TP 4 : Tables de Hachage: corrigé.

1 Fonctions de hachage

Question 1.

```
let hachage_entier w k=k mod w ;;
```

Question 2. Deux versions. Celle de droite (méthode d'Hörner) effectue un peu moins d'opérations.

```
let hachage_chaine w s=
  let x=ref 0 and p=ref 1 in
  for i=0 to String.length s - 1 do
    x:=(!x + (int_of_char s.[i])* !p) mod w ;
    p:= (!p*128) mod w
  done ;
  !x
;;
```

```
let hachage_chaine w s =
  let x=ref 0 in
  for i=String.length s - 1 downto 0 do
    x:= ( !x * 128 + int_of_char s.[i]) mod w
  done ;
  !x
;;
```

2 Tables de hachage de taille fixe

Question 3.

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

Question 4. Si la clé k est présente, elle ne peut se trouver que dans la liste numéro $h_\omega(k)$, avec h_ω la fonction de hachage. On se ramène donc à la recherche dans la liste indexée par $h_\omega(k)$ d'un couple dont la première composante est k.

```
let recherche t k=
  let rec aux l=match l with
    | [] -> false
    | x::q -> fst x=k || aux q
  in aux t.donnees.(t.hache k)
;;
```

On a utilisé `fst` (et on utilise `snd` dans la suite) spécifique aux couples. Il est possible de filtrer aussi une liste de couples avec `(a,b)::q`, ou encore de déconstruire un couple comme en Python avec `let a,b=x in ...`.

Question 5.

```
let element t k=
  let rec aux l=match l with
    | [] -> raise Not_found
    | (a,b)::q when a=k -> b
    | _::q -> aux q
  in aux t.donnees.(t.hache k)
;;
```

Remarque : la fonction auxiliaire est très semblable à la fonction `assoc` de Caml. En effet, `assoc x l` de signature `'a -> ('a * 'b) list -> 'b` retourne le premier y tel que (x, y) est dans la liste ℓ, et lève l'exception `Not_found` s'il n'y en a pas. Rattraper les exceptions se fait avec `try with`, voici une autre implémentation de la fonction `recherche` :

```
let recherche t k=
  try
    let _=assoc k t.donnees.(t.hache k) in true (* let _ =... nécessaire pour le type *)
  with Not_found -> false
;;
```

Question 6.

```
let ajout t k e= if not recherche t k then let hk=t.hache k in t.donnees.(hk) <- (k,e)::t.donnees.(hk) ;;
```

Question 7.

```
let suppression t k=
  let rec aux q =match q with
    | [] -> []
    | x::p when fst x=k -> p (* il y a au plus un couple de clé k: pas besoin d'examiner p *)
    | x::p -> x::(aux p)
  in let hk=t.hache k in t.donnees.(hk) <- aux t.donnees.(hk)
;;
```

3 Une application : trouver les points accessibles depuis un point du plan

Question 8.

```
let creation_hach tab d=
  let p=d/distance_max in
  let hach (x,y)=x/distance_max+p*(y/distance_max) in
  let th=creer_table hach (p*p) in
  for i=0 to Array.length tab-1 do
    ajout th tab.(i) i
  done ;
  th
;;
```

Question 10.

```
let candidats_accessibles d x y th=
  let hach=th.hache in
  let candidats=ref [] and t1=[|-distance_max; 0; distance_max|] in
  for i=0 to 2 do
    for j=0 to 2 do
      let x2,y2=x+t1.(i), y+t1.(j) in
      if 0<=x2 && x2<d && 0<=y2 && y2<d then
        candidats:=th.donnees.(hach (x2,y2)) @ !candidats
    done
  done ;
  !candidats
;;
```

Question 11. On calcule la liste des candidats, et on extrait via une fonction récursive uniquement les accessibles (on ne garde que les indices dans le tableau).

```
let accessibles d x y th tab=
  let ca=candidats_accessibles d x y th in
  let rec aux acc q=match q with
    | [] -> acc
    | ((a,b),i)::p when (x-a)*(x-a)+(y-b)*(y-b) <= distance_max*distance_max -> aux (i::acc) p
    | _::p -> aux acc p
  in aux [] ca
;;
```

4 Tables de hachage dynamiques

Question 12.

```
let creer_table_dyn h w={hache= h ; taille=0 ; donnees=Array.make w []} ;;

let recherche_dyn t k=
  let w=Array.length t.donnees in
  let hk=t.hache w k in
  try
```

```
    let a= (List.assoc k t.donnees.(hk)) in true
  with Not_found -> false
;;

llet element_dyn t k=
  let w=Array.length t.donnees in
  let hk=t.hache w k in
  List.assoc k t.donnees.(hk) ;
;;
```

On a ici utilisé `assoc`, on aurait également pu faire comme précédemment !

