Ré-implémentation d'un dictionnaire en Python

Nous avons vu que les dictionnaires étaient particulièrement efficaces pour la recherche d'éléments. L'objectif de cette séance de ré-implémenter en Python un tel dictionnaire, ou plutôt une table de hachage *(hash map)*, à partir d'une simple liste.

On éditera un fichier `map.py`.

1 Tableau associatif

Un tableau associatif (ou *map*) est un tableau qui associe une valeur à une clé. À la base, dans Python, il n'y a pas de tableaux, mais des listes. Et ces listes Python sont implémentées avec des tableaux `C` (cf. TD/TP précédent)...

Les clés vont être calculées à partir des noms des étudiants (voir la partie Fonction de hachage). Nous allons prendre pour clés les nombres entiers sur 16 bits (et donc 65536 valeurs possibles). Et chaque valeur devrait être a priori un entier (une note entre 0 et 20, sans demi-points). Mais s'il y a plus de 65536 étudiants, il y aura nécessairement deux noms avec la même clé. Cela s'appelle une collision. Pour gérer cela (voir la partie Gestion des collisions), chaque élément du tableau (ou plutôt de la liste) sera un (petit) dictionnaire...

Question 1 : (TD et TP) – Écrivez la fonction `map_new()`, qui va insérer $2^{16} = 65536$ dictionnaires vides dans une liste.

2 Fonction de hachage

Python propose une fonction de hachage par défaut, avec `hash()`, qui retourne un entier signé sur 64 bits, ce qui pose problème pour notre séance. On considèrera plutôt la fonction suivante, qui retourne un entier (positif) sur 16 bits :

```
def map_hash(name):
    h = hash(name)
    if h < 0:
        h = 2**64 + h
    return h & 0xffff
```

Question 2 : (TD) – Expliquez la fonction `map_hash()`.

Question 3 : (TP) – Programmez la fonction `map_hash()` en rajoutant les commentaires nécessaires à sa compréhension.

3 Gestion des collisions

Chaque élément du tableau est un dictionnaire, initialement vide. Ce dictionnaire va contenir des cellules, chacune avec un nom et une note, pour tous les étudiants ayant la même clé.

Question 4 : (TD et TP) – Écrivez la fonction `map_insert(map, name, mark)`.

4 Comparaison des types `map` et `dict`

Nous allons manipuler une `map m` et un `dict d`, créés ainsi :

```
m = map_new()
d = dict()
```

Question 5 : (TD et TP) – Écrivez une fonction `insert_both(name, mark)` qui ajoute un étudiant et sa note à la fois dans `m` et `d`.

Question 6 : (TD et TP) – Utilisez cette fonction pour insérer 20000 éléments aléatoires (cf. TD/TP de la semaine précédente).

Question 7 : (TD, à vérifier en TP) – Quelle est la taille de la **liste** `m` ? Même question pour le dictionnaire `d`. Pour le vérifier en TP, utiliser `len()`.

La fonction de recherche dans le dictionnaire est très simple :

```
def search_dict(name):
    return d[name]
```

Question 8 : (TD et TP) – Écrivez la fonction de recherche dans la `map`, `search_map(name)`, qui :

1. calcule la clé de hachage correspondant au nom ;
2. récupère la (ou les) cellule(s) associée(s) à cette clé ;
3. retourne la valeur de la bonne cellule.

L'extrait de programme suivant permet de recherche, pour chaque nom d'étudiant (parmi les clés du dictionnaire), sa note dans notre `map` et le dictionnaire, en s'assurant qu'on a bien la même valeur. Et on fait le profil temporel de tout cela.

```
import time_profiler

time_profiler.start()

for name in d.keys():
    note1 = search_map(name)
    note2 = search_dict(name)
    assert(note1 == note2)

time_profiler.stop()
```

Question 9 : (TD et TP) – Complétez le programme `map.py`.

Question 10 : (TP) – Lancez ce programme et analysez son profil temporel. Vérifiez que notre type `map` se comporte plus comme un dictionnaire (insertion et recherche en temps quasi constant) que comme une liste (cf. TD/TP précédent).