EA4 – Éléments d'algorithmique
TP n° 6 : tables de hachage

Vous téléchargerez sur Moodle les fichiers `tp6_ex*.py` à compléter, ainsi que le fichier `proust.txt`.

Dans ce TP, les tables de hachage sont représentées par des objets de type `TableHachage` possédant les attributs suivants :

- `cles` : tableau de clés,
- `lg` : longueur du tableau `cles`,
- `nbCles` : nombre de clés présentes dans la table,
- `nbMarques` : nombre de cases marquées dans la table,
- `h` : nom de la fonction permettant de calculer le haché des clés (c'est-à-dire assurant la conversion de données de type quelconque en entiers, sans exigence de dispersion),
- `h1` : nom de la fonction de hachage primaire,
- `h2` : nom de la fonction de hachage secondaire,
- `tmin` : taux minimal de remplissage,
- `tmax` : taux maximal de remplissage.

Chaque case du tableau de `cles` de la table peut prendre les valeurs :

- `None` (case vide),
- `MARQUE = (None,None)` (case où une clé a été supprimée),
- `(hcle, cle)` où `cle` est une clé (de type quelconque) et `hcle` son haché (entier).

La classe `TableHachage` définit :

- un constructeur `__init__` que l'on invoque en utilisant le nom de la classe. Par exemple : `maTable = TableHachage(8, hash, hash1, hash2, 0.25, 0.75)`,
- quelques méthodes que l'on peut appliquer aux objets de type `TableHachage` : des accesseurs comme `maTable.getCle(5)`, et un générateur `positionsSuccessives` qui permet d'itérer sur les positions successives de sondage d'une clé donnée.

Exercice 1 :

Vous allez écrire les fonctions usuelles d'accès aux tables de hachage avec adressage ouvert et sondage linéaire ou double hachage. Dans un premier temps, vous ne vous préoccuperez pas de redimensionnement (c'est-à-dire qu'il faudra commencer avec une table de taille suffisamment grande et on choisira de préférence une puissance de deux).

La méthode `positionsSuccessives`, invoquée de la façon suivante :

```
for pos in table.positionsSuccessives(cle):
    ...
```

permet d'itérer sur les positions successives de sondage dans la table `table` pour une clé `cle`.

1. Définir les fonctions `hash1(hcle, lg)` et `hash2(hcle, lg)` qui implémentent un sondage linéaire à partir de la position $k =$ `hcle` (modulo $\ell =$ `lg`), c'est-à-dire avec comme position de $(i+1)^{\text{e}}$ sondage $h(k,i) = k + i \mod \ell$.
(remarque : dans ce cas simple, `hash1` et `hash2` ne dépendent pas de `lg`, toute la dépendance en `lg` est prise en charge par `table.positionsSuccessives` ; voir la question 5 pour des exemples plus complexes)

2. Écrire la fonction `rechercher(table, cle, flag)` qui doit avoir les deux comportements différents suivants selon la valeur de `flag` :
   - quand `flag` vaut `False`, elle renvoie la position de la clé `cle` dans le tableau de clés si elle se trouve dans la `table`, et `None` sinon ;
   - quand `flag` vaut `True`, elle renvoie la position de la clé `cle` dans le tableau de clés si elle se trouve dans la `table`, et celle où elle doit être insérée dans le cas contraire.

   Lors de la comparaison de clés, on commencera par comparer les hachés des clés, puis, seulement si ceux-ci sont identiques, on comparera les clés elles-mêmes : l'efficacité est ainsi améliorée dans le cas d'objets complexes.

3. Écrire la fonction `inserer` qui, étant donné une table de hachage et une clé, insère la clé dans la table (sans créer de doublon).

4. Écrire la fonction `supprimer` qui, étant donné une table de hachage et une clé, supprime la clé de la table. Une case où une clé est supprimée prend la valeur `MARQUE`.

