EXERCICES DE COURS : TABLES ET FONCTIONS DE HACHAGE
== CORRECTION ==

EXERCICES DU CHAPITRE II : OPÉRATIONS PRISES EN CHARGE PAR LES TABLES DE HACHAGE

Exercice I.1 – Est-ce qu'une table de hachage est adaptée ?

Liste Python :

- On peut stocker une liste de couples (code_article, quantité) par ex. : [(10542, 3), (75601, 5), (43010, 12), ...].
- Pour récupérer la quantité du code 75601, on parcourt la liste et on teste chaque couple jusqu'à trouver code_article == 75601.

Dictionnaire Python :

- On stocke stock = {10542: 3, 75601: 5, 43010: 12, ...}
- Pour la quantité du code 75601 : stock[75601] → accès direct.

Complexité :

- Liste : recherche linéaire → en gros O(n) comparaisons, donc jusqu'à 10 000 comparaisons si 10 000 articles.
- Dictionnaire (table de hachage) : recherche moyenne O(1).

La table de hachage est clairement mieux adaptée pour ce type d'accès par identifiant.

Exercice I.2 – Clé ou valeur ?

Employés :

- Clé possible : Numéro de sécu : unique, stable, identifiant "officiel".
- Les autres infos (nom, date d'embauche…) dans la valeur.

Catalogue en ligne :

- Clé possible : Identifiant numérique interne (SKU, ID) : unique, stable.
- Nom, prix, etc. dans la valeur.

Agenda :

- Clé possible : Date/heure (timestamp), éventuellement combinée au lieu. Mais attention, plusieurs événements simultanés peuvent exister. Il faut donc soit stocker une liste d'événements pour une même date/heure, soit utiliser un autre identifiant unique et mettre date/heure dans la valeur.

Quizz 1 – Notion de clé & dictionnaire

1. C – Une clé sert à identifier une valeur.
2. C – La seconde affectation écrase la première ("Alice": 2).
3. B – L'intérêt principal : accès rapide par clé.
4. C – Le numéro de sécu fait une clé idéale.
5. Vrai – Plusieurs clés peuvent avoir la même valeur.
6. Faux – Les clés sont uniques : une clé n'apparaît qu'une fois.

Python 1 – Déduplication avec set/dict

```python
def ip_uniques(ips):
    vues = []
    resultat = []
    for ip in ips:
        # si ip n'a pas encore été vue
        if ip not in vues:
            vues.add(ip)
            resultat.append(ip)
    return resultat
```

Python 2 – Compter les occurrences de mots

```python
def compter_mots(mots):
    d = {}
    for mot in mots:
        if mot in d:
            d[mot] = d[mot] + 1
        else:
            d[mot] = 1
    return d
```

EXERCICES DU CHAPITRE II : OPÉRATIONS PRISES EN CHARGE PAR LES TABLES DE HACHAGE

Exercice II.1 – Identifier les opérations

1. « Vérifier si le client 1234 existe déjà » : recherche.
2. « Ajouter un nouveau client 5678 » : insertion.
3. « Supprimer le compte du client 1234 » : suppression.
4. « Mettre à jour l'adresse e-mail du client 5678 » : on fait d'abord une recherche de la clé 5678, puis on modifie la valeur associée. On peut voir ça comme « recherche + écriture de la valeur ».

Exercice II.2 – Tableau impossible

1. Chaînes de longueur 5 sur A–Z : Nombre de chaînes = 26^5 = 11 881 376
2. On n'utilise que 1000 prénoms distincts :
   Cases utilisées : 1000.
   Cases totales : 11 881 376.
   Proportion utilisée ≈ 1000 / 11 881 376 ≈ 0,000084 : quasi tout est vide.
3. Un tableau indexé par toutes les chaînes possibles gaspille presque toute la mémoire. Une table de hachage n'a besoin que d'un nombre de cases proportionnel au nombre de prénoms réellement stockés, pas au nombre total de chaînes possibles ce qui apporte un énorme gain de mémoire, tout en gardant des accès rapides.

