NSI - Terminale

Structures de données - dictionnaire

qkzk

2020/10/06

pdf : pour impression

Structure de donnée : Dictionnaire

1. Programme

Contenus :

Dictionnaires, index et clé.

Capacités attendues :

Distinguer des structures par le jeu des méthodes qui les caractérisent.
Choisir une structure de données adaptée à la situation à modéliser.
Distinguer la recherche d'une valeur dans une liste et dans un dictionnaire.

2. Définition

- Un **dictionnaire** est une structure de donnée gardant en mémoire des informations de la forme `(clé, valeur)`.
- Le but d'un dictionnaire est d'être capable d'accéder rapidement à une `valeur` à partir de sa `clé`.

3. Interface

Dans le domaine des strucutres de données, **l'interface** désigne l'ensemble des opérations qui sont possibles avec cette structure de donnée.

Elle se distingue de son **implantation** qui désigne le programme permettant de réaliser cette structure de données.

Ainsi, en travaux dirigés, nous avons vu deux implantations différentes de cette même strucure de donnée : avec des listes et avec une table de hachage.

Une seule (avec table de hachage) étant satisfaisante.

4. Interface d'un dictionnaire

Les avis divergent mais *je* considère qu'on doit pouvoir :

1. Créer un dictionnaire vide.
2. Mesurer un dictionnaire : combien d'entrées comporte-t-il ?
3. Ajouter un couple `(clé, valeur)`.
4. Accéder à une `valeur` depuis sa `clé`.
5. Retirer un couple `(clé, valeur)`.
6. Répondre à la question : ce dictionnaire comporte-t-il une entrée avec cette `clé` ?

On peut ajouter d'autres manipulations, en particulier :

1. Itérer sur les `clés`, sur les `valeurs`, sur les couples `(clés, valeurs)` ;
2. Vider un dictionnaire ;
3. Comparer deux dictionnaires (`d1 == d2`) ;
4. Créer un dictionnaire avec des valeurs existantes `d = {'nom': 'Minvussa', 'prenom': 'Gerard'}` ;
5. Faire une copie du dictionnaire etc.

Notons que Python propose toutes ces opérations avec le type `dict`.

Un exemple en programme les illustrant toutes et indiquant à quelle méthode magique ils correspondent est disponible en fin de document.

5. Implantation

1. Table de hachage

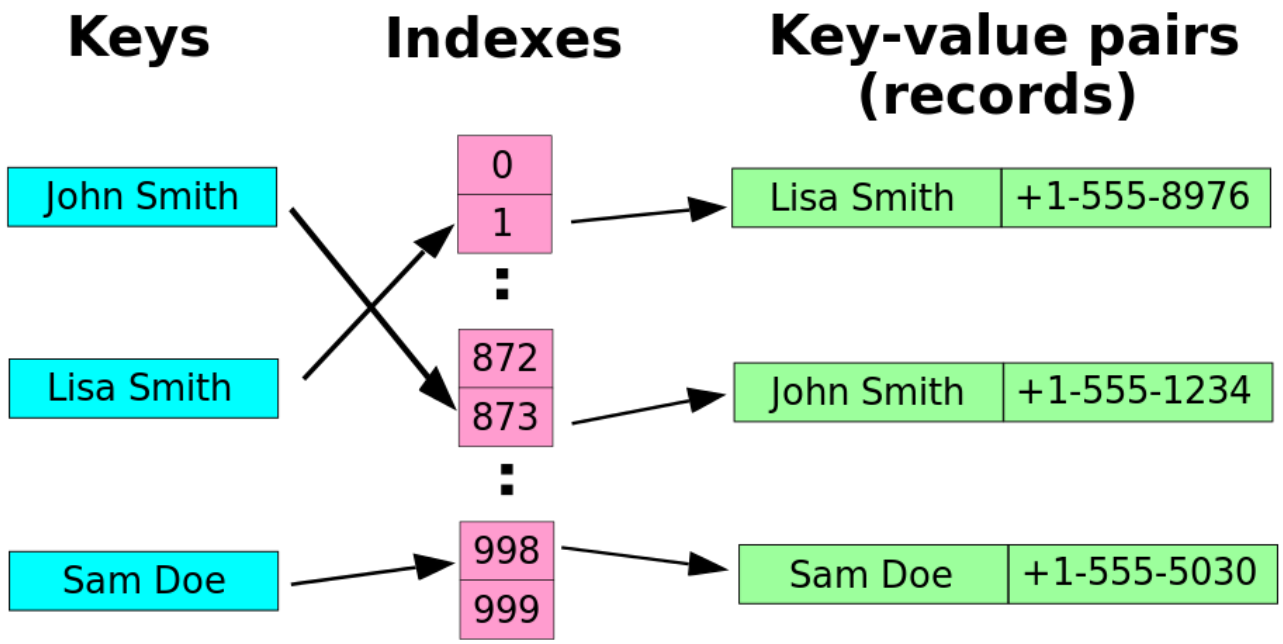

Figure 1: hachage

Le principe est d'utiliser une fonction de hachage (voir plus bas) afin de localiser rapidement les paires `(clé, valeur)`

1. On crée un dictionnaire :

   On initialise un tableau avec un grand nombre d'éléments tous identiques `None` en Python, par exemple.

