PETIT RETOUR SUR LES DICTIONNAIRES

1. Définition

La structure de données « tableaux associatifs », aussi appelée dictionnaire ou table d'associations, associe à un ensemble de clés un ensemble de valeurs ; chaque clé étant associée à au plus une valeur. On peut donc avoir des dictionnaires où certaines, voire toutes les clés, n'ont pas de valeur associée.

Les opérations habituelles sur les dictionnaires sont :

Opération/Effet	Version Formelle (interface)	Pyhton
Construire d'un dictionnaire vide	`Créer_dict_vide() -> Dictionnaire`	`dico = {}`
Tester si un dictionnaire est vide	`Est_dict_vide(Dictionnaire) -> booléen`	`dico =={}`
Ajoute l'item de clé et valeurs `(clé, valeur)` dans un dictionnaire	`ajouter(Dictionnaire, clé, valeur) -> None`	`dico[clé] = valeur`
Modifie l'item de clé `clé` dans un dictionnaire, en lui associant la nouvelle valeur `valeur`.	`modifier(Dictionnaire, clé, valeur) -> None`	`dico[clé] = valeur`
Supprime l'item de clé `clé` dans un dictionnaire. *On peut proposer une variante de l'opération de suppression où l'on renvoie la valeur correspondante*	`supprimer(Dictionnaire, clé) -> None` Ou `supprimer(Dictionnaire, clé) -> valeur`	`dico.pop(clé)` Renvoie `valeur`
Recherche l'item de clé `clé` dans un dictionnaire, renvoyer la valeur si elle existe.	`rechercher(Dictionnaire, clé) -> Valeur/None`	`dico.get(clé)` Renvoie `valeur` ou `None` On peut aussi utiliser `dico[clé]`, renvoie une erreur si la clé est absente

En Python on dispose également de :

- la fonction longueur(dictionnaire) : `len(dico)` ;
- de l'ensemble des clés / valeurs /items d'un dictionnaire ;
  - `dico.keys()` ou plus pratique `list(dico.keys())`
  - `dico.values()` ou plus pratique `list(dico.values())`
  - `dico.items()` ou plus pratique `list(dico.keys())`
    On accède à la clé et à la valeur par `item[0]` et `item[1]`
- du parcours des clés /valeurs, items d'un dictionnaire :
  - `for cle in dico` ou `for cle in dico.keys()`
  - `for valeur in dico.values()`
  - `for cle, valeur in dico.items()`

2. Implémentations efficaces

Commençons par faire mentir le titre du paragraphe, en mentionnant une implémentation inefficace sous forme de tableau dynamique Python de couples (clés, valeurs). Avec cette implémentation, les opérations de recherche sont en complexité $O(n)$.

Les deux implémentations efficaces dans le cas général sont :

a. Avec une table de hachage.
La table de hachage associe une clé à une valeur à l'aide d'une fonction de hachage, qui se calcule en temps $O(1)$.
Exemples simples :
- Hachage par division : on prend le reste de la clé modulo la taille de la table de hachage.
  Avec une table de taille 5, si la clé est 34 alors le hash est 4 car $34 \equiv 4[5]$ (version mathématique) , ou, en version Python `34 % 5 -> 4`.

- Cryptosystème de Rabin. Ici on est dans un cas particulier de hachage, le hachage cryptographique.
  On choisit deux nombres premiers p et q (de préférence grands !), les multiplier. Le hash c d'une clé (un texte en fait) m est donné par $c \equiv m^2[n]$, soit `m**2 % n -> c` en Python.
  Avec $m = 20$, $n = 7 \times 11 = 77$, le hash vaut `400 % 77 -> 15`. Remarquez comme il n'est pas évident de retrouver le texte originel à partir du hash.
- Voir ici pour des exemples de base : https://iq.opengenus.org/hash-functions-examples/

Le problème des collisions : deux clés différentes peuvent donner le même hash, par exemple `34 % 5 -> 4` et `14 % 5 -> 4`. On dit qu'il y a collusion, il faut donc prévoir un mécanisme pour éviter ce phénomène.
Complexité : en général, la recherche et l'ajour d'un élément se font en $O(1)$. En cas de collision, le temps d'ajout d'un élément devient $O(n)$.

b. Avec un arbre équilibré
Comme on l'a vu dans le chapitre sur les arbres binaires de recherche, un arbre équilibré permet d'avoir un coût d'accès de complexité $O(n \log n)$, ce qui est moins bon que le meilleur des cas d'une table de hachage, mais meilleur que le pire. Par ailleurs l'ordre des clés est préservé, ce qui n'est pas le cas avec une table de hachage

En Python, les dictionnaires sont implémentés par table de hachage. Il existe également d'autres implémentations spécialisées efficaces suivant le type des clés.