EXERCICES DU CHAPITRE III : EXEMPLES D'APPLICATIONS

Exercice III.1 – Simuler une déduplication à la main

Flux : [A, B, C, A, D, B, E, C, F]

On maintient l'ensemble Vus :

- A : Vus = {A}
- B : Vus = {A, B}
- C : Vus = {A, B, C}
- A : déjà dans Vus → ignoré
- D : Vus = {A, B, C, D}
- B : déjà dans Vus → ignoré
- E : Vus = {A, B, C, D, E}
- C : déjà dans Vus → ignoré
- F : Vus = {A, B, C, D, E, F}

IP distinctes observées : {A, B, C, D, E, F}, dans l'ordre d'apparition : A, B, C, D, E, F.

Pour compter les visites par IP : au lieu de simplement stocker « vue ou pas vue », on stocke un compteur par IP (ex. dictionnaire compte[ip] += 1 à chaque occurrence).

Exercice III.2 – Pourquoi mémoriser les sommets visités ?

1. Sur un graphe avec cycles, le BFS risque de revisiter les mêmes sommets en boucle. On peut tourner indéfiniment ou au moins exploser le temps en revoyant sans arrêt les mêmes arêtes/sommets.

2. Avec la table `visited` :
    a. On va utiliser comme clés les sommets eux-mêmes (leurs identifiants).
    b. On ajoute un sommet u à `visited` quand on le découvre pour la première fois (au moment où on l'enfile dans la file, ou juste avant). Cela garantit qu'on ne le traite qu'une seule fois.

Python 4 – BFS avec ensemble visited

```python
from collections import deque

def bfs(adj, s):
    visites = set()
    file = deque()
    resultat = []

    # initialisation
    visites.add(s)
    file.append(s)

    while file:
        u = file.popleft()
        resultat.append(u)
        for v in adj[u]:
            # si v n'a pas été visité,
            # le marquer et l'ajouter à la file
            if v not in visites:
                visites.add(v)
                file.append(v)

    return resultat
```

EXERCICES DU CHAPITRE IV : IMPLÉMENTATION - IDÉES GÉNÉRALES

Exercice IV.1 – Pourquoi la liste n'est pas suffisante

La liste contient quelque chose comme : [ip1, ip2, ip3, ..., ip10000] (ou des couples (ip, info)).

Pour savoir si ip0 est présente, on parcourt la liste du début à la fin, en comparant ip0 avec chaque ip_i donc au pire il y a 10 000 comparaisons.

Avec une table de hachage :

- On calcule un hash de ip0, ce qui donne un indice dans la table.
- On ne regarde qu'un seul seau (ou une petite liste / quelques cases en adressage ouvert).

Le temps moyen est indépendant du nombre total d'IP (O(1)), tant que la table est bien dimensionnée.

Exercice IV.2 – Calculer des indices de hachage

Table de taille n = 10, h(k) = k mod 10.

1. h(7) = 7 h(12) = 2 h(25) = 5 h(30) = 0 h(44) = 4
2. Insertion dans le tableau (sans collisions pour l'instant) :

   Case 7 : 7 Case 2 : 12 Case 5 : 25 Case 0 : 30 Case 4 : 44

Pour ces valeurs-là, la distribution semble raisonnablement répartie (indices différents, pas de gros « amas »), même si l'échantillon est très petit.

Exercice IV.3 – Collisions inévitables

1. Principe des tiroirs : Si on a plus d'objets que de cases (plus de clés que de compartiments), alors au moins deux objets doivent partager une case. Donc si $|U| > n$, il y a forcément $x \neq y$ avec $h(x)=h(y)$.
2. Anniversaires : Il suffit de 23 personnes pour avoir ≈ 50% de chance d'un anniversaire commun, alors qu'il y a 365 « cases » possibles. L'intuition naïve « il faudrait beaucoup plus de 23 personnes pour ça » est fausse. Les collisions arrivent « rapidement ».
3. Conclusion : Même avec une table beaucoup plus grande que le nombre d'éléments, on ne peut pas espérer aucune collision. Il faut accepter les collisions et prévoir une stratégie pour les gérer.

