'un dictionnaire est parfaitement adapté. Les clés seront les noms des candidats à l'élection et les valeurs le nombre de voix associé au candidat.

Un avantage avec un dictionnaire est qu'il n'est pas nécessaire de connaître à l'avance le nombre de candidats. En effet, si on traitait ce problème avec une simple liste, il serait nécessaire de savoir combien il y a de candidats pour initialiser en taille la liste et associer ensuite un index à un nom. Par exemple, le nombre de voix de `"Maurice"` serait indiqué en position 0, celui de `"Robert"` en position 1, etc.

Q1. Écrire une procédure permettant d'incrémenter la valeur associée à une clef s'il y en a, ou de l'initialiser à 1 dans le cas contraire :

```
1 def incr_dico(d,c):
2     ''' Procédure incrémentant les valeurs d'un dictionnaire d pour une clé c
3     Entrées :
4         - d : un dictionnaire
5         - c : une clef
6     Sortie : Rien (ceci est une procédure, sur un objet mutable : le
          dictionnaire d)
7     Effet :
8         Si c est une clef dans d, la valeur associée est incrémentée
9         sinon, elle est initialisée à 1.'''
```

Q2. Écrire une fonction `lecture_urne` prenant en argument une liste `urne` et retournant un diction-

naire ayant pour clés les noms des candidats et pour valeurs les nombres de voix de chacun d'entre eux.

```
def lecture_urne(urne):
    ''' Fonction qui à partir du contenu de l'urne renvoie un dictionnaire
        contenant le résultat individuel des candidats
    Entrée :
        - urne : liste des bulletins avec le nom des candidats
    Sortie :
        - dico : dictionnaire contenant le résultat individuel de chaque
            candidat
```

Q3. Créer alors une fonction `depouillement` retournant le vainqueur ou les vainqueurs de l'élection. Le ou les vainqueurs seront retournés dans une liste.

2 Implémentation à l'aide d'une fonction de hachage

2.1 Stockage

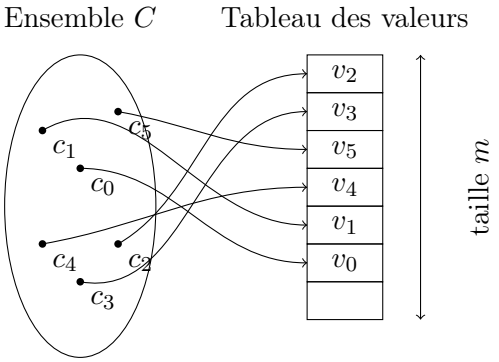


FIGURE 2 – Schéma symbolique de la représentation d'un dictionnaire quand il n'y a pas de collision (voir remarque 1).

Pour stocker les résultats, on peut imaginer un tableau de taille m. Si les clefs étaient des entiers compris entre 0 et $m-1$, le problème serait simple : on stockerait la valeur dans les cases correspondants à chacune des clés. C'est ce que l'on appelle l'**adressage direct**. Prenons une illustration de ce cas pour l'ensemble :

$$\{(c_0, v_0),\ (c_1, v_1),\ (c_2, v_2),\ (c_3, v_3),\ (c_4, v_4),\ (c_5, v_5)\} \qquad \text{voir FIGURE 2}$$

Cependant, ce n'est que rarement le cas ;-). Il y a sur des problèmes de tailles importantes plus de clés que de cases mémoire. Il va falloir trouver un moyen de faire le lien entre les clés et les cases du tableau. Soit une fonction : $h \ : \ C \longrightarrow [\![0, m-1]\!]$ qui associe aux différentes clés $c \in C$ un entier dans $[\![0, m-1]\!]$.

Remarque 1

Très souvent (en tout cas pour des dictionnaires de taille assez conséquentes) $|\ C\ | > m$. La fonction h n'est donc pas injective. Il existera donc des couples (c_1, c_2) dans C tels que $c_1 \neq c_2$ et $h(c_1) = h(c_2)$. On parle alors de **collision**.

