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 $h_w(k) = i$ où $h_w : K \mapsto \{0, \dots, w - 1\}$ s'appelle **fonction de hachage**.

On appelle w la **largeur de la table** de hachage et $h_w(k)$ 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.

I Opérations

I.1 Quelques fonctions sur les listes

Exercice 1

Écrire une fonction `mem k q` de signature `'a -> ('a * 'b)list -> bool` testant l'appartenance d'un couple dont le premier élément est `k` dans la liste `q`.

Exercice 2

Écrire une fonction `assoc k q` de signature `'a -> ('a * 'b)list -> 'b` renvoyant, s'il existe, l'élément `x` du premier couple (k, x) appartenant à la liste `q`.

Exercice 3

Écrire une fonction `plus k e q` de signature `'a -> 'b -> ('a, 'b)list -> ('a, 'b)list` ajoutant l'entrée (k, x) à la liste `q`. On n'effectuera aucun changement s'il existe déjà un couple de première valeur `k`.

Exercice 4

Écrire une fonction `moins k q` de signature `'a -> ('a, 'b)list -> ('a, 'b)list` retirant le couple de première composante `k` de la liste `q`. On n'effectuera aucun changement si la clé n'est pas présente.

I.2 Une famille de fonctions h_w

Nous commençons par nous doter d'une famille de fonctions h_w , pour les listes de couples de $\{0, 1, \dots, N-1\}^2$. Un hachage naturel d'une liste comportant les couples $(a_i, b_i)_{0 \leq i < p}$ avec $0 \leq a_i, b_i < N-1$ est donnée par :

$$P_w(N) = \left(\sum_{i=0}^{p-1} (a_i + b_i N) N^{2i} \right) \text{ modulo } w$$

Autrement dit, on évalue le polynôme dont les coefficients sont donnés par les a_i et b_i en N , et on ne considère que le reste dans la division euclidienne par w .

On considérera que N est une variable globale `let n = xxx;;`.

Exercice 5

Écrire une fonction récursive `hachage_liste w q` de signature `int -> (int * int)list -> int` calculant la quantité précédente.

II Tables de hachage de largeur fixée

Dans cette section, on fixe une largeur de hachage w . Un bon choix pour w serait par exemple un nombre premier ni trop petit, ni trop grand, comme 997. Pour les listes, on considérerait alors la fonction de hachage h_{997} donnée par `hachage_liste 997`. On définit en toute généralité le type suivant :

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

II.1 Implantation de la structure de dictionnaire

Exercice 6

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

Exercice 7

Écrire une fonction `recherche t k` de signature `('a, 'b)table_hachage -> 'a -> bool` renvoyant un booléen indiquant si la clé k est présente dans la table t .

Exercice 8

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

Exercice 9

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

Exercice 10

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

II.2 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_w 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 $1/w$, indépendante des autres clés. On note n le nombre de clés stockées dans la table et on appelle $\alpha = n/w$ le **facteur de remplissage** de la table.

On suppose de plus, que le calcul du haché d'une clé se fait en temps constant.

Exercice 11

On se donne une clé k non présente dans la table. Montrer que l'espérance de la complexité de la recherche de k dans la table est un $O(1 + \alpha)$.

Exercice 12

On prend au hasard une clé présente dans la table; toutes les clés sont équiprobables. Montrer qu'alors la recherche de la clé se fait en $O(1 + \alpha)$, en moyenne sur toutes les clés présentes.

III Tables de hachage dynamiques

Les deux questions précédentes montrent que l'on peut assurer une complexité moyenne constante pour la recherche dans une table de hachage, sous réserve que le facteur de remplissage α soit borné. Il en va de même des opérations d'insertion et de suppression, pour peu que les clés à ajouter/supprimer vérifient des hypothèses d'indépendance.

Bien souvent, et cela va être le cas dans notre problème, 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.

À une table de hachage dynamique est associée une *famille de fonctions de hachage* (h_w). Par exemple, pour les listes de couples de $\{0, 1, \dots, N-1\}^2$, la fonction `hachage_liste` précédemment écrite fournit une telle famille. On définit en toute généralité le type suivant :

```
type ('a, 'b) table_dyn = {hache: int -> 'a -> int;  
                           mutable taille: int;  
                           mutable donnees: ('a * 'b) list vect;  
                           mutable largeur: int};;
```

On notera trois différences par rapport au type précédent :

- la fonction `hache` possède un paramètre supplémentaire qui est la largeur de hachage, elle correspond maintenant à la famille de fonctions de hachage (h_w) ;
- 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.

Exercice 13

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

On admet avoir écrit deux fonctions `recherche_dyn t k` et `element_dyn t k`, variantes des fonctions `recherche` et `element` précédentes, basées sur le même principe. On va maintenant développer une stratégie pour maintenir à tout moment un facteur de remplissage borné.

Exercice 14

Écrire une fonction `rearrange_dyn t w2` prenant en entrée une table de hachage dynamique 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.

Une stratégie heuristique simple pour garantir que le facteur de remplissage reste borné, tout en garantissant une bonne répartition des clés dans le cas des listes de couples à valeurs dans $\{0, 1, \dots, N - 1\}$ avec $N = 16$, 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 $w' = 3w$.