Exercice IV.4 – Simuler une table avec chaînage

1. Table de 5 seaux, $h(k)=k \bmod 5$. Clés : 7, 12, 17, 22, 3, 8.
   - 7 mod 5 = 2 → seau 2 : [7]
   - 12 mod 5 = 2 → seau 2 : [7, 12]
   - 17 mod 5 = 2 → seau 2 : [7, 12, 17]
   - 22 mod 5 = 2 → seau 2 : [7, 12, 17, 22]
   - 3 mod 5 = 3 → seau 3 : [3]
   - 8 mod 5 = 3 → seau 3 : [3, 8]
2. État final :
   0 : [] 1 : [] 2 : [7, 12, 17, 22] 3 : [3, 8] 4 : []
3. Nombre moyen d'éléments par seau : Total = 6, n = 5 donc une moyenne = 6/5 = 1,2.
4. Pour rechercher la clé 22, on calcule h(22)=2 donc on va au seau 2. On parcourt la liste [7, 12, 17, 22] dans l'ordre jusqu'à trouver 22.

Exercice IV.5 – Sondage linéaire à la main

1. 10 : h=3 → case 3 est libre → on met 10 en 3.
   24 : h=3 → case 3 occupée (10) → on teste 4 → libre → 24 en 4.
   31 : h=3 → case 3 occupée, 4 occupée → on teste 5 → libre → 31 en 5.
   45 : 45 mod 7 = 3 → 3,4,5 occupées → on teste 6 → libre → 45 en 6.
   18 : 18 mod 7 = 4 → 4 occupée → on teste 5,6 occupées → on teste 0 → libre → 18 en 0.
2. État final (par indices 0..6) :
   0 : 18 1 : vide 2 : vide 3 : 10 4 : 24 5 : 31
   6 : 45
3. Recherche 18 :
   h(18) = 4 → on teste 4 (24, pas 18) → 5 (31) → 6 (45) → 0 (18) → trouvé à 0.
4. Recherche 11 :
   h(11) = 4 → on teste 4 (24) → 5 (31) → 6 (45) → 0 (18) → 1 (vide).

En tombant sur une case vide dans une zone qui a déjà été sondée, on peut conclure que 11 n'est pas dans la table (s'il y avait été inséré, on l'aurait rencontré avant de tomber sur cette case vide).

Exercice IV.6 – Bonne ou mauvaise fonction de hachage ?

1. h(k) = 0 (pour tous k) : Toutes les clés vont dans le même compartiment, c'est catastrophique : temps de recherche O(n).
2. h(k) = k mod 1000, salaires multiples de 1000 : Si salaire = 23 000, 45 000, etc., on a toujours k mod 1000 = 0 donc toutes les clés tombent en 0, c'est encore catastrophique.
3. h(k) = (a·k + b) mod 1000 : Si toutes les données sont encore strictement des multiples de 1000, le problème persiste (car a·k serait multiple de 1000).
   Mais si on a une partie de données variées, c'est un peu mieux que h_2 car on casse la structure simple « multiples de n ». Cela Reste néanmoins fragile si les données sont très régulières.
4. h(chaine) = hash polynomial mod n, n premier : Avec des chaînes variées et un n bien choisi (premier, pas lié à la base), c'est une fonction raisonnablement bonne : peu de structure exploitable, collisions raisonnablement réparties.

Quizz 2 – Fonction de hachage & collisions

1. B – Déterministe et rapide, avec la même clé donc même résultat.
2. C – Collision = deux clés différentes dans le même compartiment.
3. B – Plus de clés que de compartiments donc au moins une collision.
4. B – h(k) = 0 met tout dans le même compartiment.
5. Faux – Principe des tiroirs : si l'univers des clés est plus grand que le nombre de cases, collisions inévitables.

Quizz 3 – Chaînage & adressage ouvert

1. B – Une liste (ou chaîne) des clés qui y tombent.
2. C – On teste d'autres cases selon une séquence déterminée.
3. C – Sondage linéaire crée des « grappes » de cases occupées.
4. Vrai – On a des marqueurs DUMMY pour les cases supprimées.
5. Vrai – Plus α augmente, plus les listes s'allongent en chaînage.

