Dictionnaires et tables de hachage

1) Dictionnaires

Essentiellement, un dictionnaire est un ensemble E de couples (*clé,valeur*), où **valeur** peut être n'importe quel objet (un entier, une tuple, une liste, une fonction, ...) et **clé** est un **objet non mutable**, typiquement des nombres (entiers ou flottants), des chaînes de caractères ou des tuples.

On peut définir un dictionnaire soit directement :

```
dico = {cle1 :  v1 , cle2 :  v2 , cle3 :  v3}
```

```
dico_vide = {}  # on peut aussi utiliser la fonction dict() qui crée un dictionnaire vide
```

soit à partir d'une liste de couples (*clé,valeur*) :

```
couples = [(cle,v1),(cle2,v2),(cle3,v3)] ; dico = dict(couples)
```

Une fois défini, on peut récupérer les éléments du dictionnaire : `dico[cle2]` vaut `v2`.

- Longueur : `len(dico)`

- Test d'appartenance : `(x in dico)` ; la complexité est en $O(1)$.

- Ajout d'un élément à un dictionnaire : `dico[cle0] = v0` ; la complexité est en $O(1)$.

- Suppression d'un couple (*clé,valeur*) par sa clé : `del dico[cle1]` ; la complexité est en $O(1)$.

- Itérations :

`for c in dico` ... permet d'itérer sur **les clés d'un dictionnaire.**

(variante : `for c in dico.keys()`)

`for v in dico.values()` ... permet d'itérer sur les valeurs d'un dictionnaire.

`for k,v in dico.items()` permet d'itérer sur les couples (*clé,valeur*) d'un dictionnaire.

Remarque : Le coût de `dico.values()` et `dico.items()`sont linéaire en $O(n)$.

Remarque : D'un certain point de vue, le tableau standard `[3,2,1]` s'apparente au dictionnaire où les éléments sont indexés par les entiers 0, 1 et 2, c'est-à-dire `{0 :  3 , 1 :  2 , 2 :  1}`.
Mais l'implémentation physique est très différente, car dans le cas d'un tableau, PYTHON utilise 3 cases mémoires consécutives (donc naturellement ordonnées), alors que dans le cas d'un dictionnaire, les cases mémoires appartiennent en fait à un tableau plus grand qui correspond à la gestion par PYTHON des tables de hachage, et ne sont pas des mémoires consécutives dans l'ordinateur.

L'avantage majeur des dictionnaires (cf paragraphe 2) est :

- de garantir un coût d'accès en $O(1)$, (il s'agit d'une complexité moyenne en fait) ; de même le test d'appartenance `(x in dico)` est de complexité $O(1)$.
- d'être une structure dynamique (on peut ajouter ou supprimer des éléments) et ces opérations sont effectuées à temps constant, c'est-à-dire en $O(1)$ (en moyenne). Dans les tableaux, le coût est en $O(n)$ ...

2) Exemples d'utilisation des dictionnaires

Les éléments d'un dictionnaire ne sont pas ordonnés (contrairement aux éléments d'une liste ou d'un tableau). En revanche, on peut déterminer la présence d'un élément, et ajouter, modifier, supprimer un élément en temps (amorti) constant.

Exemples de problèmes utilisant un dictionnaire :

- codage d'une partie d'un ensemble très grand, par exemple une partie A de $\{0,1\}^N$, avec N très grand.
On considère alors un dictionnaire dont les clés sont des N-uplets $(x_0, ..., x_{N-1}) \in A$ et dont la valeur est arbitraire, par exemple 1. Une variante est de considérer comme valeur le nombre d'occurrences (lorsque A est une liste d'éléments). La complexité est linéaire.

- test de l'injectivité d'une liste, détermination de l'union et de l'intersection de parties en temps linéaire (amorti).

