Implémentation de Tableaux associatifs

PSI⋆

On rappel le type abstrait des tableaux associatifs :

- **ajout** : association d'une nouvelle valeur à une nouvelle clef ;
- **modification** : association d'une nouvelle valeur à une ancienne clef ;
- **suppression** : suppression d'une clef ;
- **recherche** : détermination de la valeur associée à une clef, si elle existe.

Le but, après une première est de réimplémenter la structure de dictionnaire existante en Python de deux manières différentes.

I Listes de couples

L'ensemble des clés possibles seront les chaînes de caractère et les entiers.

On décide de représenter ici un tableau associatif par une liste de couples (**clé**, **valeur**). Les clés doivent apparaître de manière unique. Par exemple :

```
[("Nadine",12), ("Cuisine",[23,34,56]), ("Tajine","Allan"), (0,23)]
```

Exercice 1. *Recherche*

Implémenter une fonction `lc_get` qui prend en entrée un tableau associatif, une clé et une valeur par défaut et qui renvoie :

- la valeur associé à la clé si elle existe dans le tableau
- la valeur par défaut sinon.

Par exemple :

```
T = [("Nadine",12), ("Cuisine",[23,34,56]), ("Tajine","Allan"), (0,23)]
lc_get(T,"Nadine", None) # Renvoie 12
lc_get(T,"Mezzanine", None) # Renvoie None
```

Donnez la complexité de la fonction.

Exercice 2. *Suppression*

Implémenter une fonction `lc_del` qui prend en entrée un tableau associatif et une clé et qui supprime la clé dans le tableau, si elle existe.

On pourra utiliser la fonction `L.pop(i)` qui permet de supprimer le $i^{ème}$ élément de la liste `L` en O(`len`(`L`))

Donnez la complexité de la fonction.

Exercice 3. *Ajout et Modification*

Implémenter une fonction `lc_set` qui prend en entrée un tableau associatif, une clé et une valeur et qui :

— modifie la valeur associée à la clé si la clé est déjà présente dans le tableau.
— ajoute le couple (clé,valeur) dans le tableau si la clé n'est pas présente.

Donnez la complexité de l'opération et prouvez sa correction.

II Fonction de hachage

L'ensemble des clés possibles seront les chaînes de caractère.

On considère la fonction de hachage suivante :

$$\mathcal{F}(k) = \sum_{i=0}^{\mathrm{len}(k)-1} \mathrm{ord}(k[i]) * 256^i$$

Il s'agit de l'écriture en base 256 de la chaîne de caractère ASCII. C'est une fonction bijective de l'ensemble des chaînes ASCII vers les entiers naturels.

Exercice 4. *Etude de* F

1. Déterminer l'espace d'arrivée de $\mathcal{F}$.
2. Écrire la fonction `hash` qui implémente $\mathcal{F}$.
3. Quelle est la complexité de `hash` ?

Exercice 5. *Etude de* F_n

On considère l'ensemble des fonctions de hachages suivant :

$$\forall n \in \mathbb{N}, \mathcal{F}_n(k) = \mathcal{F}(k) \,\%\, n$$

1. Déterminer l'espace d'arrivée de $\mathcal{F}_n$
2. On supposera que $\mathbb{P}(\mathcal{F}_n(s) = k) = \frac{1}{n}$. Quelle est la probabilité que deux clés aient la même image pour $\mathcal{F}_n$?
3. Peut on utiliser $\mathcal{F}_n$ comme fonction de hachage ? Justifier.
4. Écrire la fonction `hashn` qui implémente $\mathcal{F}_n$.
5. Quelle est la complexité de `hashn` ?

III Table de hachage avec résolution des collisions par chaînage

On décide de représenter ici un tableau associatif par une table de hachage. Une table de hachage sera implémenté en Python comme une liste de trois éléments :

— T un tableau numpy possédant dans chaque case une liste de couples clé, valeur.
— m le nombre d'éléments dans la table
— n déterminant la fonction F_n de hachage utilisée.

Le tableau et les deux entiers seront stockés dans une liste de trois éléments.

