TABLEAUX ASSOCIATIFS (DICTIONNAIRES)

Un tableau associatif (aussi appelé dictionnaire ou table d'association) est un type de données associant une valeur unique à une clé d'accès. Dans un tableau associatif, on doit pouvoir réaliser, au minimum, les opérations :

- ajout : association d'une nouvelle valeur à une nouvelle clé ;
- modification : association d'une nouvelle valeur à une ancienne clé :
- suppression : suppression d'une clé (et de la valeur associée), si elle existe ;
- recherche : détermination de la valeur associée à une clé, si elle existe.

Remarque. Les clés n'étant pas consécutives, le nombre de clés potentielles étant bien plus grand que le nombre de clefs réelles, il n'est pas possible d'utiliser un tableau T de valeurs et une fonction injective h : clé $\longmapsto$ indice , tel que $T[h(c)]$ soit la valeur associé à la clé c.

Pour représenter un tableau associatif, on peut envisager plusieurs structures de données :

1. **liste d'association**, ce qui est peu efficace puisque la recherche de la valeur associée à une clé est de complexité $O(n)$.
2. **arbre binaire de recherche équilibré**, selon un ordre défini sur les clés, la recherche de la valeur associée à une clé étant de complexité $O(\log(n))$.
   Cela peut être intéressant si on a besoin d'effectuer un parcours selon l'ordre des clés.
3. **table de hachage** (voir bon article WIKIPÉDIA), constituée par
   - un tableau (vecteur) T, indicé par des entiers consécutifs, de taille m très petit par rapport aux nombre M de clés potentielles ($m \ll M$), dont les cases (alvéoles) contiennent un ou plusieurs couples (clé, valeur) ou un moyen d'accès vers ces couples ;
   - une fonction de hachage h : clé $\longmapsto$ indice du tableau T, **non injective**, plusieurs couples (clé, valeur) pouvant partager le même indice (la même alvéole) du tableau (on dit dans ce cas qu'il y a **collision**) ;
   - une méthode de gestion des collisions,
     - en représentant l'ensemble des données qui partagent le même indice (la même alvéole) par
       - une liste de couples (clé, valeur), ce qui est la solution la plus simple,
       - un arbre binaire de recherche de couples (clé, valeur),
       - ...
     - en utilisant l'adressage ouvert : dans le cas d'une collision, les paires (clé,valeur) sont stockées dans d'autres alvéoles, déterminées par une méthode de sondage (voit détails sur WIKIPÉDIA).
     - ...

On s'intéresse ici à la mise en place de dictionnaires à l'aide de tables de hachage.
Pour résoudre les problèmes de coûts dus à la gestion des collisions, on essaie de faire un choix pertinent pour la dimension m du tableau T, la fonction de hachage h et la structure de représentation ou d'organisation des sous-ensembles de données qui sont en collision.

- le tableau T est la table de hachage, de dimension notée m
- la fonction h est la fonction de hachage (non injective)
- le nombre n de clés utilisées est appelé la taille du dictionnaire
- le rapport $\dfrac{\text{nombre de clés utilisées}}{\text{dimension de } T}$ $\left(= \dfrac{n}{m}\right)$ est le facteur de remplissage de T

1 Gestion des collisions

1.1 Utilisation de listes de collisions

Le tableau de hachage, T, est de taille m, faible devant le nombre n de données réelles, n étant très faible devant le nombre de clés potentielles M.

La case i du tableau T contient (un moyen d'accès à) la liste de collision, constituée des couples (clé, donnée) pour lesquels la clé k de la donnée est hachée en la valeur i (ide, telle que $h(k) = i$).

Remarque. Il n'y a pas de limite au nombre de données que l'on peut associer à la case i du tableau T (sauf celle de la capacité physique de stockage ...).

1.1.1 Conséquences d'un choix de la fonction h de hachage

1. Coûts structurels
   - En général le calcul de $h(k)$ se fait à l'aide d'une formule, dont le coût de calcul est un $O(1)$.
   - L'accès à la case i du tableau T se fait à un coût $O(1)$: chaque élément du tableau, qui est une adresse, est de longueur constante.
   - La recherche dans la liste de collision (de longueur n_i) associée à l'indice i de T se fait à un coût maximal de $O(n_i)$.
2. Evaluation des coûts d'accès
   - Dans le cas le plus défavorable, toutes les clés donnent le même indice (fonction de hachage constante) et la recherche d'une donnée dans un dictionnaire de taille n a un coût maximal qui est un $O(n)$.
   - Si la fonction de hachage répartit uniformément les indices (c'est à dire si chaque clé a autant de chance d'être hachée en un indice qu'en un autre), dans un dictionnaire de taille n, la longueur moyenne d'une liste de collision sera $a = \frac{n}{m}$ (facteur de remplissage de T) et le coût moyen de recherche d'une donnée sera un $O\left(\frac{n}{m}\right)$.

Dans la pratique, les clés ne sont pas uniformément réparties, ce qui perturbe la répartition des indices faite par la fonction de hachage. Une bonne fonction de hachage devrait tenir compte de la répartition statistique (si elle est connue) des clés utilisées, pour prétendre diminuer le coût moyen de recherche.

1.1.2 Exemples de fonctions de hachage pour des clés entières

1. Fonction de hachage par division : $h(k) = k \mod m$

   Ici, le choix de m (dimension de T) est important. Prendre pour m un nombre premier éloigné des puissances exactes de 2 est en général un bon choix.