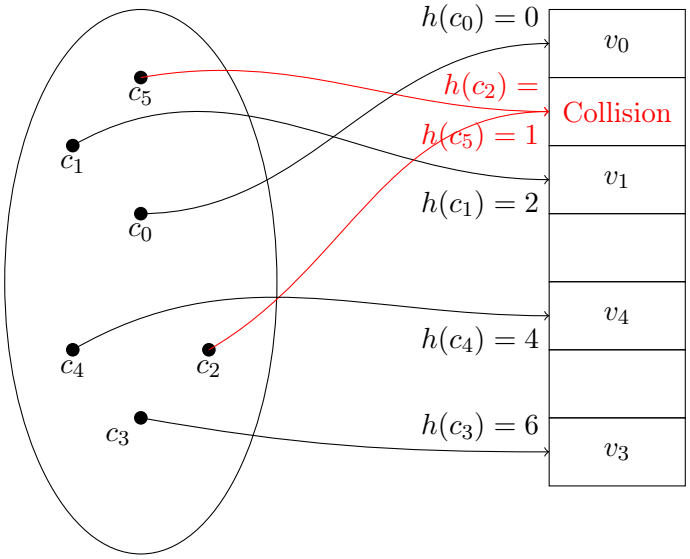


FIGURE 3 – Schéma illustrant une collision.

2.2 Fonction de hachage et table de hachage

Ici, h établit une correspondance entre l'ensemble C des clés et les valeurs v_i contenue dans une **table de hachage**.

2.2.1 Caractéristiques de la fonction h

Pour avoir une implémentation efficiente, une fonction de hachage doit :

- être facile à calculer en $O(1)$,
- avoir une distribution la plus uniforme possible (chaque clé a autant de chances d'être hachée vers l'une quelconque des m cases de la table, indépendamment des endroits où sont allées les autres clés).

Cette dernière condition est motivée par le souhait de minimiser le nombre de collisions. Cependant, il est souvent impossible en général de vérifier cette condition.

Il est parfois possible de connaître la distribution. Par exemple, supposons que les clés soient c réels aléatoires, distribués indépendamment et uniformément dans l'intervalle $0 \leqslant c < 1$. Dans ce cas, la fonction de hachage

$$h(c) = \lfloor cm \rfloor$$

satisfait à la condition du hachage uniforme simple.

2.2.2 Que faire lorsque les clés ne sont pas des `int` ?

On peut légitimement se poser la question de ce que peut être une fonction de hachage lorsque les clés ne sont pas des entiers. En effet, dans l'exemple traité sur le dépouillement d'une urne lors d'une élection, les clés du dictionnaires utilisées étaient des chaînes de caractères. Prenons l'exemple de la chaîne de caractères `pt`. Cette chaîne de caractères pourrait être interprété comme la paire d'entiers décimaux $(112, 116)$, puisque `p`$= 112$ et `t`$= 116$ dans l'ensemble des caractères ASCII. En l'exprimant dans la base 128 (car la table ASCII est originellement codée sur 7 bits), on obtient :

$$(112 \times 128) + 116 = 14452$$

Il est souvent possible de trouver une méthode aussi simple pour interpréter chaque clé comme un entier.

2.2.3 Probabilité de collisions

Étant donné un entier $i \in [\![0, m-1]\!]$, la probabilité que $h(c)$ (avec $c \in C$) soit égal à i est de $1/m$. La répartition de k clés étant uniforme, la probabilité qu'il n'y ait **pas** de collision est :

$$\frac{m \times (m-1) \times (m-2) \times \cdots \times (m-k+1)}{m^k} = \frac{m!}{m^k(m-k)!}$$

Ainsi la probabilité qu'il y ait au moins une collision est de :

$$\boxed{1 - \frac{m!}{m^k(m-k)!}}$$

Cette probabilité de collision augmente extrêmement rapidement. Par exemple, pour $m = 1\,000\,000$, la probabilité de collision dépasse 95% lorsque le nombre de clés atteint 2500.

2.2.4 Une fonction de hachage classique

Classiquement, pour créer une fonction de hachage, on fait correspondre une clé c avec l'une des m cases de la table de hachage en prenant le reste de la division de c par m. La fonction de hachage est donc

$$\boxed{h(c) = c \bmod m}$$

Cette fonction, vérifie bien $h : C \longrightarrow [\![0, m-1]\!]$.

Remarque 2

On peut toujours faire correspondre une clé c avec un nombre (voir paragraphe 2.2.2). Une telle correspondance existe nécessairement. En effet, tout objet est finalement enregistré avec une suite de bits, laquelle peut toujours être interprétée comme un nombre écrit en base deux.

Avec cette fonction de hachage, il faut choisir la taille m de façon intelligente pour essayer d'avoir une probabilité de distribution des valeurs de hachage $h(c)$, la plus uniforme possible. On se s'attardera pas sur cette façon de procéder.

