Chapitre 10

Correction des exercices

Exercice 1 La probabilité qu'il n'y ait pas de collision est égale à :

$$p = \frac{m\times(m-1)\times\cdots\times(m-k+1)}{m^k} = \prod_{i=1}^{k-1}\Bigl(1-\frac{i}{m}\Bigr).$$

On utilise l'inégalité de convexité : $\mathrm{e}^{-x} \geqslant 1-x$ pour majorer : $p \leqslant \prod_{i=1}^{k-1} \mathrm{e}^{-i/m} = \exp\Bigl(-\frac{k(k-1)}{2m}\Bigr)$.

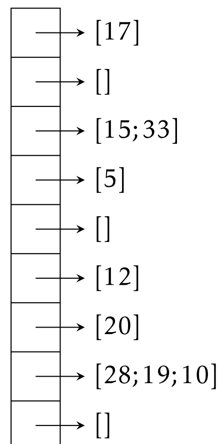
Ainsi, on a $p \leqslant \frac{1}{2}$ dès lors que $\frac{k(k-1)}{2m} \geqslant \ln 2 \iff k^2-k-2m\ \ln 2 \geqslant 0 \iff k \geqslant \frac{1+\sqrt{1+8m\ln 2}}{2}$.

Exercice 2

a) Les valeurs de hachage sont les suivantes :

c	5	28	19	15	20	33	12	17	10
c mod 9	5	1	1	6	2	6	3	8	1

Trois collisions ont lieu ; sachant que l'insertion se fait en queue de liste (en tout cas pour les primitives que nous avons écrites en cours) on obtient la répartition suivante :



b) Si la répartition dans le tableau est uniforme, la valeur moyenne des longueurs des listes de chaque emplacement est égale à $\frac{k}{m}$. La recherche d'une clé non présente dans la table nécessite le parcours de la liste dans son entier, ce qui conduit à une durée moyenne en $\Theta\Bigl(\frac{k}{m}\Bigr)$.

c) La durée moyenne de recherche de c est égale à $\frac{1}{k/m}\times\frac{k/m(k/m+1)}{2} = \frac{k/m+1}{2}$. La recherche d'une clé présente dans la table a donc une durée moyenne elle aussi en $\Theta\Bigl(\frac{k}{m}\Bigr)$.

d) lorsque $k = \mathrm{O}(m)$ la recherche d'une clé dans la table a un coût en moyenne en $\Theta(1)$, c'est à dire un coût constant.

Exercice 3

a) Les valeurs de hachage sont les suivantes :

c	5	28	19	15	20	33	12	17	10
$c \bmod 9$	5	1	1	6	2	6	3	8	1
			2		3	7	4		0

(sous les collisions sont indiquées les résolutions après sondage).
La répartition des clés est donc la suivante :

10	28	19	20	12	5	15	33	17

b) On définit ainsi les fonctions demandées :

```
let find t c =
  let rec sonde i = match t.Tbl.(i) with
    | Nil                   -> raise Not_found
    | Data d when d.Key = c -> d.Value
    | _                     -> sonde (t.H (i+1))
  in sonde (t.H c) ;;

let add t c v =
  let rec sonde i = match t.Tbl.(i) with
    | Nil -> t.Tbl.(i) <- Data {Key = c ; Value = v}
    | _   -> sonde (t.H (i+1))
  in sonde (t.H c) ;;
```

c) Si certaines clés sont supprimées de la table, un sondage risque de se terminer avant qu'une clé soit trouvée, même si celle-ci figure dans la table. Pour remédier à ce problème, une solution possible consiste pour une recherche à distinguer les cases vides (qui occasionnent la fin du sondage) des cases effacées (qui ne mettent pas fin au sondage).
Concrètement, il faut redéfinir le type *('a, 'b) data* de la façon suivante :

```
type ('a, 'b) item = Nil | Erased | Data of ('a, 'b) data ;;
```

La fonction **find** n'a pas besoin d'être redéfinie, et on lui ajoute :

```
let add t c v =
  let rec sonde i = match t.Tbl.(i) with
    | Nil | Erased -> t.Tbl.(i) <- Data {Key = c ; Value = v}
    | _            -> sonde (t.H (i+1))
  in sonde (t.H c) ;;

let remove t c =
  let rec sonde i = match t.Tbl.(i) with
    | Nil | Erased          -> ()
    | Data d when d.Key = c -> t.Tbl.(i) <- Erased
    | _                     -> sonde (i+1)
  in sonde (t.H c) ;;
```

d) Si on suppose le hachage uniforme, la probabilité qu'une case précédée d'une case vide soit désignée pour accueillir une clé donnée est égale à $\frac{1}{m}$, tandis qu'une case précédée de k cases pleines a une probabilité égale à $\frac{k+1}{m}$, puisque la désignation de l'une quelconque des k cases précédentes affecte la clé à cette case.
Ceci a pour conséquence que plus la succession de cases pleines est longue, plus grande est la probabilité d'agrandir encore cette chaîne, ce qui conduit à la formation d'agrégats. Or plus longue est la chaîne, plus long est le sondage ; ce phénomène contribue donc à ralentir la lecture et l'ajout dans une table d'association.

Exercice 4

a) Les deux fonctions demandées sont semblables à celles du cours :

```
let new m = {Size = 0 ; H = (function x -> x mod m) ; Tbl = make_vect m []} ;;

let find t c =
  let rec aux = function
    | []                  -> raise Not_found
    | d::_ when d.Key = c -> d.Value
    | _::q                -> aux q
  in aux t.Tbl.(t.H c) ;;
```

b) On double la taille de la table à l'aide de la fonction suivante :

```
let reorder t =
  let old = t.Tbl in
  let m = vect_length old in
  t.Tbl <- make_vect (2*m) [] ;
  t.H <- (function x -> x mod (2 * m)) ;
  let rec aux = function
    | []   -> ()
    | d::q -> aux q ;
              let i = t.H d.Key in
              t.Tbl.(i) <- d::t.Tbl.(i)
  in for k = 0 to m - 1 do aux old.(k) done ;;
```

La fonction d'ajout prend alors la forme suivante :

```
let add t c v =
  let n = t.Size and m = vect_length t.Tbl in
  if n >= 2 * m then reorder t ;
  let i = t.H c in
  t.Size <- t.Size + 1 ;
  t.Tbl.(i) <- {Key = c ;  Value = v}::t.Tbl.(i) ;;
```

c) Enfin, pour la fonction de suppression il faut tenir compte du fait pour maintenir le champ `Size` à jour que l'élément qu'on cherche à supprimer peut éventuellement ne pas se trouver dans la table. On utilise pour cela le mécanisme de déclenchement d'une exception.

```
let remove t c =
  let i = t.H c in
  let rec aux = function
    | []                  -> raise Not_found
    | d::q when d.Key = c -> q
    | d::q                -> d::(aux q)
  in try
       t.Size <- t.Size - 1 ;
       t.Tbl.(i) <- aux t.Tbl.(i)
    with Not_found -> () ;;
```