2. On ajoute le couple `(clé, valeur)` :

   On calcule le hash de la `clé` (en choisissant une fonction de hachage adaptée). Ce nombre devient l'`indice` où sera enregistré la paire.

Exemple

```
>>> contenu = [None] * 1024
>>> hachage('clé')
5
>>> contenu[5] = ('clé', 'valeur')
>>> contenu
[None, None, None, None, None, (clé, valeur), None, None, ...]
>>> contenu[ hachage(('clé')) ][ 1 ]
'valeur'
```

2. Implantation inefficace avec des listes

Le principe est beaucoup plus simple.

Les données sont enregistrées dans une liste :

`contenu = [('a', 1), ('b', 17), ...]`

Pour chaque élément de `contenu`, par exemple `('a', 1)`, le premier élément désigne la clé et le second la valeur. Ainsi `a` est la clé et `1` est la valeur.

Lors de l'ajout, on doit vérifier (en parcourant la liste) qu'elle ne contient pas déjà une `clé` à ce nom.

Cette opération de recherche est longue et nécessite un parcours.

Lors de l'accès à une valeur depuis sa clé, il faut encore parcourir la liste afin de trouver le couple `(clé, valeur)`. Là encore, ce parcours prend beaucoup de temps.

En terme de complexité algorithmique, cette implantation est *linéaire* $O(n)$ où n désigne le nombre d'éléments contenus. Le contrat n'est donc pas respecté !

3. Exemple d'implantation avec une table de hachage

Cet exemple n'est pas une correction du TD précédent, il va plus loin et dépasse largement les attendus du programme.

À la fin on n'obtient *pas* exactement la même interface que celle proposée par les dictionnaires Python. La différence majeure est la taille maximale choisie arbitrairement (1024). Une fois qu'on dépasse cette taille, les collisions sont certaines et les données sont écrasées !

Néanmoins, elle présente une démarche intéressante et je vous invite à l'étudier comme approfondissement.

Les sources

Compléments (hors programme) sur les fonctions de hachage

1. Fonction de hachage

On nomme **fonction de hachage**, de l'anglais *hash function* (*hash* : pagaille, désordre, recouper et mélanger) par analogie avec la cuisine, une fonction particulière qui, à partir d'une donnée fournie en entrée, calcule une empreinte numérique servant à identifier rapidement la donnée initiale, au même titre qu'une signature pour identifier une personne. Les fonctions de hachage sont utilisées en informatique et en cryptographie notamment pour reconnaître rapidement des fichiers ou des mots de passe.

1. Principe général

Une fonction de hachage est typiquement une fonction qui, pour un ensemble de très grande taille (théoriquement infini) et de nature très diversifiée, va renvoyer des résultats aux spécifications précises (en général des chaînes de caractère de taille limitée ou fixe) optimisées pour des applications particulières.

2. Utilisation diverses

On distingue deux types de fonctions de hachage :

1. Celles qui permettent de créer des dictionnaires rapides : la vitesse d'exécution doit alors être *constante* pour chaque donnée d'entrée
2. Les fonctions de hachage cryptographique Le critère fondamental est l'impossibilité à inverser la fonction.

   ```
   >>> hash(???)
   12345987
   ```

   Quelle est la donnée en entrée qui produit ce `hash` ?

Pour créer des dictionnaires, on n'a pas besoin d'obtenir TOUJOURS la même sortie. Ainsi, lors de deux exécutions d'un même programme, la fonction python `hash` ne renvoie pas les mêmes valeurs :

```
$ python -c "print(hash('a'))"
2976264172894533586
$ python -c "print(hash('a'))"
1904027818818373130
```

Par contre, durant une même exécution, la fonction `hash` renvoie toujours la même valeur. Cette propriété permet :

1. de créer des dictionnaires utilisant cette fonction
2. de s'assurer qu'il est difficile de *hacker* la fonction et de provoquer des bugs.

L'utilisation des fonctions de hachage **cryptographique** est multiple, mais grosso modo, elles permettent de s'assurer qu'un contenu n'a pas été modifié.