2. Fonction de hachage par multiplication :

   $$h(k) = Ent\big(m * Frac(k * A)\big), \quad \text{avec } A \text{ constante réelle } (0 < A < 1)$$

   Ent : partie entière,
   $Frac$: partie fractionnaire.

   D'après Knuth, $A = \frac{\sqrt{5}-1}{2} = 0.6180339887\cdots$" serait un bon choix ...

Remarque. Pour des clés non entières, on peut, par bijection, transformer au préalable la clé en un entier. Par exemple, une chaîne de caractères pourrait être transformée en la valeur entière de son codage en base 2 (ce qui n'est pas une bonne méthode !).

1.2 Résolution des problèmes de collision par adressage ouvert

On suppose que N, nombre maximal de données réelles, est connu à l'avance (légèrement majoré) ou modifiable seulement de façon exceptionnelle et on prends un tableau T, où chaque case du tableau contient (un moyen d'accès à) une et une seule donnée valide ou invalide.

- La taille de T est $m = N$, nombre maximal de données prévues,
- une information particulière indiquera si la case i du tableau T contient une donnée valide ou non (adresse de donnée valide ou invalide, donnée avec un champ supplémentaire de validité, valeur de clé dans ou en dehors de l'univers des clés possibles, ...).

Comme l'univers des clés possibles est grand par rapport au nombre maximal de données réelles, avec des valeurs de clés non consécutives et peu prévisibles, il est difficile d'imaginer une fonction de hachage qui soit injective, avec un coût de calcul constant.

La fonction de hachage pourra donc associer, à des clés distinctes, le même indice du tableau et on se trouve confronté à un risque de collision.

1.2.1 Principe de gestion des collisions

- Lors de l'insertion, si à l'indice calculé, il y a déjà une donnée valide associée à une autre clé, c'est une collision et alors . . . on ira se placer ailleurs !
- Lors de la consultation, si à l'indice calculé,
  - il y a une donnée invalide, c'est que la clé fournie est invalide
  - la donnée est bien associée à la clé, on a trouvé la donnée cherchée
  - il y a déjà une donnée valide associée à une autre clé, c'est une collision et on ira voir ailleurs !
- Lors de la suppression, . . . même technique !

Il reste à définir le "ailleurs" et à s'assurer que l'on retrouve bien les données à l'aide de leurs clés.

1.2.2 Gestion effective des collisions

1. **Insertion**. On réalise l'insertion par sondage : à partir de la clé, on définit au fur et à mesure une suite d'indices possibles, jusqu'à ce que l'on trouve une case vide où insérer la donnée.

   Pour mettre en œuvre cette stratégie, on étend la fonction de hachage h en une nouvelle fonction (fonction de hachage étendue ou fonction de sondage) :

   $$hh : (\text{cle}, \text{nombre d'indices sondés}) \longmapsto \text{nouvel indice à sonder}$$

   telle que la séquence de sondage $\big(hh(cle,0), hh(cle,1), \cdots, hh(cle,m-1)\big)$ soit une permutation de l'ensemble $\{0,1,\cdots,m-1\}$ des indices du tableau T.

   Ainsi, si le tableau T n'est pas plein, on finira par découvrir un emplacement disponible.

   La séquence $\big(hh(cle,0), hh(cle,1), \cdots, hh(cle,\alpha)\big)$, lorsque $hh(cle,\alpha+1)$ est l'indice d'une donnée invalide, est la chaîne de collision de (associée à) la clé cle.

2. **Suppression** . On utilise la fonction de hachage étendue (ou de sondage) pour parcourir la chaîne de collision jusqu'à ce que l'on trouve la donnée à supprimer.

   La suppression ne peut pas se contenter d'invalider la donnée, puisque cela indiquerait la fin de la chaîne de collision et interdirait l'accès aux éléments suivants éventuels !

   On marquera la donnée comme supprimée, ce qui conduit à préciser, mieux que par "donnée valide" ou "donnée invalide", la nature de l'information de marquage :
   - "donnée valide"
   - "donnée inexistante" (fin de chaîne et position d'insertion)
   - "donnée supprimée" (la chaîne se poursuit et on a une position d'insertion possible)

   Remarque. Cela conduit à revoir la stratégie d'insertion, pour éviter les doublons tout en ré-utilisant le plus tôt possible les emplacements supprimés . . .

3. **Recherche (consultation)** ; On utilise la fonction de hachage étendue (ou de sondage) pour parcourir la chaîne de collision de cle, jusqu'à ce que l'on trouve la donnée associée à cle ou qu'on arrive en fin de chaîne (avec une donnée inexistante).

1.2.3 Exemples de fonctions de sondage

1. Sondage linéaire : $hh(cle,i) = \big(h(cle)+i\big) \mod m$.
   Inconvénients : Effet de grappe forte (longues séquences de cases occupées, ce qui augmente le temps de recherche moyen).
2. Sondage quadratique : $hh(cle,i) = \big(h(cle)+C_1*i+C_2*i^2\big) \mod m$.
   Inconvénients : Effets de grappe (mais faible).
3. Sondage par double hachage : $hh(cle,i) = \big(h_1(cle)+i*h_2(cle)\big) \mod m$ où
   - h_1 et h_2 sont deux fonctions de hachage,
   - $h_2(cle)$ est premier avec m (ce qui assure un parcours éventuel complet de T).