3 Résolution des collisions

3.1 Introduction

Il existe plusieurs façons de résoudre les problèmes de collisions. Cependant, nous ne les développerons pas toutes. Nous avons vu que pour éviter le plus possible les collisions la première chose importante est d'avoir une fonction de hachage de « qualité ». Ensuite, pour stocker les données on

peut choisir des stratégies différentes en cherchant par exemple à répartir les collisions le plus possible. (Stratégie d'adressage ouvert par sondage linéaire ou quadratique [1]).

3.2 Résolution des collisions par chaînages

Une solution évidente est de stocker dans les cases mémoire de la table de hachage non pas la valeur v_i mais le couple (c_i, v_i). Ainsi, lors d'une collision, on ajoute le couple entré en collision avec ceux précédemment stockés sans prendre le risque d'écraser une information. Cette stratégie est appelé **résolution de collisions par chaînages** [2].

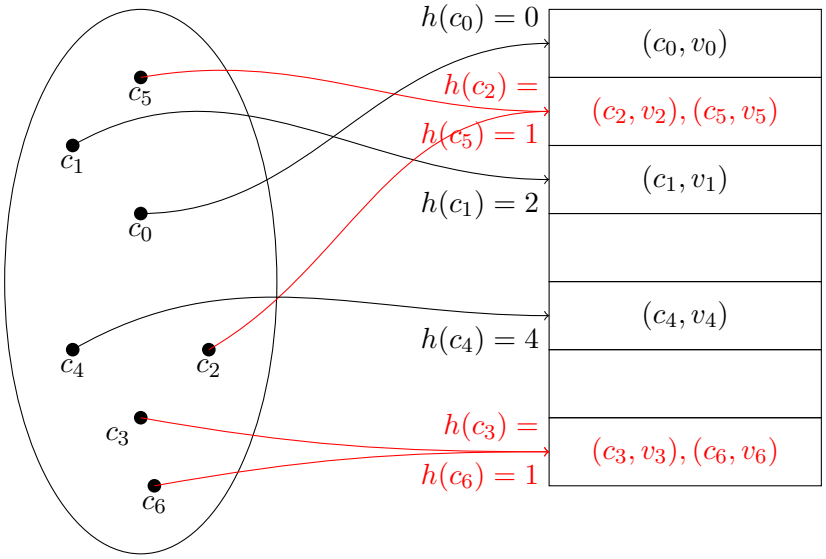


FIGURE 4 – Schéma illustrant deux collisions et gérés par chaînage

De cette façon pour connaître la valeur associé à une clé c_i par une recherche naïve :

- On calcule $h(c_i)$,
- On parcours la liste des couples (c_j, v_j) dans la case du tableau $h(c_i)$
- Lorsque l'on trouve la clé c_i lors du parcours, on retourne v_i.

4 Intérêt des tables de hachage par rapport aux listes

4.1 Cas de recherche d'un élément dans une liste

Dans une liste non triée, le temps de recherche d'un élément au pire des cas est $\mathcal{O}(n)$. Le pire des cas est quand l'élément n'est pas présent car dans ce cas il faut parcourir toute la liste avant de découvrir qu'il n'y ait pas.

```
1 def recherche_elem(L:list,elem):
2     for i in L:
3         if i==elem:
4             return True
5     return False
```

1. Introduction à l'algorithme : Cours et Exercices Edition Dunod 2ème édition
2. Ce nom provient du fait que les éléments sont stockés par liste chaînée, c'est-à-dire que pour accéder à élément $i+1$ d'une liste ℓ, il faut d'abord accéder à l'élément i qui contient la valeur $\ell[i]$ et le chemin d'accès à l'élément $i+1$. Sous Python, cette notion n'existe pas de façon native. On utilisera des listes classiques (la notion de chainage n'ayant donc pas de réelle sens ici).

4.2 Cas de recherche d'un élément dans une table de hachage sans collisions

Avec une table de hachage (on ne considère ici que les clés et on ne leur associe pas de valeur (si ce n'est elle-même)), le temps de recherche est en moyenne en $O(1)$ (on dit en moyenne car il faut gérer les cas de collisions). Prenons un exemple, pour illustrer cela. On dispose d'une fonction de hachage h me permettant de stocker des éléments dans une table de taille $m = 5$.

Les éléments dont on dispose sont :

- `"Sciences de l'ingénieur"` avec $h($`"Sciences de l'ingénieur"`$) = 2$,
- `"Informatique"` avec $h($`"Informatique"`$) = 4$,
- `"Physique"` avec $h($`"Physique"`$) = 1$.