Python 4 – Mini table de hachage avec chaînage

```python
def creer_table(n):
    # crée une table avec n seaux vides
    return [[] for _ in range(n)]

def h(k, n):
    return k % n

def inserer(table, cle):
    n = len(table)
    i = h(cle, n)
    seau = table[i]
    # insérer "cle" si elle n'est pas déjà présente
    if cle not in seau:
        seau.append(cle)

def rechercher(table, cle):
    n = len(table)
    i = h(cle, n)
    seau = table[i]
    # renvoyer True si cle est dans le seau, False sinon
    for x in seau:
        if x == cle:
            return True
    return False
```

Python 6 – Simuler l'adressage ouvert (sondage linéaire)

```python
def creer_table(n):
    return [None] * n

def h(k, n):
    return k % n

def inserer(table, cle):
    n = len(table)
    i = h(cle, n)
    # sonder jusqu'à trouver une case vide ou la clé déjà présente
    for _ in range(n):   # on limite à n sondes
        if table[i] is None:
            table[i] = cle
            return
        elif table[i] == cle:
            return   # déjà présent
        i = (i + 1) % n   # case suivante

tab = creer_table(5)
for k in [1, 6, 11]:
    inserer(tab, k)
```

EXERCICES DU CHAPITRE V : FACTEUR DE CHARGE ET PERFORMANCES

Exercice V.1 – Calculer et interpréter α

1. 20 éléments : α = 20/100 = 0,2 : très peu chargé, très bon.
2. 50 éléments : α = 0,5 : table à moitié pleine, bon compromis.
3. 80 éléments : α = 0,8 : table très chargée, risque de longues listes (chaînage) ou de sondages longs (adressage ouvert).

Exercice V.2 – Admissible ou pas ?

1. Chaînage, n=1000, α≈0,5, bonne fonction : Longueur moyenne des listes ≈ 0,5 donc accès moyen O(1). Comportement proche de constant.
2. Chaînage, n=1000, toutes les clés dans le même seau : Une liste de taille ≈ 1000 donc recherche dans ce seau = O(1000) = O(n). On est dans le pire cas.
3. Adressage ouvert (double hachage), n=10 000, α≈0,7 : Temps moyen ≈ O(1 / (1−α)) ≈ O(1 / 0,3) ≈ O(3) : constant raisonnable. Acceptable.
4. Adressage ouvert (sondage linéaire), n=10 000, α≈0,95 : 1/(1−α) ≈ 1/0,05 ≈ 20 : très gros cluster, sondages très longs. Recherche peut se rapprocher de O(n).

Exercice V.3 – Faut-il redimensionner ?

1. 400 éléments : α = 400/1000 = 0,4 donc pas besoin de redimensionner (très confortable).
2. 800 éléments : α = 800/1000 = 0,8 donc au-dessus du seuil 0,7 ; on redimensionne.
3. Redimensionnement à n = 2000 avec 800 éléments : Nouveau α = 800/2000 = 0,4 donc on revient à une table peu chargée.
   On a payé un coût ponctuel élevé (réinsertion de 800 éléments), mais ce coût est rare par rapport au nombre total d'opérations, donc le coût amorti d'une insertion reste O(1) sur le long terme.

Quizz 4 – Facteur de charge & performances

1. B – α = (nombre d'éléments)/n.
2. B – Avec adressage ouvert, α proche de 1 → sondages longs → tendance vers O(n).
3. C – Coût ponctuel élevé, mais meilleur temps moyen ensuite.
4. Vrai – Stratégie classique : redimensionner quand α dépasse un seuil.
5. Vrai – C'est le principe du coût amorti.

EXERCICES DU CHAPITRE VI : FONCTIONS DE HACHAGE UNIVERSELLES

Exercice VI.1 – Pourquoi une seule fonction ne suffit pas ?

1. Une fonction de hachage fixe h : Un adversaire peut choisir un jeu de données « malveillant » avec énormément de collisions (par ex. toutes les clés envoyées dans le même seau). Donc on ne peut pas garantir de bonnes perfs pour tous les jeux de données possibles.
2. (A) « il existe une fonction bonne pour ce jeu » vs (B) « un tirage au hasard est bon en moyenne » : (A) est existentiel : on ne dit pas comment trouver la bonne fonction.
   (B) : si on choisit au hasard, on obtient typiquement de bonnes performances (espérance contrôlée).
3. En pratique : On choisit une fonction aléatoirement dans une famille. On obtient alors un comportement en moyenne bon sur n'importe quelles données, même si quelqu'un essaie de construire un jeu pathologique.