5. Afin de faire des tests avec différentes fonctions de hachage, définir les fonctions `hash1b` et `hash2b` (en plus des fonctions `hash1` et `hash2`) permettant d'implémenter les trois fonctions de hachage suivantes (ℓ représente la longueur de la table) :
   - $h(k,i) = \lfloor \{kA\} \times \ell \rfloor + i \mod \ell$,
   - $h(k,i) = k + i(2k+1) \mod \ell$,
   - $h(k,i) = \lfloor \{kA\} \times \ell \rfloor + i(2k+1) \mod \ell$,

   où $\{r\}$ représente la partie fractionnaire du réel r et A est la constante définie au début du fichier `tp6_ex1_ex2.py`.

6. Dans le `main`, décommenter les deux premiers appels à `courbes` afin de comparer les temps d'insertion et de recherche, ainsi que la taille moyenne du plus grand cluster. Les trois premiers graphiques correspondent au hachage sur des listes qui comportent des valeurs proches sur différents intervalles, les trois suivants correspondent au hachage sur des listes formées de valeurs uniformément distribuées. Que constate-t-on ?

   Jouer avec les paramètres `tmin` et `tmax`, et observer l'évolution de ces courbes pour les deux types de listes, puis écrire les commentaires dans l'emplacement réservé pour cela.

Exercice 2 :

1. Écrire la fonction `redimensionner` qui, étant donné une table de hachage et une longueur `lg`, redimensionne la table à la longueur `lg`.

2. Modifier ensuite les fonctions `inserer` et `supprimer` pour qu'elles redimensionnent la table si nécessire :
   a. la fonction `supprimer` doit diviser la longueur de la table par deux si le taux de remplissage passe sous le taux `tmin` ;
   b. la fonction `inserer` doit (éventuellement) doubler la longueur de la table si le taux de remplissage dépasse `tmax` ; mais ***attention***, pour déclencher le redimensionnement, il faut prendre en compte ***à la fois*** le nombre de clés et le nombre de cases marquées : en effet, si le nombre de cases marquées est trop important, la recherche ne sera plus aussi efficace ; ces cases marquées ne seront en revanche pas recopiées dans la nouvelle table, qui n'a donc peut-être pas besoin d'être plus longue que la précédente : il faut décider, en fonction du nombre de clés, s'il faut un réel redimensionnement (longueur doublée) ou un simple nettoyage (longueur conservée).

3. Observer de nouveau les résultats sur les courbes et jouer de nouveau avec les paramètres `tmin` et `tmax`. Pour cela, il faudra décommenter les deux derniers appels à `courbes`.

   Que constate-t-on ? Écrire les commentaires dans l'emplacement réservé pour cela.

Exercice 3 :

Le but de cet exercice est de créer et manipuler une structure de données pour stocker un ensemble de mots. Il faudra essentiellement utiliser les fonctions écrites précédemment.
Le fichier `proust.txt` contient le roman de Marcel Proust, *À la recherche du temps perdu*.
La fonction `proust()` renvoie un générateur des mots du roman et s'utilise de la façon suivante :

```
for mot in proust():
    ...
```

1. Écrire la fonction `mot_to_int(w)` qui, étant donné un mot $w = w_0 \cdots w_n$, renvoie l'entier $h(w) = \sum_{i=0}^{n} c_i \ 31^{n-i} \mod 2^{32}$, où c_i est égal au code ascii de w_i (qui peut être obtenu à l'aide de la fonction `ord`). `mot_to_int(w)` représente le haché du mot `w`.
2. À l'aide des fonctions écrites précédemment, écrire des fonctions permettant de représenter un ensemble de mots par une table de hachage :
   a. `creer_dico(taille=0)` qui crée un ensemble de mots vide ;
   b. `ajouter_mot(dico, mot)` qui ajoute le mot `mot` à l'ensemble de mots `dico` ;
   c. `retirer_mot(dico, mot)` qui supprime le mot `mot` de l'ensemble de mots `dico` ;
   d. `dans_dico(dico, mot)` qui renvoie vrai si le mot `mot` est dans l'ensemble de mots `dico`.
3. Écrire les tests pour tester ces fonctions sur l'ensemble des mots du roman de Marcel Proust.