TD3 : tables de hachage

Exercice 1 : Adressage ouvert

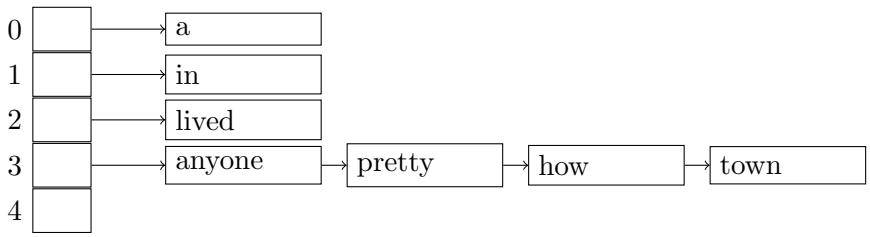
On veut ranger dans une table de hachage des couples <clé,valeur> dont les clés sont les suivantes : 231, 45, 1529, 9, 67, 333, 51. La table est de taille $M = 10$ et on utilisera la somme des chiffres modulo la taille de la table pour fonction de hachage primaire.

Q 1.1 Calculer les adresses associées aux clés que l'on souhaite insérer.

Q 1.2 Lister le parcours des alvéoles et compter le nombre de comparaisons faites pour l'insertion des éléments dans les cas où on utilise un hachage linéaire ($a = 2$).

Exercice 2 : Résolution par chaînage

On suppose disposer d'une table de hachage de taille $M = 5$ où ne sont stockées que des clés dont le contenu est décrit sur le schéma ci-dessous.



Les clés sont des chaînes de caractères et on supposera qu'elles ont une longueur maximale de 10. Les clés sont complétées à droite par des espaces si elles ne font pas la longueur maximale. La fonction transformant la clé en un entier est obtenue en réalisant la somme des rangs de chaque caractère de la clé dans la table ASCII [1].

Q 2.1 Compléter la table ci-dessus avec les mots : with, up, so, floating, many, bells, down.

Q 2.2 Discuter comment vont se répartir les clés lors de l'insertion dans la table lorsque :
- $M = 10$ et $h(k)$ est le reste de la division de la longueur de k par 10,
- $M = 128$ et $h(k)$ est le résultat de la fonction `ord` appliquée au dernier caractère de k,
- $M = 10$ et $h(k)$ est le reste de la division par 10 de la somme des rangs des caractères élevée au carré.

Exercice 3 : Le hachage qui fait coucou

Nous désirons stocker des éléments dans une table de hachage du coucou. Notre table contient dix cellules, nous allons la séparer en deux tables T_1 et T_2, de taille identique.

Nous souhaitons insérer les éléments suivants, dans l'ordre donné, dans notre table de hachage. Les résultats des fonctions de hachage h_1 et h_2 utilisées pour la table de hachage du coucou sont également indiquées.

k	17	59	31	75	66	76
h_1	1	3	1	3	1	3
h_2	4	2	2	1	1	0

Q 3.1 Quel est, après chaque étape, le contenu de T_1 et T_2 ?

1. 'a' est en position 97 et 'z' en position '122', ' ' est en position 32

Exercice 4 : Une utilisation des tables de hachages pour trier

Dans cet exercice, nous allons utiliser des tables de hachage à adressage fermé afin d'implanter un algorithme de tri. Contrairement au cours, on insèrera en queue de liste et non en tête de liste.

Nous cherchons à trier des matricules, des nombres décimaux mais écrits sur un nombre `MAX` **fixe** de chiffres (un type pour représenter un matricule pourrait être un tableau de `MAX` chiffres). On suppose disposer d'une fonction `nieme_chiffre` qui retourne le $n-$ème chiffre (en partant de la droite) d'un matricule. Par exemple, si `MAX` $= 8$, un matricule $m = 41143671$, `nieme_chiffre`$(m, 3) = 6$.

L'algorithme proposé est le suivant :

Entrée : d : un tableau contenant tous les matricules

pour *i allant de 1 à MAX* **faire**
  t : table de hachage de taille 10 dont la fonction de hachage est $h(k) =$ `nieme_chiffre`(k, i);
  pour chaque *matricule m de d* **faire**
    Ajouter m à t;
  fin
  Vider d;
  Ranger tous les matricules de t dans le tableau d en parcourant les alvéoles de 0 à 9 et les listes de chaque alvéole dans l'ordre;
fin
Imprimer le contenu de d;

Q 4.1 Tester l'algorithme sur les données suivantes : 231, 45, 529, 9, 67, 333, 51, 21. Pour chaque étape i, dessiner la table `t` et le tableau `d`.

Q 4.2 Pourquoi est-il important d'ajouter en queue dans les listes ?