- Utilisation d'un dictionnaire pour coder un graphe orienté valué $G = (S, A, \omega)$, où S ensemble fini (d'objets non mutables), $A \subset S \times S$ l'ensemble des arêtes et $\omega : A \to \mathbb{R}$ le poids des arêtes.
On représente un graphe par un dictionnaire d dont les clefs sont les sommets $x \in S$ et la valeur de $d[x]$ est la liste des couples $(y, \omega(x, y))$.

3) Tables de hachage

a) Une structure de dictionnaire peut-être réalisée à l'aide d'une **table de hachage**. Le principe des tables de hachages est d'utiliser un tableau H (assez grand) et la position de l'élément (*clé, valeur*) dans ce tableau est fonction de la clé (il s'agit de la fonction de hachage).

Ainsi, connaissant la clé k, on connaît la position $i = h(k)$ de l'élément dans le tableau. L'ajout d'un élément ou la suppression d'un élément ne modifie pas les positions des autres éléments.

La taille du tableau H est appelée largeur de la table de hachage et $h(k)$ est appélé haché de la clé k.

b) Implémentation par listes

Cette table est implémentée dans un tableau de m listes (les listes sont appelées alvéoles et contiennent des couples (*clé, valeur*). Ce tableau est organisé de façon à ce que la liste d'indice i contienne tous les couples (k, e) tels que $h(k) = i$. où $h : K \to [\![m]\!]$ est appelée fonction de hachage.

Ainsi pour rechercher ou supprimer l'élément de clé k, on commence par calculer son haché $h(k)$ qui détermine l'alvéole adéquate et on est alors ramené à une action sur la liste correspondante.
De même pour ajouter un nouvel élément au dictionnaire on l'ajoute à l'alvéole indiquée par le haché de sa clé.

On associe à chaque clé k un élément $h(k)$, où h est choisie de sorte que la fonction $h : K \to [\![m]\!]$ soit bien distribuée (de sorte qu'on puisse espérer que h soit aussi bien distribuée sur l'ensemble des clés.

On construit alors une table (en fait un tableau) indexé par $[\![m]\!]$, de sorte que pour tout $0 \leq i < m$

$$H[i] \text{ est la liste des éléments } x = (k, e) \in A \text{ dont la clé } k \text{ vérifie } h(k) = i$$

Comme h est supposée bien distribuée, on peut espérer que les listes $H[i]$ soient de longueurs raisonnables lorsque la largeur de la table m est au moins du même ordre de grandeur que n le nombre d'éléments de la table (appelé taille).

Les trois opérations (recherche, insertion, suppression) consiste à calculer la clé $i = h(k)$ de l'élément x concerné, puis à parcourir la liste $H[i]$ afin d'effectuer l'opération souhaitée.

Le coût de chacune de ces opérations est donc proportionnel à la longueur de la liste $H[i]$.

Exemple : Pour un ensemble de n étudiants du lycée, on peut choisir pour k le numéro s de sécurité sociale et pour la fonction de hachage h l'application $s \longmapsto s \bmod m$. Pour déterminer si un étudiant est dans la table, on calcule $i = h(s) = s \bmod m$, et on parcourt la liste $H[i]$. En prenant m du même ordre de grandeur que n, on peut espérer que chaque $H[i]$ contient un petit nombre d'étudiants.

Prop : Supposons que la fonction de hachage soit bien répartie, c'est-à-dire que la probabilité qu'une clé soit insérée en position i soit $\frac{1}{m}$, où m est la largeur de la table. Alors la longueur moyenne d'une liste de n éléments rangés aléatoirement dans la table vaut $\boxed{\tau = \frac{n}{m}}$.

Preuve ; Posons $p = \frac{1}{m}$. Le nombre de termes suit une loi binomiale de paramètre $B(n, p)$.

Donc la longueur moyenne d'une liste après n ajouts vaut $np = \frac{1}{m}$, espérance de loi binomiale.

