▶ **MP & MP* – Option Informatique**
Année 2002–2003, Premier TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Tables de hachage

Soient K et E deux ensembles, les éléments de K étant appelés clefs. *Une* table d'association *(ou dictionnaire) m de K vers E est une partie de $K \times E$ telle que pour toute clef $k \in K$ il existe au plus un $e \in E$ tel que le couple (k, e) soit dans m. Une implémentation simple d'une telle table d'association peut naturellement être effectuée à l'aide d'une liste d'association formée de couples de $K \times E$.*
*Les tables de hachage (*hash tables *en anglais) sont une implémentation souvent plus efficace d'une telle structure de données. L'idée est la suivante : pour chaque clef k, on calcule un* entier de hachage $h_w(k)$ *compris entre 0 et $w-1$ (w est appelé la* largeur *de la table). On utilise ensuite un tableau de w listes pour stocker les enregistrements : la liste numéro i contenant les couples (k, e) de la table d'association tels que $h_w(k) = i$.*

1 Fonctions de hachage

Pour implémenter une table de hachage, il est nécessaire de disposer d'une fonction de hachage h_w de K vers $[\![0, w-1]\!]$. Pour que le hachage soit efficace, il est utile que cette fonction assure une bonne répartition des clefs dans les différentes cases de la table, c'est à dire, de manière informelle, que étant donné un entier i de $[\![0, w-1]\!]$, la probabilité que $h_w(k) = i$ soit voisine de $1/w$. Dans cette section, nous donnons quelques exemples de telles fonctions.

1.1 Entiers naturels

Pour le cas où les clefs sont des entiers, le *hachage par division* de largeur w consiste à hacher la clef entière k en le reste de la division de k par w ($k \bmod w$) qui appartient bien à l'intervalle $[\![0, w-1]\!]$.

▶ **Question 1** *Écrivez une fonction* `hash_int` *telle que* `hash_int w k` *retourne le haché de la clef entière k en utilisant un hachage par division de largeur* `w`.

```
hash_int: int -> int -> int
```

Il arrive que la répartition des clefs ne permette pas au hachage par division de donner de bons résultats. Dans ce cas, on peut utiliser un *hachage par multiplication*. Pour cela, on choisit un entier a fixé et on définit h_w par

$$h_w(k) = \left\lfloor \frac{w(ak \bmod \mathrm{card}(K))}{\mathrm{card}(K)} \right\rfloor$$

1.2 Chaînes de caractères

Dans le cas où les clefs sont des chaînes de caractères, on peut se ramener au cas des entiers en utilisant le code ASCII des caractères (le code ASCII d'un caractère est un entier compris entre 0 et 255 identifiant le caractère de manière unique). Une chaîne $s = s_0 \dots s_{n-1}$ peut alors être vue comme la représentation en base 256 de l'entier

$$\sum_{k=0}^{n-1} \mathrm{code}(s_k) \times 256^k$$

Le code d'un caractère est retourné en Caml par la fonction `int_of_char` (de type `char -> int`).

▶ **Question 2** *Écrivez une fonction* `convert_string` *qui retourne l'entier représenté par une chaîne de caractères comme expliqué ci-dessus.*

```
convert_string: string -> int
```

▶ **Question 3** *Déduisez-en une fonction de hachage (par division) des chaînes de caractères.*

```
hash_string: int -> string -> int
```

2 Tables de hachage de taille fixe

Nous représentons en Caml une table de hachage de clefs de type 'a vers des valeurs de type 'b par un enregistrement du type suivant :

```
type ('a, 'b) hashtbl =
  { hash: 'a -> int;
    data: ('a * 'b) list vect
  }
;;
```

Soit m une table de hachage de largeur w. m.hash est la fonction de hachage utilisée pour hacher les clefs stockées dans la table (notée dans l'énoncée h_w) et m.data est un tableau de longueur w. La case numéro i de ce tableau contient la liste des entrées (k, e) de la table tels que $h_w(k) = i$.