   Avantage : cette méthode répartit les données d'une façon presque uniforme, avec un coût moyen d'accès faible.

2 Réalisation d'un dictionnaire (exemple)

Une banque de notes d'un groupe de concours est un dictionnaire où les données sont les informations concernant un candidat.

Pour notre exemple, on supposera que

- les clés sont les numéros de TIPE, calculés d'après l'identité du candidat, dans l'intervalle [1 .. 99999],
- les données comprennent les champs "clé", ("validité",) "nom", "prénom", "filière", "résultats", ...).

L'univers U des clés possibles est donc $\{1, \cdots 99999\}$ et les clés réelles sont les numéros de TIPE (non consécutifs) des n candidats qui se sont inscrits à un concours ($n \ll 99999$) .

On traitera une version simplifiée, avec un dictionnaire concernant, par exemple, les élèves de l'option info du lycée (plus quelques élèves "bidon" pour faire masse), en prenant

- pour clé, le numéro d'ordre alphabétique dans la réunion des sections MP1, MP2, MP*, option info, (ou le numéro de TIPE s'il est connu),
- des données réduites aux champs : clé (integer), nom (string 40), filière (string 3 = "MP␣") complétés par un champ éventuel de validité (boolean ou code entier) si besoin.

On construira deux versions du dictionnaire :

1. par hachage, avec résolution des collisions par listes de collisions ;
2. par hachage, avec résolution des collisions par adressage ouvert.

2.1 Dictionnaire 1 (avec résolution des collisions par listes de collisions)

La table de hachage est un tableau `T1`, de taille m fixée, faible par rapport au nombre de clés.

Pour notre exemple, on prendra $m = 13$ (pour 40 à 50 candidats), une fonction de hachage par division ($h(\text{clef}) = \text{clef} \mod m$) et (mais cela revient au même en Caml) on considérera que le contenu de `T1`.(i) est la liste de collision des données dont la clé est hachée en l'indice i (plutôt qu'un pointeur vers cette liste).

2.1.1 Généralités (déclarations globales)

- Définir le type ("data") de donnée, ici un type ensemble, à champs modifiables en place :
  ```
  type data = {mutable cle:int; mutable nom : string; mutable filiere : string};;
  ```
- Définir `m` (on prendra ici $m = 13$).
- Définir `T1`, vecteur de listes de données, de taille m, initialement vide.
- Définir `h` fonction de hachage (ici, hachage par division : $h(\text{clef}) = \text{clef} \mod m$).

2.1.2 Exemples d'écriture

1. Test d'existence, selon une valeur de clé, dans une liste de type `data list` :

   ```
   let rec data_list_existe cle = function              (* version a *)
       []           -> false
     | {cle = k}::q -> (cle = k) or (data_list_existe cle q)
   ;;
   let rec data_list_existe cle = function              (* version b *)
       []                        -> false
     | {cle = k}::q when k = cle -> true
     | a::q                      -> data_list_existe cle q
   ;;
   (* data_list_existe : int -> data list -> bool = <fun> *)
   ```

   data_list_existe cle alist renvoie true si une donnée de clé cle se trouve dans la liste alist et false sinon.

   Remarque. J'exagère en donnant, presqu'en même temps, le même nom à deux objets ! Ces objets étant de nature différentes, l'écriture est correcte ! Si cela vous gène, vous pouvez changer les noms :

   ```
   let rec data_list_existe clef = function             (* version a' *)
       []           -> false
     | {cle = k}::q -> (clef = k) or (data_list_existe clef q)
   ;;
   ```

2. Recherche, selon une valeur de clé, dans une liste de type `data list` :

```
let rec data_list_fonctionbidon cle = function
   | [] -> failwith "Clé non trouvée."          (* provoque l'exception Failure s *)
   | {cle = k; nom = nom; filiere = filiere}::q when k = cle -> nom, filiere
(* | {cle = k} as d ::q when k = cle -> d.nom, d.filiere          (* identique *) *)
   | t::q -> ... ;;
(* data_list_fonctionbidon : int -> data list -> string * string *)
```

Remarque. Vous préférerez certainement nommer `clef` l'argument et écrire, avec des noms distincts :

```
    | {cle = k; nom = n; filiere = f}::q when k = clef -> n, f
ou  | {cle = k} as d ::q when k = clef -> d.nom, d.filiere
```

2.1.3 Ressources de base : primitives sur les listes de données (data list)

`data_list_insere : int -> string -> string -> data list -> data list` `A ECRIRE (3 lignes)`

data_list_insere cle nom filiere alist introduit la donnée {cle, nom, filiere} dans la liste alist.
Renvoie l'exception Failure "Clé déjà présente." si une donnée de même clé cle se trouve déjà dans la liste.

`data_list_supprime : int -> data list -> data list` `A ECRIRE (3 lignes)`

data_list_supprime cle alist supprime la donnée de clé cle dans la liste alist.
Renvoie l'exception Failure "Clé non trouvée." si il n'y a pas de donnée correspondant à la clé cle.

`data_list_recherche : int -> data list -> string * string` `A ECRIRE (3 lignes)`

data_list_recherche cle alist renvoie le couple (nom, filiere) correspondant à la clé cle dans la liste alist.
Renvoie l'exception Failure "Clé non trouvée." si il n'y a pas de donnée correspondant à la clé cle.