Conséquence : **Insertion, test d'appartenance et suppression ont un coût amorti en $O(1 + \tau)$ lorsque la fonction de hachage est bien répartie** (ce qui en pratique est difficilement garanti).

Terminologie : L'analyse *amortie* d'une structure étudie la complexité (temporelle) d'une suite d'opérations effectuées sur une même structure de données. Elle répartit le surcoût de certaines opérations dispendieuses sur toutes les opérations en prenant en compte le fait que la plupart des opérations sont économes. Elle attribue à chaque opération d'une séquence un coût amorti qui est la moyenne arithmétique du coût total sur l'ensemble de ces opérations.

c) Implémentation par adressage ouvert

Il s'agit d'une alternative à l'implémentation par listes.

Principe : On considère ici H comme un tableau (de longueur $m \geq n$).
Le principe est de placer lors d'une insertion l'élément x dans la première position libre de la forme $h(x) + k \bmod m$, où k prend (par exemple) successivement les valeurs $0\ , 1\ , ...,\ m - 1$.

La suppression consiste à chercher x puis à le supprimer à partir de la position $h(x)$.

Ce type d'implémentation est mal adapté aux recherches d'éléments n'appartenant pas à la liste.

4) Implémentation en Python d'une table de hachage

On considère un ensemble A de points du plan coloriés en blanc ou en noir.
Chaque point est codé par un couple $a = (x, y)$, avec $(x, y) \in [\![0, N]\!]^2$.
L'intérêt des tables de hachage (ou dictionnaires) apparaît lorsque $n = \operatorname{card} A << N^2$.
On va créer une table dont les couples (*clé,valeur*) sont les (a, e), où $a \in A$ et $e \in \{0, 1\}$ la couleur

a) On commence par considérer des fonctions de hachage $h_m : A \to [\![m]\!]$.

Pour toute clé $a = (x, y)$ et pour tout entier $m \in \mathbb{N}^*$, on pose : $h_m(a) = (x + N * y) \bmod m$.

(1) Écrire une fonction `hache(m,N,a)` qui renvoie $h_m(a)$.

Une table de hachage est définie par le triplet `(hachage,m,H)`, où :

- `hachage` est la fonction $h_m : A \to [\![m]\!]$

- `m` est l'entier donnant la largeur de la table de hachage

- `H` est une liste de listes indexées par $[\![m]\!]$, de sorte que pour $0 \leq i < m$, $H[i]$ est la liste des éléments (a, e) de la table dont le haché $h_m(a)$ de la clé a vaut i.

(2) Écrire une fonction `creer_table_vide(m,N)` renvoie une nouvelle table de hachage vide de largeur m munie de la fonction de hachage h_m.

(3) Écrire une fonction booléenne `recherche(table,a)` renvoyant `True` si `table` contient un élément de clé a.

(4) Écrire une fonction `valeur(table,a)` renvoyant l'élément e associé à la clé a dans la table si cette clé est bien présente dans la table, et renvoie `None` sinon.

(5) Écrire une procédure `ajoute(table,x)` qui ajoute l'élément $x = (a, e)$ à la table de hachage. On n'effectuera aucun changement si la clé est déjà présente.

En supposant que les clés hachées soient bien réparties, on obtient une complexité moyenne en $O(1+\tau)$, où $\tau = n/m$. Bien souvent, on ne sait pas à l'avance quel sera le nombre n de clés à stocker dans la table, et on préfère ne pas surestimer ce nombre pour garder un espace mémoire linéaire en le nombre de clés stockées. Ainsi, il est utile de faire varier la largeur m de la table de hachage : si le facteur de remplissage devient trop important, on réarrange la table sur une largeur plus grande (de même, onpeut réduire la largeur de la table lorsque le facteur de remplissage devient petit). On parle alors de tables de **hachage dynamique** pour ces tables à largeur variable.

On va maintenant développer une stratégie pour maintenir à tout moment un facteur de remplissage borné.