Par exemple, lors de la publication d'un document, elles peuvent être employées pour le signer.

On publie le document `important.pdf` et sa signature (son hash) `9876786161`

La personne qui télécharge le document calcule de son côté le hash avec la même fonction.

1. Si elle trouve un hash différent, le document OU la signature ont été modifiés
2. Si elle trouve le même hash, le document est beaucoup plus sûr.

L'édition d'un seul bit dans le document provoque un hash totalement différent :

```
$ echo "bonjour" | sha1sum
e7bc546316d2d0ec13a2d3117b13468f5e939f95  -
$ echo "bonjouR" | sha1sum
f6849d79cbdb3be947d586ba195fd4584107fa9c  -
```

- `sha1` est un algorithme de hachage cryptographique qui est maintenant désuet, il est toujours utilisé par Git pour créer un dictionnaire entre les noms de fichiers et leur contenu.
- Le meilleur algorithme de hachage cryptographique à ce jour est `sha256` (ou `sha2`)
- L'enregistrement sécurisé des mots de passe consiste justement… à ne pas enregistrer le mot de passe mais seulement son `hash`.

  Lorsqu'on se connecte, le mot de passe est haché chez le client et seul le hash est transite sur le réseau (de manière sécurisé néanmoins).

  Il est alors comparé avec le hash enregistré dans la BDD.

  Cela rend impossible l'utilisation des données de la BDD pour se connecter si elle venait à être piratée. Il faudrait *remonter* jusqu'au mot de passe et c'est généralement très difficile.