On disposera en outre de fonctions pré-écrites destinées aux tests et à la visualisation :

`data_list_voir : int -> int -> data list -> int` `(* non commentée, usage interne *)`

2.1.4 Primitives du dictionnaire

Exemple de gestion d'exception dans un bloc "`try ... with ... -> ...`" :

```
let T1_existe cle =
  try data_list_existe cle T1.(h cle)
  with exn -> raise exn
;;
(* T1_existe : int -> bool = <fun> *)
```

Si `data_list_existe cle T1.(h cle)` ne provoque pas d'exception, on finit ce qui est spécifié dans la partie `try` et sinon, on abandonne l'exécution pour traiter ce qui est spécifié par `with` (ici, on se contente de propager ("relever") la même exception).

`T1_insere : int -> string -> string -> unit` `A ECRIRE (2 lignes)`

T1_insere cle nom filiere introduit la donnée {cle, nom, filiere} dans le dictionnaire de table T1.
Renvoie l'exception Failure "Clé déjà présente." si une donnée de même clé existe déjà.

`T1_supprime : int -> unit` `A ECRIRE (2 lignes)`

T1_supprime cle supprime la donnée de clé cle du dictionnaire de table T1.
Renvoie l'exception Failure "Clé non trouvée." si il n'y a pas de donnée associée à cette clé.

`T1_recherche : int -> string * string` `A ECRIRE (2 lignes)`

T1_recherche cle renvoie le couple (nom, filiere) associée à la clé cle dans le dictionnaire de table T1.
Renvoie l'exception Failure "Clé non trouvée." si il n'y a pas de donnée associée à cette clé.

On disposera en outre de fonctions pré-écrites destinées aux tests et à la visualisation :

`T1_clear : unit -> unit` T1_clear () initialise le contenu de T1 à [ ].
`T1_voir : unit -> unit` T1_voir () affiche le contenu de T1 sur la fenêtre graphique de Caml.

2.1.5 Tests : insérer, rechercher, supprimer, visualiser ...

1. Insérer, dans le dictionnaire 1, les (pseudo) données préparées (elles sont dans la liste `LBide`), ce qui peut se faire en une instruction : "`do_list (fun (x,y,z) -> ...) LBide;;`" avec `...` à préciser ;
2. Insérer les données vous concernant dans le dictionnaire 1 ;
3. Effectuer des opérations classiques de recherche, suppression, insertion ... et visualiser, sur la fenêtre graphique, les conséquences dans le tableau T1 (`T1_voir ()`).

Tester signifie aussi qu'il faut chercher à ... planter le programme !

2.2 Dictionnaire 2 (avec résolution des collisions par adressage ouvert)

Comme le nombre de données réelles, après inscriptions, est stable et prévisible, cette technique est tout à fait justifiée, quitte à majorer légèrement la taille du tableau par rapport aux besoins réels.
Dans notre exemple (version simplifiée), on prendra $m = n = 47$ comme nombre maximal de données réelles et une fonction de sondage par double hachage :

$$hh(\text{clef}, i) = \big(h_1(\text{clef}) + i * h_2(\text{clef})\big) \mod m \quad \text{avec} \quad \begin{cases} h_1(\text{clef}) = \text{clef} \mod m \\ h_2(\text{clef}) = 1 + \big(\text{clef} \mod (m-1)\big) \end{cases}$$

2.2.1 Généralités (déclarations globales)

- Le type ("data") de donnée, à champs modifiables en place, est le même que précédemment :
  `type data = {mutable cle: int; mutable nom: string; mutable filiere: string};;`
  Par convention, ici (de façon simpliste, sans introduire de champs supplémentaire),
  - une donnée jamais validée est repérée par une valeur de clé égale à -1 ("fin de chaîne"),
  - une donnée supprimée est repérée par une valeur de clé égale à -2 ("trou dans une chaîne").
- Définir `n` et `m` ($n = 47, m = n$).
- Définir `T2`, vecteur de data, de taille m, initialement rempli de données jamais validées :
  `{cle = -1 ; nom = ""; filiere = "" }`.
- Définir les fonctions `h1` et `h2` de hachage auxiliaires.
- Définir la fonction de sondage `hh`.

2.2.2 Primitives du dictionnaire

Remarque. Un parcours par sondage se termine avec la rencontre de la clé précisée ou d'une clé égale à -1 (fin de chaîne utile) ou lorsque le numéro de sondage atteint la taille de la table (table pleine).

`T2_insere : int -> string -> string -> unit` A ECRIRE (10 lignes)

T2_insere cle nom filiere introduit la donnée {cle, nom, filiere} dans le dictionnaire de table T2.
Renvoie l'exception Failure "Clé déjà présente." si une donnée de même clé existe déjà.
Renvoie l'exception Failure "Table pleine." si la table T2 ne comporte plus d'emplacement disponible.

Remarque. Pour éviter les doublons, et à cause des suppressions éventuelles,

> l'insertion se fait au premier emplacement disponible (donnée supprimée ou fin de chaîne), à condition qu'une donnée de même clé ne soit pas présente dans le reste de la chaîne ;
>
> pour cela, en parcourant toute la chaîne, on repère au passage le premier emplacement disponible (s'il en existe un) et, en final, on insérera la nouvelle donnée à l'emplacement repéré
>
> (insérer systématiquement, une nouvelle donnée, en fin de chaîne de collision, serait un choix non conforme aux objectifs de réduction des coûts de consultation et risque de conduire à un blocage par manque de place en cas de suppressions et ajouts nombreux).

