Table de hachage (Centrale 2018)

Une structure de dictionnaire est un ensemble de couples (clé, élément), les clés (nécessairement distinctes) appartenant à un même ensemble K, les éléments à un ensemble E.
La structure doit garantir les opérations suivantes :

- recherche d'un élément connaissant sa clé ;
- ajout d'un couple (clé, élément) ;
- suppression d'un couple connaissant sa clé

Une structure de dictionnaire peut être réalisée à l'aide d'une table de hachage. Cette table est implantée dans un tableau de w listes (appelées *alvéoles*) de couples (clé, élément). Ce tableau est organisé de façon à ce que la liste d'indice i contienne tous les couples (k, e) tels que $h(k) = i$ où $h : K \longrightarrow \{0, ..., w-1\}$ s'appelle *fonction de hachage*. On appelle w la *largeur* de la table de hachage et $h(k)$ le *haché* de la clé k. Ainsi pour rechercher ou supprimer l'élément de clé k, on commence par calculer son haché qui détermine l'alvéole adéquate et on est alors ramené à une action sur la liste correspondante. De même pour ajouter un nouvel élément au dictionnaire on l'ajoute à l'alvéole indiquée par le haché de sa clé.
On utilisera le type suivant :

```
type ('a, 'b) table_hachage = {hache: 'a -> int; donnees: ('a * 'b) list array; largeur: int};;
```

I Implantation de la structure de dictionnaire

1. Écrire une fonction `creer` de signature `('a -> int) -> int -> ('a, 'b) table_hachage` telle que `creer h w` renvoie une nouvelle table de hachage vide de largeur `w` munie de la fonction de hachage `h`.

   Solution :

   ```
   let creer h w =
       {hache = h; donnees = Array.make w []; largeur = w}
   ```

2. Écrire une fonction `recherche` de signature `('a, 'b) table_hachage -> 'a -> bool` telle que `recherche t k` indique si la clé `k` est présente dans la table `t`.

   Solution : On peut utiliser `List.exists : ('a -> bool) -> 'a list -> bool` pour savoir si une liste vérifie une propriété.

   ```
   let recherche t k =
       List.exists (fun c -> fst c = k) t.donnees.(t.hache k)
   ```

3. Écrire une fonction `element` de signature `('a, 'b) table_hachage -> 'a -> 'b` telle que `element t k` renvoie l'élément associé à la clé `k` dans la table `t`, si cette clé est bien présente dans la table.

   Solution :

   ```
   let element t k =
       List.filter (fun c -> fst c  = k) t.donnees.(t.hache k)
       |> List.hd
   ```

4. Écrire une fonction `ajout` de signature `('a, 'b) table_hachage -> 'a -> 'b -> unit` telle que `ajout t k e` ajoute l'entrée `(k, e)` à la table de hachage `t`. On n'effectuera aucun changement si la clé est déjà présente.

   Solution :

   ```
   let ajout t k e =
       t.donnees(t.hache k) <- (k, e)::t.donnees(t.hache k)
   ```

5. Écrire enfin une fonction `suppr` de signature `('a, 'b) table_hachage -> 'a -> unit` telle que `suppr t k` supprime l'entrée de la clé `k` dans la table `t`. On n'effectuera aucun changement si la clé n'est pas présente.

   Solution :

```
let suppr t k =
    t.donnees(t.hache k) <- List.filter (fun c -> fst c <> k) t.donnees.(t.hache k)
```

II Étude de la complexité de la recherche d'un élément

Nous étudions ici la complexité de la recherche d'une clé dans une table de hachage. Dans le pire cas, toutes les clés sont hachées vers la même alvéole, ainsi la complexité de la recherche d'une clé dans une table de hachage n'est pas meilleure que la recherche dans une liste. Cependant, si la fonction de hachage h est bien choisie, on peut espérer que les clés vont se répartir de façon apparemment aléatoire dans les alvéoles, ce qui donnera une complexité bien meilleure. Nous faisons donc ici l'hypothèse de hachage uniforme simple : pour une clé donnée, la probabilité d'être hachée dans l'alvéole i est $\frac{1}{w}$, indépendante des autres clés. On note n le nombre de clés stockées dans la table et on appelle $\alpha = \frac{n}{w}$ le facteur de remplissage de la table. On suppose de plus que le calcul du haché d'une clé se fait en temps constant.