Exercice VI.2 – Cette famille est-elle universelle ?

1. $\mathcal{H}_1$ = toutes les fonctions U → {0, …, n−1} : Pour x ≠ y, et h choisi uniformément parmi toutes ces fonctions, les paires (h(x), h(y)) sont réparties uniformément parmi n² possibilités. Le nombre de paires avec h(x) = h(y) est n (toutes les paires (i, i)). La probabilité est donc n / n² = 1/n donc $\mathcal{H}_1$ est universelle.

2. $\mathcal{H}_2$ = fonctions constantes $h_i(k) = i$: Pour toute clé k, P[h(k) = i] = 1/n (la fonction constante choisie est uniforme). Mais pour x≠y, h(x) = h(y) toujours, car toute fonction de $\mathcal{H}_2$ est constante. Donc P[h(x) = h(y)] = 1, pas ≤ 1/n → $\mathcal{H}_2$ n'est pas universelle.

Exercice VI.3 – Hachage d'adresses IP : calcul concret

1. $h_a(x) = 2\cdot10 + 5\cdot0 + 3\cdot5 + 1\cdot3 = 20 + 0 + 15 + 3 = 38$; 38 mod 11, reste 5 donc $h_a(x) = 5$. $h_a(y) = 2\cdot10 + 5\cdot0 + 3\cdot5 + 1\cdot4 = 20 + 0 + 15 + 4 = 39$; 39 mod 11 reste 6 donc $h_a(y) = 6$.
2. Pas de collision : 5 ≠ 6.
3. Nombre d'opérations : 4 multiplications, 3 additions, un modulo (constante, indépendante de la taille de la table) donc O(1).
4. Le fait que 11 soit premier : évite certaines structures de sous-groupes qui pourraient créer des collisions systématiques (on garantit une meilleure « répartition » des valeurs possibles).

Exercice VI.4 – Pourquoi choisir n premier, > 255 ?

1. n > 255 : L'idée clé dans la preuve de l'universalité de cette famille, c'est :
   - on prend deux adresses distinctes x ≠ y,
   - on regarde leur différence coordonnée par coordonnée : $\delta_i = x_i - y_i$.

Comme chaque coordonnée est entre 0 et 255, δ_i est un entier entre −255 et 255. Maintenant, on travaille modulo n. Si n > 255, alors un entier δ tel que |δ| ≤ 255 ne peut être congruent à 0 modulo n que si δ = 0. Donc si $x_i \neq y_i$ et n > 255, alors $\delta_i = x_i - y_i$ n'est pas congruent à 0 modulo n.

Autrement dit, dès que deux IP sont différentes, il existe au moins une coordonnée j telle que $(x_j - y_j)$ n'est pas congruent à 0 modulo n.

De plus quand on réécrit l'égalité $h_a(x) = h_a(y)$: $a_1\cdot(x_1 - y_1) + \ldots + a_4\cdot(x_4 - y_4) \equiv 0 \pmod n$, on veut montrer qu'il n'y a au plus qu'un seul choix possible de a_j quand on fixe les autres, donc une probabilité ≤ 1/n. Pour ça, il faut que l'un des $(x_j - y_j)$ soit non nul modulo n, sinon tout s'écroule.

2. n premier : Dans $\mathbb{Z}/n\mathbb{Z}$, si n est premier, tout entier non nul est inversible. C'est ce qui a permis de montrer qu'il n'y a au plus qu'un seul choix possible de a_j quand on fixe les autres. Cela donne une structure algébrique meilleure, limite les solutions triviales des équations qui conduisent à des collisions systématiques.
3. Lien avec IV.6.6 : On avait déjà dit qu'il est conseillé de choisir n premier et loin des puissances de 2 ou de 10 pour éviter que certaines régularités des données ne soient « alignées » avec n. Ici, c'est exactement la même idée.