EXERCICES

Objectif de base : savoir parcourir un dictionnaire, suivant les clés ou les valeurs.
Vous pouvez reprendre les exercices du chapitre de 1ère :
http://www.maths-info-lycee.fr/pdfs/nsi_2_programmations_construits.pdf
Merci au collègue dont j'ai oublié le nom pour les exercices 4 et 5.

1. On se donne un dictionnaire. Trouver le maximum des clés (respectivement des valeurs), sans passer par l'usage des listes ni de la fonction intégrée `max()`.

2. Un problème de https://adventofcode.com/ (Décembre 2020) : « écrire une fonction qui, étant donnée une liste de nombres, trouve les deux nombres de la liste ayant pour somme 2020 et renvoie leur produit. On suppose que ces deux nombres existent et sont uniques ».
   Exemple : `recherche2020([1721, 979, 366, 299, 675, 1456])` renverra `514 579` car 1 721 + 299 = 2 020 et 1 721 × 299 = 514 579
   a. Solution « lente » :
   ```
   def recherche2020(liste) :
       for n in liste :
           if 2020 - n in liste :
   ```

```
return n*(2020 - n)
```

Donner la complexité de cette fonction.

b. Trouver une solution plus rapide en utilisant un dictionnaire et donner sa complexité.

3. Écrire une fonction qui, à partir d'une chaîne de caractères (qui peut être aussi longue que l'on veut, comme une encyclopédie), renvoie un dictionnaire dont chaque item a pour clé un caractère et comme valeur le nombre d'occurrences de ce caractère. Faire une variante de cette fonction qui compte les occurrences des différents mots dans un texte.

4. Écrire une fonction qui échange clés et valeurs d'un dictionnaire. Que se passe-t-il si plusieurs valeurs sont identiques ?

5. Écrire une fonction fusionnant deux dictionnaires de la manière suivante : si `dic1 = {'Tom' : 2, 'Ben' : 1}` et `dic2 = {'Tom' : 1, 'Ben' : 3, 'Luc' : 2}`, alors `fusion(dic1, dic2)` renverra `{'Tom' : 3, 'Ben' : 4, 'Luc' : 2}`.

6. L'ARN contient le codage des protéines, ou des chaînes d'acides aminés. La séquence AUG, par exemple, correspond à la méthionine, noté M. Le dictionnaire ci-dessous donne les correspondances entre les séquences d'ARN, à prendre par groupes de trois nucléotides, et les acides aminés.

```
dico_gen = {'UUU' : 'F', 'UUC' : 'F', 'UUG' : 'L', 'UUA' : 'L', 'UCU' : 'S', 'UCC' : 'S', 'UCG' : 'S', 'UCA' : 'S', 'UAU' : 'Y', 'UAC' : 'Y', 'UAG' : 'STOP', 'UAA' : 'STOP', 'UGU' : 'C', 'UGC' : 'C', 'UGG' : 'W', 'UGA' : 'STOP', 'CUU' : 'L', 'CUC' : 'L', 'CUG' : 'L', 'CUA' : 'L', 'CCU' : 'P', 'CCC' : 'P', 'CCG' : 'P', 'CCA' : 'P', 'CGU' : 'R', 'CGC' : 'R', 'CGG' : 'R', 'CGA' : 'R', 'CAU' : 'H', 'CAC' : 'H', 'CAG' : 'Q', 'CAA' : 'Q', 'ACU' : 'T', 'ACC' : 'T', 'ACG' : 'T', 'ACA' : 'T', 'AUG' : 'M', 'AUU' : 'I', 'AUC' : 'I', 'AUA' : 'I', 'AAU' : 'N', 'AAC' : 'N', 'AAG' : 'K', 'AAA' : 'K', 'AGU' : 'S', 'AGC' : 'S', 'AGG' : 'R', 'AGA' : 'R', 'GUU' : 'V', 'GUC' : 'V', 'GUG' : 'V', 'GUA' : 'V', 'GCU' : 'A', 'GCC' : 'A', 'GCG' : 'A', 'GCA' : 'A', 'GGU' : 'G', 'GGC' : 'G', 'GGG' : 'G', 'GGA' : 'G', 'GAU' : 'D', 'GAC' : 'D', 'GAG' : 'E', 'GAA' : 'E'}
```

Sources : http://www.i3s.unice.fr/~comet/SUPPORTS/Nice-MasterSVS-Python/TD3.pdf
https://www.khanacademy.org/science/ap-biology/gene-expression-and-regulation/translation/a/translation-overview

Écrire une fonction `traduction(arn)` qui traduit une chaîne d'ARN en protéine. On suppose que la longueur de la chaîne d'ARN est un multiple de trois. On pourra écrire un générateur aléatoire d'ARN pour tester le programme.