`T2_supprime : int -> unit` A ECRIRE (8 lignes)

T2_supprime cle supprime la donnée de clé cle du dictionnaire de table T2 en donnant à la clé enregistrée la valeur -2. (les autres champs resteront inchangés).
Renvoie l'exception Failure "Clé non trouvée." si il n'y a pas de donnée correspondant à cette clé.

`T2_recherche : int -> string * string` A ECRIRE (8 lignes)

T2_recherche cle renvoie le couple (nom, filiere) associé à la clé cle dans le dictionnaire de table T2.
Renvoie l'exception Failure "Clé non trouvée." si il n'y a pas de donnée correspondant à cette clé.

On disposera en outre de fonctions pré-écrites destinées aux tests et à la visualisation :

`T2_clear : unit -> unit` T2_clear () initialise le contenu de T2 à {cle = -1 ; nom = "" ; filiere = ""}.
`T2_voir : unit -> unit` T2_voir () affiche le contenu de T2 sur la fenêtre graphique de Caml.

2.2.3 Tests : de même que pour le premier dictionnaire ...

Références

Knutt	Sorting and Searching. - Addison Wesley, 1973
Gonnet	Handbook of Algorithms and Data Structures. - Addison Wesley, 1984
Cormen Lieserson Rivest	Introduction to Algorithms - MIT Press 1990