Par exemple pour le tableau associatif sur $\mathcal{F}(5)$:

— `"Nadine"` $\longrightarrow$ 12
— `"Cuisine"` $\longrightarrow$ `[12,23,45]`

— "Tajine" → "Allan"
— "Mezzanine" → 23

```
import numpy as np
T_tmp = np.array([[("Cuisine", [12,23,45]),("Mezzanine",23)],
                  [("Nadine",12)], [], [("Tajine","Allan")], [] ] )  # Données
T = [T_tmp, 4, 5] # Données, nombre d'éléments et numéro de la fonction de hachage
```

La création d'un tableau numpy de taille n contenant des listes vides devra se faire comme ceci :

```
import numpy as np
T = np.empty(n,dtype=object)
for i in range(len(T)) :
    T[i] = []
```

L'accès et la modification dans un tableau numpy se font comme pour les listes. Idem pour la taille :

```
a = T[i] + 5 # Accès
T[i] = 4 # Modification
n = len(T) # Taille
```

Exercice 6. *Création*

Implémenter une fonction `th_new` qui ne prend aucune entrée et qui renvoie un tableau associatif vide dont la fonction de hachage est F_2.

Exercice 7. *Recherche*

Implémenter une fonction `th_get` qui prend en entrée un tableau associatif, une clé et une valeur par défaut et qui renvoie :

— la valeur associée à la clé si elle existe dans le tableau
— la valeur par défaut sinon.

```
import numpy as np
T_tmp ~ np.array([ [("Cuisine", [12,23,45]),("Mezzanine",23)],
                  [("Nadine",12)], [], [("Tajine","Allan")], [] ])
T ~ [T_tmp, 5]
th_get(T,"Nadine", None) # Renvoie 12
th_get(T,"Grenadine", None) # Renvoie None
```

Quelle est la complexité au pire cas de votre algorithme ?

Exercice 8. *Complexité de la Recherche en moyenne.*

On suppose maintenant pour commencer n fixé et on s'intéresse à la complexité *en moyenne* d'une recherche dans une table de hachage implémenté à l'aide de la fonction de hachage $\mathcal{F}_n$.

1. On fixe une table de hachage $\mathcal{T}$ qu'on suppose remplie avec m éléments de telle manière que l'alvéole i de la table contiennent k_i couples clé valeurs.
   (a) Que dire de $\sum_{i=0}^{n-1} k_i$?
   (b) Soit x une clé de l'espace de clé. Montrer que la complexité au pire de la recherche de x dans $\mathcal{T}$ et la même pour deux clés ayant le même hash. On la notera $\mathcal{C}_i(\mathcal{T})$. Calculer cette complexité.

(c) On note p_i la probabilité que pour $x \in \mathrm{U}$ tiré uniformément, $\mathcal{F}_n(x) = i$. On défini la complexité en moyenne de la recherche de x dans $\mathcal{T}$ comme étant :

$$\mathcal{C}(\mathcal{T}) = \sum_{i=0}^{n-1} p_i \, \mathcal{C}_i(\mathcal{T}))$$

Calculer $\mathcal{C}(\mathcal{T})$.

2. On note $\mathbb{T}_m$ l'ensemble des tables remplies avec m éléments. On suppose que toutes les tables de taille m ont la même probabilité d'apparaître en pratique et on défini donc :

$$\mathcal{C}(m) = \sum_{\mathcal{T} \in \mathbb{T}_m} \frac{1}{|\mathbb{T}_m|} \mathcal{C}(\mathcal{T})$$

la complexité en moyenne de la recherche sur les tables de taille m. Calculez $\mathcal{C}(m)$.

On suppose maintenant que n n'est plus fixe mais évolue en fonction du nombre d'éléments contenus dans le tableau associatif. Donnez une condition à vérifier sur le facteur de remplissage de la table, pour que la complexité en moyenne de la recherche, $\mathcal{C}(m)$, soit en $\mathrm{O}(1)$ quelque soit m.

Exercice 9. *Suppression*

Implémenter une fonction `th_del` qui prend en entrée un tableau associatif et une clé et qui supprime la clé dans le tableau, si elle existe.

Exercice 10. *Redimensionnement*

L'ajout dans la table de hachage se fait tant que le facteur de remplissage est plus petit que 0.5. Au delà, on changera la fonction de hachage pour doubler le nombre n utilisé.

Implémenter une fonction `th_extend` qui prend en entrée un tableau associatif ayant comme fonction de hachage F_n et renvoie un nouveau tableau associatif contenant les mêmes valeurs et dont la fonction de hachage est maintenant F_{2n}.

Exercice 11. *Ajout et Modification*

Implémenter une fonction `th_set` qui prend en entrée un tableau associatif, une clé et une valeur et qui :

— modifie la valeur associée à la clé si la clé est déjà présente dans le tableau.
— ajoute le couple (clé,valeur) dans le tableau si la clé n'est pas présente, tout en maintenant un facteur de compression inférieur à 0.5, quitte à doubler n.