La table de hachage obtenue est alors :

Table de hachage T

`"Physique"`	$h($`"Physique"`$) = 1$
`"Sciences de l'ingénieur"`	$h($`"Sciences de l'ingénieur"`$) = 2$
`"Informatique"`	$h($`"Informatique"`$) = 4$

FIGURE 5 – Table de hachage T

Ainsi, pour savoir si `"Lettres"` est présent dans la table de hachage, il suffit de faire :

$$T[h(\text{"}\texttt{Lettres}\text{"})] == \text{"}\texttt{Lettres}\text{"}$$

qui renverra dans notre cas ici `False`. La recherche (lorsqu'il n'y a pas de collisions) est donc en $O(1)$ avec une table de hachage.

C'est pour cela, qu'il faut utiliser la commande `cle in dico` pour savoir si une clé `cle` est présente dans le dictionnaire et non `cle in dico.keys()`.

4.3 Cas de recherche d'un élément dans une table de hachage avec collisions

Lorsqu'il y a collisions la complexité de recherche est dans le pire des cas en $O(1 + \ell)$ avec ℓ le nombre de couples (c_j, v_j) présent dans la case de la table de hachage $T[h(c)]$.

Si la fonction de hachage est uniforme (équiprobabilité que la valeur de hachage obtenue $h(c)$ soit un nombre de $[\![0, m-1]\!]$) alors l'espérance du nombre d'éléments présents dans la case $T[h(c)]$ est $\ell = \dfrac{n_c}{m}$ avec n_c le nombre de clés utilisés.

En plus du temps $\mathcal{O}(\ell)$ requis pour la recherche d'un élément au pire des cas dans la liste de $T[h(c)]$, il faut tenir compte du temps $\mathcal{O}(1)$ nécessaire pour le calcul de $h(c)$ et l'accès à $T[h(c)]$.

4.4 Intérêt pour les dictionnaires

La commande vue au paragraphe 1.1 `c in d` est donc très efficace et permet de savoir très rapidement si une clé `c` est présente dans un dictionnaire `d`.

TD Info - Table de hachage et dictionnaire

I - Révision sur l'utilisation des dictionnaires

Objectif

L'objectif de cet exercice est de réviser l'utilisation simple des dictionnaires sous Python

En *Python* les tables de hachage sont appelées *dictionnaires*. Dans un dictionnaire, on associe une valeur à une clé.
Les clés peuvent être de beaucoup de types : entiers, chaînes de caractères, tuple, fonction etc. (mais pas une liste python). En fait, on peut choisir un peu près ce que l'on veut comme clé tant que celle-ci est construite avec un objet non mutable. On peut créer un dictionnaire de plusieurs manières, par exemple :

- `d={"a" : 12, "blop" : "blip", 42 : [1,2,3]}`
- ```
  d={}
  d["a"]=12
  d["blop"]="blip"
  d[42]=[1,2,3]
  ```
- ```
  l=[("a",12),("blop",blip"),(42,[1,2,3])]
  d=dict(l)
  ```

Pour accéder à la valeur d'un dictionnaire `d` correspondant à la clé `k` on utilise la syntaxe `d[k]`. Par exemple, si `d` est le dictionnaire défini précédemment, `d["blop"]` vaut la chaîne de caractères `"blip"`.

Q1. Choisissez 5 mots de la langue française et créez un dictionnaire qui associe à chacun de ces mots sa traduction en anglais.

Q2. Ajoutez une entrée au dictionnaire de la question précédente (un nouveau mot et sa définition).

Pour savoir si une clé `k` est présente dans un dictionnaire `d` on peut utiliser la syntaxe `d.has_key(k)` (qui renvoie `True` ou `False`). On peut aussi utiliser la syntaxe `k in d` comme vu en cours.

Q3. Écrivez une fonction `ajoute(mot1, mot2, d)` qui prend en argument un mot en français, sa traduction en anglais et ajoute ces deux mots dans le dictionnaire d uniquement si `mot1` n'est pas une clé du dictionnaire (si `mot1` apparaît dans `d` la fonction ne fait rien).

Pour obtenir la liste de toutes les clés du dictionnaire, on utilise la syntaxe `d.keys()` qui renvoie une liste python contenant les clés.

Q4. Écrivez une fonction qui affiche à l'écran toutes les valeurs correspondant aux clés qui sont dans votre dictionnaire (ici, tous les mots en anglais qui apparaissent dans votre dictionnaire).