2. La fonction `hash` de Python

Elle prend une unique paramètre (d'un type non mutable) et renvoie un entier. Python l'utilise en interne pour la structure de donnée `dict`

```
>>> {[1], 2} # plante parce que la clé '[1]' est une liste mutable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

En interne elle appelle la méthode `__hash__` du type d'entrée.

Mais comment c'est programmé en interne ?

Comme le reste des fonctions natives, le code est en C.

Cette discussion (en anglais) aborde justement cet aspect. Grosso modo, chaque type d'objet natif dispose de sa propre fonction. Par exemple, celui des tuples qui fait 1000 lignes de C… La liste des contributeurs fait apparaître plusieurs 'célébrités' de Python (Guido Von Rossum, Raymond Hettinger, Tim Peters etc.)

Fonction de hachage cryptographique

Une fonction de hachage cryptographique est une fonction de hachage qui, à une donnée de taille arbitraire, associe une image de taille fixe, et dont une propriété essentielle est qu'elle est pratiquement impossible à inverser, c'est-à-dire que si l'image d'une donnée par la fonction se calcule très efficacement, le calcul inverse d'une donnée d'entrée ayant pour image une certaine valeur se révèle impossible sur le plan pratique. Pour cette raison, on dit d'une telle fonction qu'elle est à sens unique.

Une fonction de hachage cryptographique idéale possède les quatre propriétés suivantes :

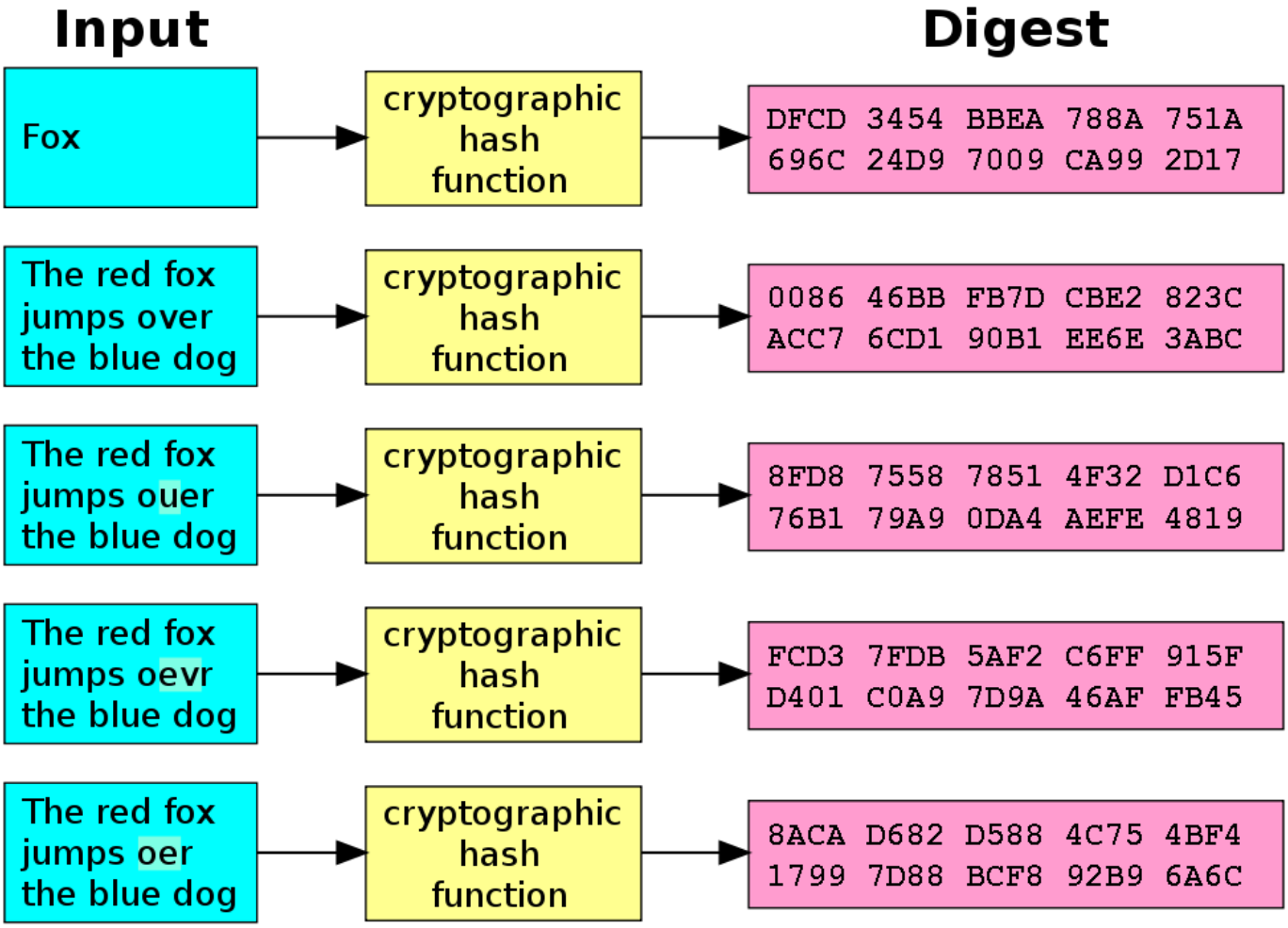

Figure 2: hash crypto

la valeur de hachage d'un message se calcule « très rapidement » ;

- il est, par définition, impossible, pour une valeur de hachage donnée, de construire un message ayant cette valeur de hachage ;
- il est, par définition, impossible de modifier un message sans changer sa valeur de hachage ;
- il est, par définition, impossible de trouver deux messages différents ayant la même valeur de hachage.

Exemples de fonctions de hachage cryptographique

1. MD5 est souvent utilisé pour vérifier qu'un fichier n'a pas été altéré.

   Cette fonction n'est plus sûre depuis des décénies

2. SHA dispose de plusieurs variantes.

   - SHA0 et SHA1 ne sont plus sures mais ont toujours des usages (GIT utilise SHA1 pour comparer les fichiers)
   - SHA2 est celle utilisée partout en pour enregister les mots de passe etc.
   - SHA3 repose sur un autre algorithme (fonction éponge) et n'est pas encore utilisée partout. Cela sera peut-être le cas d'ici une dizaine d'années.

Description rapide de SHA2

Les entrées de la fonction de compression sont découpées

- en mots de 32 bits pour SHA-256 et SHA-224,
- en mots de 64 bits pour SHA-512 et SHA-384.
- La fonction de compression répète les mêmes opérations un nombre de fois déterminé, on parle de tour ou de ronde, 64 tours pour SHA-256, 80 tours pour SHA-512. Chaque tour fait intervenir comme primitives l'addition entière pour des entiers de taille fixe, soit une addition modulo 232 ou modulo 264, des opérations bit à bit : opérations logiques, décalages avec perte d'une partie des bits et décalages circulaires, et des constantes prédéfinies, utilisées également pour l'initialisation.

Avant traitement, le message est complété par bourrage de façon que sa longueur soit un multiple de la taille du bloc traité par la fonction de compression. Le bourrage incorpore la longueur (en binaire) du mot à traiter : c'est le renforcement de Merkle-Damgård ((en)Merkle-Damgård strengthening), ce qui permet de réduire la résistance aux collisions de la fonction de hachage à celle de la fonction de compression. Cette longueur est stockée en fin de bourrage sur 64 bits dans le cas de SHA-256 (comme pour SHA-1), sur 128 bits dans le cas de SHA-512, ce qui « limite » la taille des messages à traiter à 264 bits pour SHA-256 (et SHA-224) et à 2128 bits pour SHA-512 (et SHA-384).