▶ **Question 4** *Écrivez une fonction* create *telle que* create h w *retourne une nouvelle table de hachage vide de largeur* w *utilisant la fonction de hachage* h.

```
create: ('a -> int) -> int -> ('a, 'b) hashtbl
```

Nous allons maintenant écrire des fonctions permettant d'effectuer les opérations de base sur ces tables : recherche, ajout et suppression.

▶ **Question 5** *Écrivez une fonction* mem *telle que* mem m k *rende un booléen indiquant si la clef* k *est présente dans la table* m.

```
mem: ('a, 'b) hashtbl -> 'a -> bool
```

▶ **Question 6** *Écrivez une fonction* find *telle que* find m k *retourne la valeur* e *associé à la clef* k *dans la table* m. *Si la clef* k *n'est pas présente dans la table* m, *votre fonction lèvera l'exception* Not_found.

```
find: ('a, 'b) hashtbl -> 'a -> 'b
```

▶ **Question 7** *Écrivez une fonction* add *telle que* add m k e *ajoute l'entrée* (k, e) *à la table de hachage* m. *(Vous préciserez le comportement de votre fonction si la clef* k *est déjà présente dans la table.)*

```
add: ('a, 'b) hashtbl -> 'a -> 'b -> unit
```

▶ **Question 8** *Écrivez enfin une fonction* remove *telle que* remove m k *supprime l'entrée de la clef* k *dans la table* m. *(Vous préciserez le comportement de votre fonction si la clef* k *n'est pas présente dans la table.)*

```
remove: ('a, 'b) hashtbl -> 'a -> unit
```

3 Tables de hachage de taille dynamique

On constate en général que la partie coûteuse de la recherche d'une entrée dans la table est le parcours de la liste des enregistrements correspondant à la valeur de hachage de la clef considérée. Dans les tables que nous avons considéré jusqu'à présent, la largeur est fixée une fois pour toutes au moment de la création de la table. Ainsi, au fur et à mesure que l'on ajoute des entrées, la longueur des listes et susceptibles d'augmenter et, par conséquent, le coût des recherches.
Dans cette section, on se propose d'améliorer ce point en utilisant des tables de hachage de taille dynamique : l'idée est d'augmenter la largeur de la table dès lors qu'il y a trop d'éléments. Ainsi, au fur et à mesure de l'ajout d'entrées, on garde (en moyenne) des listes courtes dans lesquelles la recherche est rapide.
Pour cela, on définit une nouvelle représentation des tables de hachage :

```
type ('a, 'b) dyn_hashtbl =
  { hash: int -> 'a -> int;
    mutable size: int;
    mutable data: ('a * 'b) list vect
  }
;;
```

Dans cette nouvelle représentation, on dispose d'un champ supplémentaire, size, qui permet de stocker le nombre d'entrées de la table. Ce champ est déclaré *mutable* de manière à pouvoir être mis à jour à chaque ajout ou suppression. Vous noterez également que la fonction de hachage d'une table de hachage dynamique prend un argument supplémentaire : la largeur de hachage.

▶ **Question 9** *Écrivez une fonction* dyn_create *permettant de créer une table de hachage dynamique. Les arguments de cette fonction seront la fonction de hachage et la largeur initiale. Écrivez ensuite deux fonctions* dyn_mem *et* dyn_find.

```
dyn_create: (int -> 'a -> int) -> int -> ('a, 'b) dyn_hashtbl
dyn_mem: ('a, 'b) dyn_hashtbl -> 'a -> bool
dyn_find: ('a, 'b) dyn_hashtbl -> 'a -> 'b
```

Nous utiliserons le principe de redimensionnement suivant : lors de l'ajout d'une entrée, si la taille de la table (i.e. le nombre d'entrées) dépasse le double de la largeur courant w, alors on réarrange la table sur une largeur $2w$ avant de procéder à l'ajout.