< $\mathscr{FIN}$ >

3 Squelette du programme (à compléter). Non exécutable !

```
let A_ECRIRE s = print_string s; print_newline ();;    (* Avertissement ! NE PAS MODIFIER *)
(* A_ECRIRE : string -> unit = <fun> *)
#open "graphics";;  (* utile pour visualisation *)
(*=====================================================================*)
(* 3.1. DICTIONNAIRE 1 : hachage avec collisions par liste de collision *)
(*=====================================================================*)
(* === 3.1.1. Données globales  ==== *)

type data = {mutable cle: int; mutable nom: string; mutable filiere: string};;

let m = 13;;
let T1 = make_vect m [];;
let h cle = cle mod m
;;
(* h : int -> int = <fun> *)

(* === 3.1.3. Exemples data_list         *)
let rec data_list_existe cle = function    (* version 2 *)
  | []           -> false
  | {cle = k}::q -> (cle = k) or (data_list_existe cle q)
;;
(* data_list_existe : int -> data list -> bool = <fun> *)

(* === 3.1.3. Ressources data_list === *)

let rec data_list_insere cle nom filiere = function x ->                    A_ECRIRE "3 lignes"
;;
(* data_list_insere: int -> string -> string -> data list -> data list =<fun>*)

let rec data_list_supprime cle = function x ->                              A_ECRIRE "3 lignes"
;;
(* data_list_supprime : int -> data list -> data list = <fun> *)

let rec data_list_recherche cle = function x ->                             A_ECRIRE "3 lignes"
;;
(* data_list_recherche : int -> data list -> string * string = <fun> *)

(*--- utilitaire d'affichage, sur la fenêtre graphique -----*)
let rec data_list_voir x y = ( function
    | []   -> y - 20
    | [a]  -> aux a; y - 20
    | a::q -> aux a;  data_list_voir x (y - 20) q  )
  where aux a =
    let s = (string_of_int a.cle) ^ " == " ^ a.nom ^ " == " ^ a.filiere
    in moveto x y; draw_string s;
;;
(* data_list_voir : int -> int -> data list -> int = <fun> *)

(* === 3.1.4. Primitives Dictionnaire 1 === *)
let T1_existe cle =  try data_list_existe cle T1.(h cle)
                     with exn -> raise exn
;;
(* T1_existe : int -> bool = <fun> *)

let T1_insere cle nom filiere =                                             A_ECRIRE "2 lignes"
and T1_supprime cle =                                                       A_ECRIRE "2 lignes"
and T1_recherche cle =                                                      A_ECRIRE "2 lignes"
;;
(* T1_insere : int -> string -> string -> unit = <fun> *)
(* T_supprime : int -> unit = <fun> *)
(* T1_recherche : int -> string * string = <fun> *)
```

```
(*------------ ressources prédéfinies --------*)
let T1_clear () = for i = 0 to (vect_length T1)-1 do T1.(i) <- [] done
and T1_voir () =
  clear_graph ();
  let x = 10 and y = ref (-30 + size_y ()) in
  for i = 0 to (vect_length T1)-1
  do moveto x !y; draw_string ((string_of_int i ) ^ " -> ");
     y := data_list_voir (x+60) !y T1.(i)
  done
;;
(* T1_clear : unit -> unit = <fun> *)  (* T1_voir : unit -> unit = <fun> *)

(* === 3.1.5. Test Dictionnaire 1 === *)
let LBide = (* Liste d'informations "bidon", pour faire masse ... *)
[ 2317,"Untel",  "MP1"; 1242,"Machin", "MP2"; 2537,"Truc",  "MP*"; 12511,"Bidule","MP1";
  5234,"Autre",  "MP2"; 7589,"Cestlui","MP*"; 4567,"Paslui","MP1";  1237,"Suite", "MP2";
   145,"Chapeau","MP*";  278,"Veston", "MP1"; 157, "Télév", "MP2";   453,"Cinema","MP*"  ];;

T1_clear ();;
(* >>> Insérer les éléments de LBide dans le dictionnaire 1                  A ECRIRE *)
T1_voir ();;

T1_existe 25;;
T1_insere 5412 "Certain" "MP1";;       T1_recherche 5412;;
T1_insere (5412+13) "Cestlui" "MP*";;  T1_recherche (5412+13);;  T1_supprime (5412+13);;
T1_insere 5417 "Lesuivant" "MP*";;     T1_insere (5417+13) "Ledernier" "MP2";;
T1_recherche (5417+13);; T1_voir () ;;

(* ==> Test extrême, pour un tableau T1 === *)
let rempliT1 k =
  for i = 0 to k do T1_insere (1+random__int 5000) ("numéro ("^(string_of_int i)^")") "Mp" done
;;
(* rempliT1 : int -> unit = <fun> *)

T1_clear ();;        rempliT1 20;;        T1_voir ();;
(*=================================================*)
(* 3.2. DICTIONNAIRE 2 : hachage par adressage ouvert *)
(*=================================================*)
(* === 3.2.1. Données globales === *)
(* type data = {mutable cle: int; mutable nom: string; mutable filiere: string};;*)
(*    définition data: même type que pour T1: déjà fait ci-dessus *)
(*    Rappel.  cle = -1 : donnée jamais validée ;  cle = -2 : donnée supprimée *)

let n = 47;; (* premier, taille maximale de données prévisibles *)
let m = n;;
let T2 = make_vect m {cle = -1; nom = ""; filiere = "" } ;;

let h1 cle = cle mod m and h2 cle = 1 + (cle mod (m-1)) and hh cle i = ( h1 cle + i * h2 cle ) mod m;;

(* === 3.2.2. Primitives Dictionnaire 1 === *)
let T2_insere cle nom filiere =
  let z = pos 0 (-1)      (* z = indice d'insertion, -1 (invalide) par défaut *)

  where rec pos i z =    (* i = numéro sondage, z = premier indice insertion *)
    if i >= vect_length T2                                          (* 5 lignes à ECRIRE *)
    then A_ECRIRE "1 ligne"
    else let j = hh cle i in
         match T2.(j).cle with
          | -1 -> A_ECRIRE "1 ligne"                          (* fin de chaine *)
          | -2 -> A_ECRIRE "1 ligne"                          (* donnée supprimée *)
          | k when k = cle -> failwith "Cette clé existe déjà."   (* *)
          | _              -> A_ECRIRE "1 ligne"                  (* poursuivre*)
  in if z = -1 then failwith "Table pleine."
               else T2.(z) <- A_ECRIRE "1 ligne"
;;
(* T2_insere : int -> string -> string -> unit = <fun> *)
```

```
let T2_supprime cle = aux 0
  where rec aux i =           (* i = numéro de sondage *)                    A_ECRIRE "7 lignes"
;;
(* T2_supprime : int -> unit = <fun> *)

let T2_recherche cle = aux 0
  where rec aux i =           (* i = numéro de sondage *)                    A_ECRIRE "7 lignes"
;;
(* T2_recherche : int -> string * string = <fun> *)

(*------------ ressources prédéfinies --------*)
let T2_clear () =
      for i = 0 to (vect_length T2)-1
      do T2.(i) <- {cle = -1; nom = ""; filiere = ""} done
and T2_voir () =
  clear_graph ();
  let x = ref 0 and y = ref (-10 + size_y ()) in
  for i = 0 to (vect_length T2)-1
  do if i mod 2 = 0
     then begin x := 10; y := !y - 20 end
          else x := 410;
     let s = (string_of_int T2.(i).cle) ^ " == " ^ T2.(i).nom ^ " == " ^ T2.(i).filiere  in
     moveto !x      !y; draw_string ((string_of_int i ) ^ " -> ");
     moveto (!x+60) !y; draw_string s
  done
;;
(* T2_clear : unit -> unit = <fun> *)
(* T2_voir : unit -> unit = <fun> *)

(* === 3.2.3. Test Dictionnaire 2 === *)
T2;;
T2_clear ();;
(* >>>>>>  Insérer les éléments de LBide dans le dictionnaire 2              A ECRIRE *)
T2_voir ();;

T2_insere 5412 "Certain" "mp1";;         T2_recherche 5412;;       T2_supprime 5412;;
T2_insere (5412+13) "Cestlui" "mp*";;    T2_recherche (5412+13);;  T2_supprime (5412+13);;
T2_insere 5417 "Lesuivant" "mp*";;       T2_recherche 5417;;       T2_supprime 5417;;
T2_insere (5417+13) "Ledernier" "mp2";;  T2_recherche (5417+13);;  T2_supprime (5417+13);;

(* ==> Test extrême, pour un tableau T2 plein *)
let rempliT2 () =
  for i = 0 to vect_length T2 -1
  do T2_insere (1 + random__int 5000) ("numéro ("^(string_of_int i)^")") "mp" done
;;
(* rempliT2 : unit -> unit = <fun> *)

T2_clear ();; rempliT2 ();; T2_voir ();;

(*================================*)
(*=== Fin Hache-S.ml (squelette)   *)
(*================================*)
```