Exercice VI.5 – Longueur moyenne d'un seau avec hachage universel

1. On a $X_y = 1$ si h(y) = h(x), 0 sinon. Longueur du seau : $\sum_{y \in S} X_y$
2. Par universalité, pour y ≠ x : $E[X_y] = P[h(y) = h(x)] \leq 1/n$.
3. Espérance de la longueur :
$$E\left[longueur\right] = E\left[\sum_{y \in S} X_y\right] = \sum_{y \in S} E\left[X_y\right] = m \cdot E\left[X_y\right] \leq m \cdot \left(1/n\right) = \alpha$$
4. Interprétation : en moyenne, les listes (seaux) ont une longueur ≤ α. Si α est borné (par ex. ≤ 2), alors une recherche infructueuse se fait en O(1 + α), donc en temps constant en pratique.

Quizz 5 – Hachage universel

1. C – Pour x ≠ y, P[h(x) = h(y)] ≤ 1/n.
2. D – But : garantir des performances bonnes en moyenne, même contre des données adverses.

EXERCICES DU CHAPITRE VII : DICTIONNAIRES PYTHON

Exercice VII.1 – Table d'indices vs table compacte

1. Table d'indices : Chaque case contient soit EMPTY, soit DUMMY, soit un index vers la table compacte. Elle est « clairsemée » car beaucoup de cases restent EMPTY pour garder un faible facteur de charge.
2. Table compacte d'entrées (mode combiné) : Chaque entrée stocke : (hash, pointeur vers la clé, pointeur vers la valeur). Elle est dense : toutes les cases 0 … (n - 1) de cette table sont occupées par des entrées valides (ou presque).
3. Ordre d'insertion : Les entrées sont ajoutées à la fin de la table compacte donc l'ordre des indices de cette table correspond à l'ordre d'insertion. Quand on parcourt le dictionnaire, on parcourt la table compacte dans l'ordre donc on retrouve l'ordre d'insertion.
4. Localité mémoire : Les entrées sont stockées de manière contiguë ce qui entraine de bonnes propriétés de cache. Mieux que des listes chaînées ou des pointeurs dispersés en mémoire.

Exercice VII.2 – Comprendre i = h & (m – 1)

1. $h_1 = 13$: 13 mod 8 = 5 ; 13 en binaire = 1101, 7 = 0111 ; 1101 & 0111 = 0101 = 5.
   $h_2 = 42$: 42 mod 8 = 2 ; 42 = 101010, 7 = 000111 ; 101010 & 000111 = 000010 = 2.
   $h_3 = 57$: 57 mod 8 = 1 ; 57 = 111001, 7 = 000111 ; 111001 & 000111 = 000001 = 1.
2. Dans les trois cas, h % 8 == h & 7. Conclusion : Quand m est une puissance de 2, h & (m - 1) revient à prendre les k bits de poids faible de h, ce qui est équivalent à h % m.
3. Intérêt : L'opération & (et binaire) est souvent plus rapide que le modulo général. Implémentation simple au niveau machine, sans division.

Exercice VII.4 – Insertion avec EMPTY et DUMMY

1. d["c"] = 3 (h("c") compressé = 1) :
   - On sonde 1 : occupé (index 0 -> "a").
   - On sonde 2 : DUMMY donc on mémorise cette case comme meilleure place libre pour l'instant.
   - On sonde 3 : occupé ("b").
   - On sonde 4 : EMPTY donc fin du sondage.
   - Python va insérer "c" en réutilisant la première case DUMMY (2).
   - On ajoute une nouvelle entrée dans la table compacte (index 2 : ("c", 3)).
   - Table d'indices : case 2 → 2.

Adresse mémoire	**0**	**1**	**2**
	(hash("a"), ptr_"a", ptr_1)	(hash("b"), ptr_"b", ptr_2)	**(hash("c"), ptr_"c", ptr_3)**

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Valeur	-	0	**2**	1	EMPTY	EMPTY	EMPTY	EMPTY

2. d["b"] = 5 (mise à jour)
   - h("b") compressé = 3.
   - On sonde 3 → index 1 → on retrouve "b".
   - Mise à jour de la valeur (pointeur vers 5) dans l'entrée 1.
   - Indices inchangés.