Question 13.

```
let rearrange_dyn t=
  let w=Array.length t.donnees in
  let nv_donnees=Array.make (2*w) [] in
  let nv_h=t.hache (2*w) in
  let rec aux l=match l with
    | [] -> ()
    | x::q -> nv_donnees.(nv_h (fst x)) <- x::nv_donnees.(nv_h (fst x)) ; aux q
  in
  for i=0 to w-1 do
    aux t.donnees.(i)
  done ;
  t.donnees <- nv_donnees
;;
```

Question 14.

```
let ajout_dyn t k e=
  let w=vect_length t.donnees in
  if not recherche_dyn t k then  begin
    let hk=t.hache w k in t.donnees.(hk) <- (k,e)::t.donnees.(hk) ;
    t.taille <- t.taille + 1 ;
    if t.taille > 2*w then rearrange_dyn t
  end
;;
```

Dans la fonction ci-dessus, on réarrange la table après ajout, ce qui est légèrement plus simple.

Question 15. Le problème de réarranger une table (en diminuant sa largeur de moitié) lorsque sa taille devient inférieure à sa largeur est que certains enchaînements d'ajouts et de suppressions entrainent un réarrangement systématique de la table, ce qui est très coûteux. Réduire la largeur de hachage d'un facteur 2 lorsque la taille de la table est inférieure à la moitié de sa largeur est bien meilleur (voir en fin de corrigé).

```
let rearrange_bis_dyn t=
  let w=Array.length t.donnees in
  let nv_donnees=Array.make (w/2) [] in
  let nv_h=t.hache (w/2) in
  let rec aux l=match l with
    | [] -> ()
    | x::q -> nv_donnees.(nv_h (fst x)) <- x::nv_donnees.(nv_h (fst x)) ; aux q
  in
  for i=0 to w-1 do
    aux t.donnees.(i)
  done ;
  t.donnees <- nv_donnees
;;
```

On a écrit ici une fonction de réarrangement de la table, qui diminue de moitié la taille de la table. La fonction de suppression qui suit utilise une fonction auxiliaire qui traite le cas où la clé à supprimer n'est pas présente avec l'exception `Not_found`. On pourrait aussi adapter la fonction `aux` de la fonction de suppression précédente pour renvoyer également un booléen (suivant si la valeur était présente ou non) ou faire comme pour l'ajout et commencer par une recherche.

```
let suppression_dyn t k=
  let w=vect_length t.donnees in
  let hk=t.hache w k in
  let rec aux q = match q with
    | [] -> raise Not_found
    | x::p when fst x=k -> p
    | x::p -> x::(aux p)
  in try
     t.donnees.(hk) <- aux t.donnees.(hk) ;
     t.taille <- t.taille - 1 ;
     if t.taille < w/2 then rearrange_bis_dyn t
  with Not_found ->  ()
;;
```

La complexité dans les tables de hachage. Si l'on suppose que les clés se répartissent de manière à peu près équitable dans les différentes listes, on obtient une complexité amortie constante pour l'ajout, la suppression et la recherche dans les tables dynamiques. En effet, dans le cas où il n'y a pas réarrangement, la complexité est constante puisque le ratio $\frac{\text{taille de la table}}{\text{largeur de la table}}$ est inférieur à 2 ici. Une fois un réarrangement effectué (linéaire en le nombre d'éléments n présents dans la table), on peut procéder à $n/2$ suppressions ou n insertions sans réarrangement : en moyenne sur ces opérations (en comptant celle ayant nécessité le réarrangement) l'insertion ou la suppression ont pris un temps constant : la complexité *amortie* est donc constante.

Hypothèse de hachage uniforme. Il est important d'avoir des fonctions de hachage d'apparence aléatoire, pour que les opérations sur la table s'effectuent en temps constant (amorti). Les fonctions présentées ici sont très basiques : il en existe de bien meilleures, pour lesquelles on peut supposer que le hachage est uniforme. En fait, ces fonctions (qui produisent des suites d'une centaine de bits) sont si complexes qu'il est difficile de construire un couple de clés produisant le même haché [1]. On peut éventuellement composer ces fonctions avec des fonctions « type modulo » pour réduire la largeur de hachage suivant l'application.

Utilisation des fonctions de hachage. Outre la réalisation de dictionnaire, les fonctions de hachage sont également utilisées en cryptologie ; on peut s'en servir par exemple pour vérifier l'intégrité d'un document : il suffit de comparer son haché (par une fonction de hachage fixée) avec le haché du document originel. Ce principe est très conseillé lorsqu'on télécharge un fichier « critique » comme un fichier d'installation (fichier ISO) : il vaut mieux vérifier que le fichier n'a pas été corrompu pendant le transfert avant de procéder à l'installation. À titre d'exemple, le haché du fichier pdf du TP4 par la fonction de hachage `md5` (une des plus utilisées) est `2e33bfa3a5e875ae0cd0090a20de44ac` (un entier de 32 chiffres hexadécimaux). Avec un espace en plus dans le fichier, on obtient `721e800aca1a8d9e8b9d2ad15a6ce0ef` : on voit que `md5` paraît assez aléatoire !

1. Voir par exemple les fonctions des familles MD ou SHA.