Pour supprimer une entrée du dictionnaire, on peut utiliser la fonction `del` (qui permet de supprimer une variable quelconque de manière générale). Ainsi, dans l'exemple initial, pour supprimer l'entrée correspondant à la clé `"blop"` on peut utiliser l'instruction `del(d["blop"])`.

Q5. Écrivez une fonction `delete(char, dict)` qui prend en argument un caractère `char` et un dictionnaire `dict` et supprime du dictionnaire toutes les entrées correspondant à des clés qui commencent par la lettre `char`.

II - Descriptif d'un graphe par un dictionnaire

Objectif

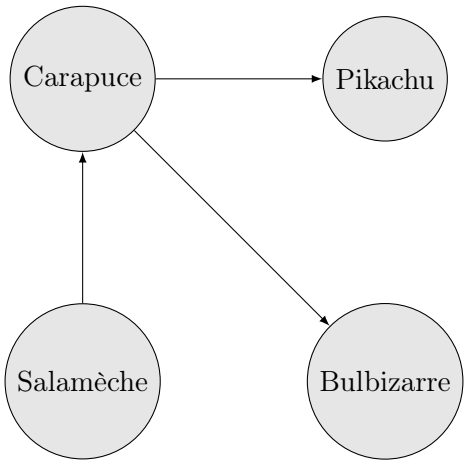
L'objectif de cet exercice est de rappeler l'écriture possible d'un graphe sous forme de dictionnaire d'adjacence.

On a vu en première année que l'on peut représenter les graphes

- soit par une matrice d'adjacence, c'est-à-dire une matrice M de taille $n \times n$ (où n est le nombre de nœuds du graphe) telle que ($M_{i,j} = 1 \iff$ s'il existe un arc reliant les nœuds d'indice i et j et 0 sinon).
- soit par une liste d'adjacence, c'est-à-dire une liste de longueur n telle que l'élément à la position i est la liste des indices des nœuds j pour lesquels il existe un arc allant de i vers j.

Dans les deux cas, l'encodage du graphe repose sur une indexation des nœuds par des entiers. On se propose d'étendre la notion de liste d'adjacence à celle de dictionnaire d'adjacence : un dictionnaire d'adjacence est un dictionnaire tel que :

- les clés correspondent aux identifiants des nœuds ;
- chaque clé i est associée à une valeur qui est la liste (éventuellement vide) des nœuds j tels qu'il existe un arc de i vers j.



Q1. Représenter sous la forme d'un dictionnaire d'adjacence le graphe ci-contre,tel qu'il existe un arc entre les pokemons i et j si le pokemon i est faible contre j. Attention : ce dictionnaire est censé avoir 4 clés.

Q2. On considère le dictionnaire d'adjacence `dico` obtenu à l'issue de la définition ci-dessous. Le graphe correspondant est il orienté ? Acyclique ?

```
dico={'A':'B','C':'B'}
dico['B']='D'
dico['A']='D'
dico['D']='C'
print(dico)
```

III - Codage d'une fonction de hachage

Objectif

L'objectif de cet exercice est de coder sa propre fonction de hachage et de vérifier une propriété.

On considère des clés sur un ensemble de 256 caractères (l'alphabet ASCII 8 bits par exemple) et l'on associe à chaque clé l'entier qu'elle représente en base 256. Ainsi, par exemple, puisque les caractères `B`, `l`, `o` et `p` correspondent aux valeurs 66, 108, 111 et 112 respectivement, la clé « Blop » est associée à l'entier :

$$66 \times 256^3 + 108 \times 256^2 + 111 \times 256 + 112 = 1\,114\,402\,672$$

Q1. Écrivez une fonction `str_to_int` qui prend en entrée une chaîne de caractères en ASCII 8 bits et renvoie l'entier associé.
Indication : On pourra utiliser la fonction `ord(c)` qui renvoie la valeur ASCII du caractère `c`.

Si l'on utilise la fonction de hachage

$$h : x \mapsto (x \mod 255)$$

pour tout mot x, si un mot y est obtenu à partir de x par permutation de ses lettres (mêmes lettres, même nombre d'occurrences, mais l'ordre est quelconque alors $h(x) = h(y)$.

Q2. Écrivez la fonction `hachage` associée à h qui prend en argument une chaîne de caractères, la convertit en entier puis la hache et vérifiez la propriété annoncée sur quelques exemples.

Q3. Expliquez effectivement pourquoi $h(x) = h(y)$ lorsque le mot y contient les mêmes lettres en même quantité mais dans un ordre différent.