▶ **Question 10** *Écrivez une fonction* dyn_rearrange *qui réarrange une table en doublant sa largeur.*

```
dyn_rearrange: ('a, 'b) dyn_hashtbl -> unit
```

▶ **Question 11** *Déduisez-en une fonction* add_dyn *telle que* add_dyn m k e *ajoute l'entrée* (k, e) *à la table* m *en effectuant un réarrangement si nécessaire.*

```
dyn_add: ('a, 'b) dyn_hashtbl -> 'a -> 'b -> unit
```

On s'intéresse maintenant à la suppression d'une entrée.

▶ **Question 12** *Que pensez-vous de la stratégie consistant à réarranger une table lors d'une suppression si sa taille devient inférieure à la moitié de sa largeur. Quelle autre solution proposez-vous ? Implémentez ainsi une fonction* dyn_remove *supprimant une entrée d'une table dynamique.*

```
dyn_remove: ('a, 'b) dyn_hashtbl -> 'a -> unit
```

▶ **MP & MP* – Option Informatique**
Année 2002–2003, Premier TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Tables de hachage

Un corrigé

▶ Question 1

```
let hash_int w k =
  k mod w
;;
```

▶ Question 2

```
let convert_string s =
  let i = ref 0 in
  let n = string_length s in
  for k = n-1 downto 0 do
    i := int_of_char s.[k] + 256 * !i
  done;
  !i
;;
```

▶ Question 3

```
let hash_string w s =
  hash_int w (convert_string s)
;;
```

▶ Question 4

```
let create h n =
  { hash = h;
    data = make_vect n []
  }
;;
```

▶ Question 5

```
let rec list_mem k = function
    [] -> false
  | (k', _) :: tail ->
      (k = k') || (list_mem k tail)
;;

let mem m k =
  let i = m.hash k in
  list_mem k m.data.(i)
;;
```

▶ Question 6

```
let rec list_find k = function
    [] -> raise Not_found
  | (k', e) :: _ when k = k' -> e
  | _ :: tail -> list_find k tail
;;

let find m k =
  let i = m.hash k in
  list_find k m.data.(i)
;;
```

▶ Question 7

```
let add m k e =
  let i = m.hash k in
  m.data.(i) <- (k, e) :: m.data.(i)
;;
```

▶ Question 8

```
let rec list_remove k = function
    [] -> []
  | (k', _) :: tail when k = k' -> tail
  | c :: tail ->
      c :: (list_remove k tail)
;;

let remove m k =
  let i = m.hash k in
  m.data.(i) <- list_remove k m.data.(i)
;;
```

▶ Question 9

```
let dyn_create h n =
  { hash = h;
    size = 0;
    data = make_vect n []
  }
;;

let dyn_mem m k =
  let w = vect_length m.data in
  let i = m.hash w k in
  list_mem k m.data.(i)
;;

let dyn_find m k =
  let w = vect_length m.data in
  let i = m.hash w k in
  list_find k m.data.(i)
;;
```

▶ Question 10

```
let dyn_rearrange m =
  let old_table = m.data in
  let w = vect_length old_table in
  let new_table = make_vect (2 * w) [] in
  m.data <- new_table;

  let rec list_add = function
      [] -> ()
    | (k, e) :: tail ->
        list_add tail;
        let i = m.hash (w * 2) k in
        new_table.(i) <- (k,e) :: new_table.(i)
  in
  for i = 0 to w - 1 do
    list_add old_table.(i)
  done
;;
```

▶ Question 11

```
let dyn_add m k e =
  if m.size >= 2 * (vect_length m.data)
  then dyn_rearrange m;

  let w = vect_length m.data in
  let i = m.hash w k in
  m.data.(i) <- (k, e) :: m.data.(i);
  m.size <- m.size + 1
;;
```

▶ Question 12

```
let rec list_force_remove k = function
    [] -> raise Not_found
  | (k', _) :: tail when k = k' -> tail
  | c :: tail -> c :: list_force_remove k tail
;;

let dyn_remove m k =
  let w = vect_length m.data in
  let i = m.hash w k in
  try
    m.data.(i) <- list_force_remove k m.data.(i);
    m.size <- m.size - 1
  with
    Not_found -> ()
;;
```