1. On se donne une clé k non présente dans la table. Quelle est l'espérance de la complexité de la recherche de k dans la table (on donnera la réponse sous la forme O(...))?

   Solution : Il faut parcourir `t.donnees(t.hache k)` en entier. La taille moyenne de `t.donnees(t.hache k)` étant α, ceci demande $O(\alpha)$ opérations.

2. On prend au hasard une clé k présente dans la table ; toutes les clés sont équiprobables. Quelle est la complexité en moyenne (sur toutes les clés présentes) de la recherche de la clé k (on donnera la réponse sous la forme O(...))?

   Solution : Chaque autre clé ayant une probabilité de $1/w$ d'être hachée dans la même alvéole, Donc l'espérance de la taille de `t.donnees(t.hache k)` est $1 + \frac{n-1}{w} \leq 1 + \alpha$. La complexité de la recherche de la clé k est donc $O(1 + \alpha)$.

III Tables de hachage dynamique

Bien souvent, on ne sait pas à l'avance quel sera le nombre de clés à stocker dans la table, et on préfère ne pas surestimer ce nombre pour garder un espace mémoire linéaire en le nombre de clés stockées. Ainsi, il est utile de faire varier la largeur w de la table de hachage : si le facteur de remplissage devient trop important, on réarrange la table sur une largeur plus grande (de même, on peut réduire la largeur de la table lorsque le facteur de remplissage devient petit). On parle alors de tables de hachage dynamiques pour ces tables à largeur variable.
On définit alors le type suivant :
`type ('a,'b) table_dyn = {hache: int -> 'a -> int; mutable taille: int;}\\ \ocaml { mutable donnees: ('a >`
On notera trois différences par rapport au type précédent :

- la fonction `hache` possède un paramètre supplémentaire qui est la largeur de hachage
- on a rendu les champs `donnees` et `largeur` modifiables ;
- un champ `taille` (modifiable) est rajouté, il doit à tout moment contenir le nombre de clés présentes dans la table.

1. Écrire une fonction `creer_dyn h` permettant de créer une table de hachage dynamique initialement vide, avec la fonction de hachage `h` et la largeur initiale 1.

   Solution :

```
let creer_dyn h =
    {hache = h; taille = 0; donnees = [| [] |]; largeur = 1}
```

2. Écrire une fonction `rearrange t w2` prenant en entrée une table de hachage dynamique `t` et une nouvelle largeur de hachage `w2`, qui réarrange la table sur une largeur `w2`. En supposant que le calcul des valeurs de hachage se fasse en temps constant, la complexité doit être en $O(n + w + w_2)$ où n est le nombre de clés présentes dans la table (sa taille), w est l'ancienne largeur de la table, w_2 la nouvelle.

   Solution :

```
let rearrange_dyn t w2 =
    let d = Array.make w2 [] in
    let rec ajout q = match q with
        | [] -> ()
        | (a, b) :: tl -> d.(t.hache w2 a) <- (a, b) :: d.(t.hache w2 a);
                          ajout tl in
    for i = 0 to t.largeur - 1 do
        ajout t.donnees.(i)
    done;
    t.donnees <- d;
    t.largeur <- w2;;
```

Une stratégie heuristique simple pour garantir que le facteur de remplissage reste borné, tout en garantissant une bonne répartition des clés, est d'utiliser les puissances de 3 comme largeurs de hachage. Après ajout d'un élément à la table, si celle-ci est de taille strictement supérieure à trois fois sa largeur w, on la réarrange sur une largeur $w = 3w$.

3. Écrire une fonction `ajout_dyn t k e` ajoutant le couple (`k`, `e`) à la table de hachage `t`, en réarrangeant si nécessaire la table, en suivant le principe ci-dessus.

   Solution :

```
let ajout_dyn t k e =
    if not (recherche_dyn t k) then begin
        let i = t.hache t.largeur k in
        t.donnees.(i) <- (k, e) :: t.donnees.(i);
        t.taille <- t.taille + 1
        end;
    if t.taille > 3 * t.largeur then
        rearrange_dyn t (3 * t.largeur);;
```