IV - Création d'une table de hachage par tuple

Objectif

L'objectif de cet exercice est de créer sa propre table de hachage avec une fonction de hachage simple.

On souhaite créer notre propre table de hachage simplifiée à l'aide d'un tuple de taille m. Un élément de ce tuple est une liste de taille variable. Pour vos différents tests, on travaillera avec un tuple de taille $m = 3$.

Par exemple voici, une table de hachage représentée sous forme de tuple.

```
tab_hachage=(["chien","niche"],["maman"],["PTSI"])
```

La fonction de hachage utilisée sera :

$$h : x \mapsto (\texttt{hash(x)} \mod ???)$$

Avec `hash`, la fonction de hachage utilisée par Python. On précise que la fonction de hachage h renvoie l'indice du tuple où sera stocké le mot `x` dans la table de hachage. L'expression $a \mod b$ signifie le reste de la division euclidienne de a par b.

Q1. Par quoi faut-il remplacer les ??? dans la définition de la fonction h afin de permettre son utilisation comme fonction de hachage adaptée pour une table de hachage de taille m.

Q2. Créer une fonction `initialisation(m)` permettant d'initialiser un tuple de taille m composé de liste vide.

Q3. Créer une fonction `set_valeur(mot)` permettant d'ajouter à la table de hachage `mot`.

Voici une liste de mots que l'on cherche à stocker dans la table de hachage : `chien`, `niche`, `chat`, `adieu`, `veau`, `vache`, `cochon`.

Q4. Vous constatez que contrairement au dictionnaire, il ne vous ai donné ici que des mots à stocker et non pas une paire clé-valeur. Si l'on devait ici réfléchir avec cette structure, quelle serait la clé et la valeur ?

Q5. En utilisant vos fonctions `initialisation` et `set_valeur`, créer alors la table de hachage contenant les 7 mots demandés. Pour tester votre code, on rappelle que l'on fixe $m = 3$. Vérifiez que vous obtenez bien : `([’cochon’], [’chien’, ’niche’, ’chat’, ’vache’], [’adieu’, ’veau’])`.

Q6. Créer une fonction `test_presence(table,mot)` renvoyant un booléen permettant de savoir si `mot` est présent dans la table de hachage `table`. Vous veillerez à être le plus efficace possible. Vérifiez que votre fonction retourne le résultat attendu en effectuant une série de test.

V - Création et exploitation d'un dictionnaire

Objectif

L'objectif de cet exercice est d'exploiter un dictionnaire pour répondre à différentes questions. Le but pédagogique est de manipuler les dictionnaires.

1 Présentation

Un collectionneur de BDs référence celles-ci dans une liste Python, depuis qu'il a débuté sa collection . Chaque élément de la liste est un tuple contenant le nom de la BD, le numéro du tome de celle-ci, un booléen indiquant si cette BD est rare ou non (`True` pour rare et `False` sinon), et l'estimation du prix de celle-ci. La structure de la liste est donc la suivante :

```
mes_BDs=[(nom_BD1,numéro tome,Booléen,prix),(nom_BD2,numéro tome,Booléen,prix), ... ,
                                        etc.]
```

Cette liste est disponible dans le fichier `BD1.py`.

2 Travail demandé

Q1. Écrire le code permettant de créer le dictionnaire renseignant sur le nombre de BDs d'un titre. **Par exemple**, ce dictionnaire aura la forme suivante `dico_BD={"Astérix":20,"Tintin":40,etc...}`. On tient compte des doublons. On veut juste référencer ici de manière triviale la quantité de BDs que le collectionneur possède dans sa collection par le nom des BDs la composant.

Q2. À partir de la question précédente, déterminez combien il y a de noms de BDs différents

Q3. Déterminer le dictionnaire permettant de donner le montant maximal de chacun des titres de BD de la collection.

Q4. Quelle est la complexité de votre code précédent ? En notant m la quantité de titre différents composant la collection (voir question **Q2**), quelle serait la complexité de l'algorithme naïf pour obtenir le maximum de valeur de chacun des titres de BDs.

Q5. En utilisant éventuellement un ou des dictionnaires intermédiaires, déterminer un dictionnaire fournissant la valeur moyenne de chacun des titres composant l'ensemble de BDs de la collection.

Q6. Créez un dictionnaire, renseignant sur le nombre de BDs d'un titre sans tenir compte des doublons. Vous pouvez utiliser un dictionnaire intermédiaire au besoin.