Adresse mémoire	**0**	**1**	**2**
	(hash("a"), ptr_"a", ptr_1)	(hash("b"), ptr_"b", **ptr_5**)	**(hash("c"), ptr_"c", ptr_3)**

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Valeur	-	0	**2**	1	EMPTY	EMPTY	EMPTY	EMPTY

3. del d["a"] :
   - On cherche "a" : Supposons que h("a") compressé fasse qu'on retombe sur l'indice 4, puis EMPTY, etc., jusqu'à trouver la case 1 → index 0 → "a".
   - On marque la case de la table d'indices où se trouvait "a" comme DUMMY.
   - L'entrée 0 de la table compacte n'est plus utilisée (mais Python ne la compacte pas forcément tout de suite).

Adresse mémoire	**0**	**1**	**2**
	~~**(hash("a"), ptr_"a", ptr_1)**~~	(hash("b"), ptr_"b", **ptr_5**)	**(hash("c"), ptr_"c", ptr_3)**

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Valeur	-	**DUMMY**	**2**	1	EMPTY	EMPTY	EMPTY	EMPTY

4. d["d"] = 4 (h("d") compressé = 1)
   - On sonde 1 : DUMMY → possible emplacement.
   - On sonde 2 : 2 ("c") → occupé.
   - On sonde 3 : 1 ("b") → occupé.
   - On sonde 4 : EMPTY → fin.
   - Python insère "d" en réutilisant la première DUMMY rencontrée (1).

- Nouvelle entrée dans la table compacte (index 3 : ("d", 4)).
- Table d'indices : case 1 → 3.

0	1	2	3
~~(hash("a"), ptr_"a", ptr_1)~~	(hash("b"), ptr_"b", ptr_5)	(hash("c"), ptr_"c", ptr_3)	(hash("d"), ptr_"d", ptr_4)

i	0	1	2	3	4	5	6	7
Valeur	-	3	2	1	EMPTY	EMPTY	EMPTY	EMPTY

Exercice VII.5 – Coût moyen vs pire cas dans dict

1. Dico classique, 10 000 entrées, α ≈ 0,6 : Table bien dimensionnée, bonne distribution donc opérations en O(1) en moyenne.
2. Dico attaqué, sans hachage randomisé : Un adversaire peut fabriquer des chaînes qui ont toutes le même hash pour engendrer des collisions massives. Les opérations peuvent devenir O(n) dans le pire cas.
3. Dico attaqué, mais avec hachage randomisé + redimensionnement : L'adversaire ne connaît pas la fonction de hachage exacte. Très difficile de provoquer des collisions systématiques donc le coût moyen reste O(1) (même si quelques cas isolés peuvent être plus longs).
4. Redimensionnement : Une insertion donnée peut faire re-hasher toutes les entrées donc coût O(n) pour cette insertion. Mais ces événements sont rares donc ils sont amortis sur beaucoup d'insertions, le coût moyen par insertion reste O(1).

Quizz 6 –dict Python

1. B – La table d'indices est sparse (beaucoup de cases vides).
2. C – En pratique, les clés doivent être hashables et immuables.
3. Vrai – Hash randomisé pour str/bytes (entre exécutions).
4. Faux – Depuis Python 3.7, l'itération respecte l'ordre d'insertion, pas l'ordre de hachage.

Python 5 – Clés hashables ou non

```
d = {}

d[42] = "entier"           # OK : int est hashable
d["hello"] = "chaine"      # OK : str est hashable
d[(1, 2, 3)] = "tuple"     # OK : tuple immuable est hashable

liste = [1, 2, 3]
# d[liste] = "liste mutable"   # ERREUR : list n'est pas hashable

ens = {1, 2, 3}
# d[ens] = "ensemble mutable"  # ERREUR : set n'est pas hashable
```

Corrections possibles : utiliser une version immuable :

```
d[tuple(liste)] = "liste immuable"
d[frozenset(ens)] = "ensemble immuable"
```

Explication : `list` et `set` sont mutables donc leur hash pourrait changer. Python interdit leur usage comme clé. Les `tuple` et `frozenset` sont immuables donc ils sont hashables donc utilisables comme clés.