< $\mathscr{FIN}$ (énoncé + squelette) >

4 TP Tables de Hachage : Corrigé

```
(*-------------------------------------------------------
 | Hache-C.ml                                          |  CORRIGE
 -------------------------------------------------------*)
#open "graphics";;  (* utile pour visualisation *)

(* === 1. === *)
(* === 2. === *)
(* === 3. === *)
(*=====================================================================*)
(* 3.1. DICTIONNAIRE 1 : hachage avec collisions par liste de collision *)
(*=====================================================================*)
(* === 3.1.1. Données globales === *)
type data = {mutable cle: int; mutable nom: string; mutable filiere: string};;
let m = 13;;
let T1 = make_vect m [];;

let h cle = cle mod m
;;
(* h : int -> int = <fun> *)

(* === 3.1.3. Exemples data_list === *)
let rec data_list_existe cle = function       (* version a *)
    []           -> false
  | {cle = k}::q -> (cle = k) or (data_list_existe cle q)
;;
(* data_list_existe : int -> data list -> bool = <fun> *)

(* let rec data_list_existe cle = function    (* version b *)
       []                       -> false
     | {cle = k}::q when k = cle -> true
     | a::q                     -> data_list_existe cle q
   ;;
*)

(* === 3.1.3. Ressources data_list === *)
let rec data_list_insere cle nom filiere = function
  | []                       -> [{cle = cle; nom = nom; filiere = filiere}]
  | {cle = k}::q when k = cle -> failwith "Cette clé existe déjà."
  | a::q                     -> a::data_list_insere cle nom filiere q
;;
(* data_list_insere: int -> string -> string -> data list -> data list =<fun>*)

let rec data_list_supprime cle = function
  | []                       -> failwith "Clé non trouvée."
  | {cle = k}::q when k = cle -> q
  | a::q                     -> a::data_list_supprime cle q
;;
(* data_list_supprime : int -> data list -> data list = <fun> *)

let rec data_list_recherche cle = function
  | []                             -> failwith "Clé non trouvée."
  | {cle = k} as d ::q when k = cle -> d.nom, d.filiere
  | a::q                           -> data_list_recherche cle q
;;
(* data_list_recherche : int -> data list -> string * string = <fun> *)

(* let rec data_list_recherche cle = function   (* version 2 *)
     | [] -> failwith "Clé non trouvée."
     | {cle = k; nom = nom; filiere = filiere}::q when k = cle -> nom, filiere
     | a::q -> data_list_recherche cle q
   ;;
*)
```

```
let rec data_list_voir x y =  ( function              (* utilitaire d'affichage *)
     | []   -> y-20
     | [a]  -> aux a; y - 20
     | a::q -> aux a; data_list_voir x (y - 20) q
   )
  where aux a = let s = (string_of_int a.cle) ^ " == " ^ a.nom ^ " == " ^ a.filiere
                in moveto x y; draw_string s;
;;
(* data_list_voir : int -> int -> data list -> int = <fun> *)

(* === 3.1.4. Primitives Dictionnaire 1 === *)
let T1_existe cle =
  try data_list_existe cle T1.(h cle)
  with exn -> raise exn
;;
(* T1_existe : int -> bool = <fun> *)

let T1_insere cle nom filiere =
  try let i = h cle in T1.(i) <- data_list_insere cle nom filiere T1.(i)
  with exn -> raise exn
;;
(* T1_insere : int -> string -> string -> unit = <fun> *)

let T1_supprime cle =
  try let i = h cle in T1.(i) <- data_list_supprime cle T1.(i)
  with exn -> raise exn
;;
(* T_supprime : int -> unit = <fun> *)

let T1_recherche cle =
  try data_list_recherche cle T1.(h cle)
  with exn -> raise exn
;;
(* T1_recherche : int -> string * string = <fun> *)

(*------------ ressources prédéfinies --------*)
let T1_clear () = for i = 0 to (vect_length T1)-1 do T1.(i) <- [] done
;;
(* T1_clear : unit -> unit = <fun> *)

let T1_voir () =
  clear_graph ();
  let x = 10 and y = ref (-30 + size_y ()) in
  for i = 0 to (vect_length T1)-1
  do  (* y := !y - 20; *)
      moveto x !y; draw_string ((string_of_int i ) ^ " -> ");
      y := data_list_voir (x+60) !y T1.(i)
  done
;;
(* T1_voir : unit -> unit = <fun> *)

(* === 3.1.5. Test Dictionnaire 1 === *)
(* Liste d'information "bidon", pour faire masse ... *)
let LBide = [ 2317,  "Untel",     "MP1"; 1242,"Machin",    "MP2"; 2537,"Truc",   "MP*";
              12511, "Bidule",    "MP1"; 5234,"Autre",     "MP2"; 7589,"Cestlui","MP*";
    4567,  "Cestpaslui","MP1"; 1237,"Suite",     "MP2"; 145, "Chapeau","MP*";
    278,   "Veston",    "MP1"; 157, "Télévision","MP2"; 453, "Cinema", "MP*"; 625, "Album", "MP1" ];;

(* ---> Insérer les éléments de LBide dans le dictionnaire 1 : ----*)
T1_clear ();;
do_list (fun (x,y,z) -> T1_insere x y z) LBide;;
T1_voir ();;
```

```
T1_existe 25;;
T1_insere 5412 "Certain" "MP1";;            T1_recherche 5412;; T1_existe 5412;; T1_supprime 5412;;