Exercice 15

Écrire une fonction `ajout_dyn t k e` ajoutant le couple (k, e) à la table de hachage (si la clé k n'est pas présente), en réarrangeant si nécessaire la table, en suivant le principe ci-dessus.

Dans l'hypothèse que chaque ajout se fait en temps $O(1 + \alpha)$, où α est le facteur de remplissage de la table, on peut montrer qu'une série de p ajouts dans une table initialement vide prend un temps $O(p)$.

On pourrait écrire de même une fonction de suppression dynamique, de sorte de maintenir un facteur de remplissage de la table borné, et qu'une série de p opérations licites d'insertion/suppression dans la table prenne un temps $O(p)$.

Solutions

Solution de l'exercice 1

```
let rec mem k liste =
  match liste with
  | [] -> false
  |(a, b)::reste when k = a -> true
  |t::reste-> mem x reste;;
```

Solution de l'exercice 2

```
let assoc k liste =
  match liste with
  | [] -> failwith "Element non present"
  |(a, b)::reste when k = a -> b
  |t::reste-> assoc k reste;;
```

Solution de l'exercice 3

```
let plus k x liste =
  match liste with
  | [] -> [(k, x)]
  |(a, b)::reste when k = a -> liste
  |t::reste-> t :: (plus k x reste);;
```

Solution de l'exercice 4

```
let moins liste =
  match liste with
  | [] -> []
  |(a, b)::reste when k = a -> reste
  |t::reste-> t :: (moins reste);;
```

Solution de l'exercice 5

```
let rec hachage_liste w liste =
  match liste with
  | [] -> 0
  |(a, b)::reste ->
    ((hachage_liste w reste)*n*n + a + b*n) mod w;;
```

Solution de l'exercice 6

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

Solution de l'exercice 7

```
let recherche table k =  
  let ind = table.hache k in  
  let liste = table.donnees.(ind) in  
  mem1 k liste;;
```

Solution de l'exercice 8

```
let element table k =  
  let ind = table.hache k in  
  let liste = table.donnees.(ind) in  
  assoc k liste;;
```

Solution de l'exercice 9

```
let ajout table k e =  
  let ind = table.hache k in  
  table.donnees.(ind) <- add k e table.donnees.(ind);;
```

Solution de l'exercice 10

```
let suppression table k =  
  table.donnees.(ind) <- moins k t.donnees.(ind);;
```

Solution de l'exercice 11

La probabilité que l'on recherche k dans l'alvéole i est, d'après l'hypothèse d'uniformité $\frac{1}{w}$. Lorsque que l'on cherche k sans le trouver dans l'alvéole i on parcourt toute la liste.

Si in note A la complexité du calcul de $h_w(k)$ et B la complexité de comparaison d'un élément dans une liste, la complexité de recherche infructueuse de k dans l'alvéole i est donc $A + l_i \cdot B$ où l_i est le nombre d'éléments de l'alvéole.

Ainsi l'espérance de la complexité de la recherche de k dans la table est

$$\sum_{i=0}^{w-1} \frac{1}{w} (A + l_i B) = \frac{1}{w} \left(A \sum_{i=0}^{w-1} 1 + B \sum_{i=0}^{w-1} l_i \right) = \frac{1}{w} (Aw + Bn) = A + B\alpha$$

Solution de l'exercice 12

La probabilité que 2 clés données, k et k' , soient dans la même alvéole est

$$\sum_{i=0}^{w-1} P(h_w(k) = i; h_w(k') = i) = \sum_{i=0}^{w-1} P(h_w(k) = i) P(h_w(k') = i) = \sum_{i=0}^{w-1} \frac{1}{w} \cdot \frac{1}{w} = \frac{1}{w}$$

On a utilisé l'indépendance du hachage et l'équiprobabilité de sont résultat.

Ainsi, étant donnée une clé k , l'espérance de la variable aléatoire X_i valant 0 ou 1 selon que k_i

appartient à la même alvéole que k est $\frac{1}{w}$ donc l'espérance de $\sum_{i=0}^{n-1} X_i$ est $\frac{n}{w} = \alpha$.

Or X est la variable aléatoire dénombrant le nombre d'éléments dans l'alvéole contenant k . On en déduit que la complexité moyenne de recherche de k est majoré par $A + B\alpha$.

Solution de l'exercice 13

```
let creer_table_dyn h =  
  let tab = Array.make 1 [] in  
  {hache = h; taille = 0; donnees = tab; largeur = 1};;
```

On peut écrire les fonctions admises :

```
let recherche_dyn table k =  
  let w = table.taille in  
  let ind = table.hache w k in  
  let liste = table.donnees.(ind) in  
  mem1 k liste;;
```

```
let element_dyn table k =  
  let w = table.taille in  
  let ind = table.hache w k in  
  let liste = table.donnees.(ind) in  
  assoc k liste;;
```

Solution de l'exercice 14

```
let rearrange_dyn table w2 =  
  let d2 = Array.make w2 [] in  
  let h2 = table.hache w2 in  
  let rec ajout liste =  
    match liste with  
    | [] -> ()  
    |(a, b)::reste -> d2.(h2 a) <- (a, b)::(d2.(h2 a)); ajout  
    reste  
  in for i = 0 to t.largeur - 1 do  
    ajout table.donnees.(i) done;  
  t.donnees <- d2;  
  t.largeur <- w2;;
```

Solution de l'exercice 15

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