Q 4.3 Traduire en Python l'algorithme donné ci-dessus.

Q 4.4 Quelle est le comportement asymptotique de la complexité en espace de cet algorithme ?

Q 4.5 Quelle est le comportement asymptotique de la complexité en temps de cet algorithme ?

Q 4.6 Quels changements faudrait-il opérer pour trier en ordre décroissant ?

Exercice 5 : Occurrences maximales

Etant donné un tableau t de taille n contenant des entiers compris entre 0 et $n-1$, on souhaite calculer la valeur du tableau dont le nombre d'occurrences est maximal.

Q 5.1 Concevoir un algorithme dont la complexité en temps et en espace est en $\Theta(n)$.

Exercice 6 : CycleSort

L'algorithme CycleSort a été proposé en 1990 par Haddon (*The Computer Journal*, vol 33, no 4, p 365–367) pour trier des tableaux d'objets auxquels on pouvait associer à chacun une clé entière dans l'intervalle $[1..n]$. Le problème du tri de tableaux d'objets peut alors se réduire au tri de tableaux d'entiers : en réalisant les même opérations d'échange de valeurs sur le tableau d'objets que sur le tableau d'entiers correspondant, on aura trié les objets.

Si il existe une bijection entre les objets et l'intervalle $[1..n]$, on a alors à trier des tableaux particuliers que sont les permutations. On rappelle qu'une permutation de taille n est la suite des éléments de 1 à n dans n'importe quel ordre.

Ici, et afin de pouvoir manipuler les permutations dans des tableaux, on définira une permutation à n éléments comme la suite des éléments de 0 à $n-1$ dans n'importe quel ordre. Par exemple si $n = 10$: `[2, 4, 7, 6, 3, 9, 8, 0, 5, 1]` est une permutation.

CycleSort est basé sur la remarque que dans une permutation on peut détecter des cycles. Un cycle de taille m est une suite d'éléments $t[i_1], t[i_2], \ldots, t[i_m]$ qui sont tels que $t[i_1] = i_2, t[i_2] = i_3, ..., t[i_m] = i_1$. Par exemple dans la permutation ci-dessus $(2, 7, 0)$ est un cycle puisque $t[0] = 2, t[2] = 7, t[7] = 0$.

Réorganisation d'un cycle Pour un cycle donné, on peut assez simplement remettre ses éléments à la bonne place dans la permutation triée : il suffit de faire « tourner » les éléments du cycle. Sur l'exemple on passera t[0] = 2 dans t[2] (il sera donc bien rangé), t[2] = 7 dans t[7] et t[7] = 0 dans t[0].

Q 6.1 Indiquer tous les cycles de la permutation `[4, 7, 2, 1, 3, 8, 9, 0, 5, 6]`.

Q 6.2 Donner le code en Python d'une procédure `reorganiser` qui étant donné un tableau t et un indice i réorganise le cycle qui passe par la position i. Par exemple :

```
> t = [2, 4, 7, 6, 3, 9, 8, 0, 5, 1]
> reorganiser(t,0)
> t
[0, 4, 2, 6, 3, 9, 8, 7, 5, 1]
```

Q 6.3 Si m est la taille d'un cycle, combien d'itérations seront nécessaires pour réorganiser un cyle ? Justifier.

Q 6.4 Combien y a-t-il de cycles au maximum dans une permutation de taille n, au minimum ?

Processus de tri Une procédure de tri utilisant cette propriété pourrait être :

pour chaque *cycle c* **faire**
  Réorganiser le cycle c;
fin

Q 6.5 Quel est le meilleur des cas pour le nombre d'itérations. Expliquer.

Q 6.6 En identifiant le pire des cas, donner le comportement asymptotique de la procédure de tri.

Extension au tri de tableaux d'éléments quelconques L'introduction de l'article de Haddon débute ainsi :

> The two sorting algorithms to be presented here have applications in rather specialised situations, but situations that are not uncommon in practice. They both operate in linear time, and sort 'in place'.
>
> The first of the algorithms, *general_cycle_sort*, is useful where the following conditions apply:
>
> (1) The keys are integers drawn from a reasonably small set (or can be mapped into such a set):
>
> (2) The number of occurrences of each key is known.
>
> ...
>
> To make this clear, consider a compiler that lists, after other processing, all symbols collected together by type. Each type can be denoted within the compiler by some small ordinal, and the number of occurrences of declarations of symbols of each type can be counted as the source program is compiled. The conditions for *general_cycle_sort* are then met, and the sorting can be accomplished in linear time.

Q 6.7 Proposer les modifications à apporter à l'algorithme pour pouvoir trier un tableau d'objets quelconques.

Q 6.8 Discuter de l'incidence sur la complexité en temps ; et en espace.