T1_insere (5412+13) "Cestlui" "MP*";;       T1_recherche (5412+13);;       T1_supprime (5412+13);;
T1_insere (5412+13+13) "Encoreun" "MP2";;   T1_recherche (5412+13+13);;    T1_supprime 5438;;
T1_insere (5412+13+13+13) "Unautre" "MP1";; T1_recherche (5412+13+13+13);; T1_supprime 5451;;
T1_insere 5417 "Lesuivant" "MP*";;          T1_recherche 5417;;            T1_supprime 5417;;
T1_insere (5417+13) "Ledernier" "MP2";;     T1_recherche (5417+13);;       T1_supprime (5417+13);;

(* ==> Test extrême, pour un tableau T1 === *)
let rempliT1 k =
  for i = 0 to k do T1_insere (1 + random__int 5000) ("numéro ("^(string_of_int i)^")") "Mp" done
;;
(* rempliT1 : int -> unit = <fun> *)

T1_clear ();;  rempliT1 20;;  T1_voir ();;

(*=====================================================*)
(* 3.2. DICTIONNAIRE 2 : hachage par adressage ouvert *)
(*=====================================================*)
(* === 3.2.1. Données globales === *)
(* définition data: même type que dans le cas de T1: Déjà fait ci-dessus *)
(* type data = {mutable cle: int; mutable nom: string; mutable filiere: string};; *)
(*    Rappel : cle = -1 pour donnée jamais validée; cle = -2 pour donnée supprimée  *)

let n = 47;; (* premier, taille maximale de données prévisibles *)
let m = n;;

let T2 = make_vect m {cle = -1; nom = ""; filiere = "" } ;;

let h1 cle = cle mod m;; let h2 cle = 1 + (cle mod (m-1));;
let hh cle i = ( h1 cle + i * h2 cle ) mod m;;

(* === 3.2.2. Primitives Dictionnaire 2 === *)
let T2_insere cle nom filiere =
  let z = pos 0 (-1)    (* indice d'insertion, -1 (invalide) par défaut *)

  where rec pos i z =   (* i = numéro de sondage, z = premier indice d'insertion possible, -1 sinon *)
    if i >= vect_length T2
    then z
    else let j = hh cle i in
         match T2.(j).cle with
           | -1           -> if z = -1 then j else z              (* en fin de chaîne *)
           | -2           -> pos (i+1) (if z = -1 then j else z)  (* passage sur donnée supprimée *)
           | k when k=cle -> failwith "Cette clé existe déjà."    (* *)
           | _            -> pos (i+1) z                          (* poursuivre *)

  in if z = -1 then failwith "Table pleine."
               else T2.(z) <- {cle = cle; nom = nom; filiere = filiere}
;;
(* T2_insere : int -> string -> string -> unit = <fun> *)

let T2_supprime cle = aux 0
  where rec aux i =           (* i = numéro de sondage *)
    if i >= vect_length T2
    then failwith "Clé non trouvée."
    else let j = hh cle i in
         match T2.(j).cle with
           | -1           ->  failwith "Clé non trouvée."
           | k when k=cle -> T2.(j).cle <- -2
           | _            ->  aux (i+1)
;;
(* T2_supprime : int -> unit = <fun> *)
```

```
let T2_recherche cle = aux 0
  where rec aux i =          (* i = numéro de sondage *)
    if i >= vect_length T2
    then failwith "Clé non trouvée."
    else let j = hh cle i in
         match T2.(j).cle with
           | -1           ->  failwith "Clé non trouvée."
           | k when k=cle -> T2.(j).nom, T2.(j).filiere
           | _            ->  aux (i+1)
;;
(* T2_recherche : int -> string * string = <fun> *)

(*-----------*)
let T2_clear () =
  for i = 0 to (vect_length T2)-1 do T2.(i) <- {cle = -1; nom = ""; filiere = ""} done
;;
(* T2_clear : unit -> unit = <fun> *)

let T2_voir () =
  clear_graph ();
  let x = ref 0 and y = ref (-10 + size_y ()) in
  for i = 0 to (vect_length T2)-1
  do if i mod 2 = 0
     then begin x := 10; y := !y - 20 end else x := 410;

     let s =   (string_of_int T2.(i).cle) ^ " == " ^ T2.(i).nom ^ " == " ^ T2.(i).filiere in
     moveto !x      !y; draw_string ((string_of_int i ) ^ " -> ");
     moveto (!x+60) !y; draw_string s
  done
;;
(* T2_voir : unit -> unit = <fun> *)

(* === 3.2.3. Test Dictionnaire 2 === *)
(*---> Insérer les éléments de LBide dans le dictionnaire 2 : ---- *)
T2_clear ();;
do_list (fun (x,y,z) -> T1_insere x y z) L1;;
T2_voir ();;

T2_insere 5412 "Certain" "MP1";;        T2_recherche 5412;;         T2_supprime 5412;;
T2_insere (5412+13) "Cestlui" "MP*";;   T2_recherche (5412+13);;    T2_supprime (5412+13);;
T2_insere 5417 "Lesuivant" "MP*";;      T2_recherche 5417;;         T2_supprime 5417;;
T2_insere (5417+13) "Ledernier" "MP2";; T2_recherche (5417+13);;    T2_supprime (5417+13);;
T2_voir ();;

(* ==> Test extrême, pour un tableau T2 plein *)
let rempliT2 () =
  for i = 0 to vect_length T2 -1
  do T2_insere (1 + random__int 5000) ("numéro ("^(string_of_int i)^")") "mp"
  done
;;
(* rempliT2 : unit -> unit = <fun> *)

T2_clear ();;  rempliT2 ();;  T2_voir ();;

(*================================*)
(*=== Fin Hache-C.ml (corrigé)   *)
(*================================*)
```

< $\mathscr{FIN}$ (énoncé + squelette + corrigé) >