Une table dynamique est désormais une liste de 4 éléments `[hachage,m,n,H]`, où m est la largeur de la table et n le nombre d'éléments stockés dans la table.
L'objectifs des tables dynamiques est d'avoir à chaque étape n et m du même ordre de grandeur.

(6) Écrire une fonction `rearrange(table,m2)` prenant en entrée une table de hachage dynamique et une nouvelle largeur `m2` et renvoie une nouvelle table contenant les mêmes éléments.
En supposant que le calcul des valeurs de hachage se fasse en temps constant et que les largeurs soient au du même ordre que n, la complexité doit être en $O(n + m + m_2)$ où n est le nombre de clés présentes dans la table (sa taille), m est l'ancienne largeur de la table, et m_2 la nouvelle largeur.

Une stratégie heuristique simple pour garantir que le facteur de remplissage reste borné est d'utiliser les puissances de 2 comme largeurs de hachage. Après ajout d'un élément à la table, si celle-ci est de taille n strictement supérieure à sa largeur m, on la réarrange sur une largeur $m_2 = 2m$.

(7) Écrire une procédure `ajout(table,n,x)` ajoutant le couple $x = (a, e)$ à la table de hachage (si la clé k n'est pas présente), et en réarrangeant la table dès que $n > m$.

(8) Montrer que le coût amorti d'un ajout de 2^p éléments à une table initialement vide et de largeur 1 est en $O(1)$.

Remarque : Noter qu'on a en permanence $n \leq m < 2n$ (pour n non nul).

4) Exemple d'implémentation en Python. Corrigé

(1)

```
def hache(m,N,a) :
     x,y = a
     return (x+N*y)%m
```

(2)

```
def creer_table_vide(m,N) :
     H = [ [] for i in range(m) ]
     def hachage(a) :  return hache(m,N,a)
     return (hachage,m,H)
```

(3)

```
def recherche(table,a) :
     (hachage,m,H) = table
     i = hachage(a)
     for x in H[i] :  if x[0] == a :  return True
     return False
```

(4)

```
def valeur(table,a) :
     (hachage,m,H) = table
     i = hachage(a)
     for (b,e) in H[i] :  if b == a :  return e
     return None
```

(5)

```
def ajoute(table,x) :
     (hachage,m,H) = table ; a = x[0]
     if recherche(table,a) :  H[hachage(a)].append(x)
```

(6)

```
def rearrange(table,m2) :
     H2 = [ [] for i in range(m2) ]
     def hachage2(a) :  return hache(m,N,a)
     [hachage,m,n,H] = table
     for i in range(m) :
          for x in H[i] :  H2[hachage2(x[0])].append(x)
     return [hachage2,m2,n,H2]
```

Le nombre d'appels à `ajoute` vaut exactement n.

Le coût des deux boucles `for` imbriquées est donc $O(m+n)$.

La création de `table2` est en $O(m_2)$. Donc au total un coût en $O(n + m + m_2)$.

(7)

```
def ajout(table,n,x) :
     [hachage,m,n,H] = table
     k = x[0] ; i = hachage(k) ; flag = True
     for y in H[i] :  if y[0] == k :  flag = False
     if flag :  H[i].append(x) ; table[3] = n+1
     if n > m :      # réarrangement si la taille n dépasse la largeur m
          new_table = rearrange(table,2*m)
          for i in range(4) :  table[i] = new_table[i]
     # la dernière ligne permet de modifier l'objet table sans modifier l'adresse
```

(8)

Les coûts hors réarrangements sont en $O(n)$.

Il y a p réarrangements nécessaires pour les valeurs $(m, m_2) = (2^{j-1}, 2^j)$, avec $1 \leq j \leq p$.

D'où un coût de réarrangement en $O(\sum_{j=1}^{p}(2^{j-1} + 2^j + 2^j)) = O(2^p)$.

Donc le coût de n ajouts est en $O(n)$, et le coût amorti d'un ajout est donc bien en $O(1)$.